



IMSL[®] C Stat Library

Version 2020.0

PERFORCE

www.perforce.com



Copyright 1970-2020 Rogue Wave Software, Inc., a Perforce company.

Visual Numerics, IMSL, and PV-WAVE are registered trademarks of Rogue Wave Software, Inc., a Perforce company.

IMPORTANT NOTICE: Information contained in this documentation is subject to change without notice. Use of this document is subject to the terms and conditions of a Rogue Wave Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability.

ACKNOWLEDGMENTS

Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Perforce Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation

ROGUE WAVE MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL PERFORCE SOFTWARE, INC. BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

IMSL by Perforce
<https://www.imsl.com>

Contents

Introduction

IMSL C Stat Library	2
Organization of the Documentation	3
Finding the Right Function	5
Naming Conventions	6
Getting Started and the imsls.h file	7
Error Handling, Underflow, Overflow, and Document Examples.	8
Memory Allocation for Output Arrays	9
Printing Results	10
Missing Values	11
Passing Data to User-Supplied Functions	12
Return Values from User-Supplied Functions	14
Thread Safe Usage	15
OpenMP Usage	16
Vendor Supplied Libraries Usage	18
C++ Usage	19
Matrix Storage Modes	22

Basic Statistics

Functions	23
Usage Notes	24
simple_statistics	25
empirical_quantiles	35
normal_one_sample	39
normal_two_sample	44
table_oneway	56
table_twoway	61
sort_data	67
ranks	76

Regression

Functions.....	84
Usage Notes	85
regressors_for_glm	98
regression	108
regression_summary.....	124
regression_prediction.....	133
hypothesis_partial	145
hypothesis_scph.....	152
hypothesis_test	157
regression_selection	165
regression_stepwise	176
poly_regression	187
poly_prediction.....	196
nonlinear_regression	206
nonlinear_optimization.....	219
Lnorm_regression	230
pls_regression.....	249

Correlation and Covariance

Functions.....	264
Usage Notes	265
covariances.....	266
partial_covariances	275
pooled_covariances.....	281
robust_covariances	288

Analysis of Variance and Designed Experiments

Functions.....	298
Usage Notes	299
anova_oneway	313
ancovar	325
anova_factorial	339
anova_nested	349

anova_balanced	363
crd_factorial	375
rcbd_factorial	386
latin_square	396
lattice	405
split_plot	422
split_split_plot	436
strip_plot	452
strip_split_plot	463
homogeneity	484
multiple_comparisons	491
false_discovery_rates	497
yates	505

Categorical and Discrete Data Analysis

Functions	517
Usage Notes	518
contingency_table	519
exact_enumeration	533
exact_network	536
categorical_glm	543
logistic_regression	560
logistic_reg_predict	576

Nonparametric Statistics

Functions	585
Usage Notes	586
sign_test	587
wilcoxon_sign_rank	591
noether_cyclical_trend	596
cox_stuart_trends_test	600
tie_statistics	606
wilcoxon_rank_sum	609
kruskal_wallis_test	615
friedmans_test	618

cochran_q_test	624
k_trends_test	628

Tests of Goodness of Fit

Functions	633
Usage Notes	634
chi_squared_test	635
shapiro_wilk_normality_test	648
lilliefors_normality_test	651
chi_squared_normality_test	654
kolmogorov_one	657
kolmogorov_two	661
multivar_normality_test	665
ad_normality_test	670
cvm_normality_test	673
randomness_test	676

Time Series and Forecasting

Functions	688
Usage Notes	690
arma	696
max_arma	709
arma_forecast	716
arima	724
regression_arima	737
auto_uni_ar	748
seasonal_fit	755
ts_outlier_identification	764
ts_outlier_forecast	774
auto_arima	783
auto_parm	801
bayesian_seasonal_adj	813
box_cox_transform	822
difference	827
autocorrelation	833

crosscorrelation	839
multi_crosscorrelation	846
partial_autocorrelation	857
lack_of_fit	861
estimate_missing	865
hw_time_series	873
garch	883
kalman	889
vector_autoregression	901

Multivariate Analysis

Functions	919
Usage Notes	920
dissimilarities	923
cluster_hierarchical	928
cluster_number	934
cluster_k_means	939
principal_components	945
factor_analysis	952
discriminant_analysis	973

Survival and Reliability Analysis

Functions	998
Usage Notes	999
kaplan_meier_estimates	1000
prop_hazards_gen_lin	1007
survival_glm	1022
survival_estimates	1046
nonparam_hazard_rate	1053
life_tables	1063

Probability Distribution Functions and Inverses

Functions	1069
Usage Notes	1071
binomial_cdf	1074

binomial_pdf	1076
geometric_cdf	1078
geometric_inverse_cdf	1080
geometric_pdf	1082
hypergeometric_cdf	1084
hypergeometric_pdf	1087
poisson_cdf	1089
poisson_pdf	1092
discrete_uniform_cdf	1094
discrete_uniform_inverse_cdf	1096
discrete_uniform_pdf	1098
beta_cdf	1100
beta_inverse_cdf	1102
non_central_beta_cdf	1104
non_central_beta_inverse_cdf	1107
non_central_beta_pdf	1110
bivariate_normal_cdf	1113
chi_squared_cdf	1115
chi_squared_inverse_cdf	1118
complementary_chi_squared_cdf	1120
non_central_chi_sq	1123
non_central_chi_sq_inv	1127
non_central_chi_sq_pdf	1129
exponential_cdf	1132
exponential_inverse_cdf	1134
exponential_pdf	1136
F_cdf	1138
F_inverse_cdf	1141
complementary_F_cdf	1143
non_central_F_pdf	1146
non_central_F_cdf	1150
complementary_non_central_F_cdf	1153
non_central_F_inverse_cdf	1157

gamma_cdf	1160
gamma_inverse_cdf	1163
lognormal_cdf	1165
lognormal_inverse_cdf	1167
lognormal_pdf	1169
normal_cdf	1171
multivariate_normal_cdf	1174
normal_inverse_cdf	1182
t_cdf	1184
t_inverse_cdf	1187
complementary_t_cdf	1189
non_central_t_cdf	1193
non_central_t_inv_cdf	1196
non_central_t_pdf	1198
pareto_cdf	1201
pareto_pdf	1203
max_likelihood_estimates	1205

Random Number Generation

Functions	1215
Usage Notes	1218
random_binomial	1223
random_geometric	1226
random_hypergeometric	1228
random_logarithmic	1231
random_neg_binomial	1233
random_poisson	1236
random_uniform_discrete	1238
random_general_discrete	1240
discrete_table_setup	1245
random_beta	1251
random_cauchy	1254
random_chi_squared	1256
random_exponential	1258

random_exponential_mix	1260
random_gamma	1263
random_lognormal	1266
random_normal	1269
random_stable	1272
random_student_t	1275
random_triangular	1277
random_uniform	1279
random_von_mises	1282
random_weibull	1284
random_general_continuous	1287
continuous_table_setup	1290
random_normal_multivariate	1294
random_orthogonal_matrix	1298
random_mvar_from_data	1301
random_multinomial	1305
random_sphere	1308
random_table_twoway	1311
random_mvar_gaussian_copula	1314
random_mvar_t_copula	1319
canonical_correlation	1325
random_order_normal	1331
random_order_uniform	1334
random_arma	1337
random_npp	1342
random_permutation	1346
random_sample_indices	1348
random_sample	1350
random_option	1354
random_option_get	1356
random_seed_get	1357
random_substream_seed_get	1359
random_seed_set	1361

random_table_set	1362
random_table_get	1363
random_GFSR_table_set	1365
random_GFSR_table_get	1366
random_MT32_init	1370
random_MT32_table_get	1371
random_MT32_table_set	1374
random_MT64_init	1375
random_MT64_table_get	1376
random_MT64_table_set	1379
faure_next_point	1380

Data Mining

Functions	1385
Data Mining Usage Notes	1388
apriori	1394
aggr_apriori	1401
write_apriori_itemsets	1412
write_association_rules	1413
free_apriori_itemsets	1414
free_association_rules	1415
Decision Trees – An Overview	1416
decision_tree	1419
decision_tree_predict	1455
decision_tree_print	1462
decision_tree_free	1467
bagged_trees_free	1468
gradient_boosting	1469
Genetic Algorithms – An Overview	1492
ga_chromosome	1501
ga_copy_chromosome	1507
ga_clone_chromosome	1508
ga_individual	1510
ga_copy_individual	1515

ga_clone_individual	1516
ga_mutate	1518
ga_decode	1520
ga_encode	1521
ga_free_individual	1522
ga_population	1523
ga_random_population	1530
ga_copy_population	1539
ga_clone_population	1540
ga_grow_population	1542
ga_merge_population	1544
ga_free_population	1546
genetic_algorithm	1547
Naive Bayes – An Overview	1569
naive_bayes_trainer	1571
naive_bayes_classification	1590
nb_classifier_free	1599
nb_classifier_write	1600
nb_classifier_read	1605
Neural Networks – An Overview	1611
Multilayer Feedforward Neural Networks	1622
mlff_network_init	1632
mlff_network	1634
mlff_network_free	1646
mlff_network_write	1647
mlff_network_read	1653
mlff_initialize_weights	1659
mlff_network_trainer	1674
mlff_network_forecast	1686
mlff_classification_trainer	1693
mlff_pattern_classification	1719
scale_filter	1734
time_series_filter	1741

time_series_class_filter	1744
unsupervised_nominal_filter	1749
unsupervised_ordinal_filter	1753
kohonenSOM_trainer	1758
kohonenSOM_forecast	1769
Support Vector Machines – An Overview	1771
support_vector_trainer	1773
support_vector_classification	1791
svm_classifier_free	1796

Printing Functions

Functions	1797
write_matrix	1798
page	1805
write_options	1807

Utilities

Functions	1811
output_file	1813
version	1817
error_options	1819
error_code	1826
error_type	1828
error_message	1829
initialize_error_handler	1831
set_user_fcn_return_flag	1834
free	1839
fopen	1841
fclose	1843
ascii_read	1844
omp_options	1863
machine (integer)	1865
machine (float)	1868
data_sets	1871
mat_mul_rect	1875

permute_vector	1880
permute_matrix	1882
impute_missing	1885
binomial_coefficient.....	1895
beta	1897
beta_incomplete	1900
log_beta	1902
gamma.....	1904
gamma_incomplete.....	1907
log_gamma	1910
ctime.....	1913

Reference Material

Contents	1915
User Errors	1915

Appendix A References	1919
------------------------------------	------

Appendix B Alphabetical Summary of Functions	1954
---	------

Product Support

Contacting IMSL Support	1985
-------------------------------	------

Index

Introduction

Table of Contents

IMSL C Stat Library	2
Organization of the Documentation	3
Finding the Right Function	5
Naming Conventions	6
Getting Started and the imsls.h file	7
Error Handling, Underflow, Overflow, and Document Examples	8
Memory Allocation for Output Arrays	9
Printing Results	10
Missing Values	11
Passing Data to User-Supplied Functions	12
Return Values from User-Supplied Functions	14
Thread Safe Usage	15
OpenMP Usage	16
Vendor Supplied Libraries Usage	18
C++ Usage	19
Matrix Storage Modes	22

IMSL C Stat Library

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

Organization of the Documentation

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

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

- **Section Name:** Usually, the common root for the type *float* and type *double* versions of the function.
- **Purpose:** A statement of the purpose of the function.
- **Synopsis:** The form for referencing the function 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.

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.

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
<i>float</i>	imsls_f_
<i>double</i>	imsls_d_
<i>int</i>	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:

IMPORTANT: Do not choose a name beginning with “`imsls_`” in any combination of uppercase or lowercase characters.

Getting Started and the `imsls.h` file

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.

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*.

Error Handling, Underflow, Overflow, and Document Examples

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 `IMSL_FATAL`, `IMSL_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.

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

Memory Allocation for Output Arrays

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

```
IMSL_S_RETURN_USER, float a[ ]
```

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

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

```
IMSL_S_ANOVA_TABLE_USER, float anova_table[ ] (Output)
```

```
IMSL_S_ANOVA_TABLE, float **p_anova_table (Output)
```

specify two mutually exclusive optional arguments. If the first option is chosen, the ANOVA table is stored in the user-provided array `anova_table`.

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

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 14, [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 [ims1s_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);

int 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(NULL, 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) contains the value 0.5
 * *((float*)data+1) contains the value 0.2886751.
 */
float cdf_w_data(float x, void *data)
{
    float mean, std, z;
    mean = *((float*)data);
    std = *((float*)data+1);

    z = (x-mean)/std;
```

```
    return(imsls_f_normal_cdf(z));  
}
```

Return Values from User-Supplied Functions

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

In addition to the techniques described below, it is also possible to instruct the IMSL C Stat Library to return control to the calling program in case an unrecoverable error occurs within a user-supplied function. See function [imsls_set_user_fcn_return_flag](#) for a description of this feature.

Example

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

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

int 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[])
{
    float result, exparg;
    exparg = theta[1]*(x[0] - 8);
    /* check that argument to exp does not get too large */
    if (exparg > 10.) {
        result = 22000.;
    } else {
        result = theta[0] + (0.49 - theta[0])*exp(exparg);
    }
}
```

Thread Safe Usage

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

Error Handling

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 the file to which 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.

OpenMP Usage

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



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

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

OpenMP is used by the Library in these main ways:

1. To implement thread safety within the C Stat Library.
2. To speed up computationally intensive functions by exploiting data parallelism in their processing.
3. To give users more control of scheduling by using the "schedule(runtime)" clause for the parallelized for-loops. The scheduling option chosen, set by using the OMP_SCHEDULE environment variable, can significantly affect the performance of user's program depending on the workload of the system during execution. If OMP_SCHEDULE is not set, the default behavior depends on implementation. Please refer to OpenMP specifications on schedule type and chunk.
4. To set and control the number of threads to use for parallel region and nested parallel region by using the OMP_NUM_THREADS and OMP_NESTED environment variables. If OMP_NUM_THREADS and OMP_NESTED are not set, the default behavior depends on the implementation. Thus, all computing resources may be used, affecting other applications' performance on the system. Please refer to OpenMP specifications for more information.
5. To parallelize the evaluation of user-supplied functions in routines that use them, e.g. in the genetic algorithm routines.

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

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

Specifications of the OpenMP standards are provided at (<http://www.openmp.org/specifications/>).

Vendor Supplied Libraries Usage

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



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

C++ Usage

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

The function `imsls_f_chi_squared_test` performs a chi-squared goodness-of-fit test, using a user defined cumulative distribution (CDF) function. For C++ usage the user defined function is defined as a member function of the abstract class `CdfFunction` defined as follows:

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

class CdfFunction
{
public:
    virtual float cdf(float x) = 0;
};
```

The function `imsls_f_chi_squared_test` is wrapped as the C++ class `ChiSquaredTest`. This implementation uses the optional argument, `IMSLS_FCN_W_DATA`, to call `local_function` which in turn calls the method `cdf` to evaluate the user defined CDF function. For simplicity, this implementation only wraps a single optional argument, `IMSLS_CHI_SQUARED`, the chi-squared test statistic. More could be included in a similar manner.

```
#include <imsl.h>

class ChiSquaredTest
{
private:
    int    m_nObservations, m_nCategories;
public:
    float  m_chi_squared;
    ChiSquaredTest(int nObservations, int nCategories);
    float  test(CdfFunction *Cdf, float *x);
};

static float local_function(float x, void *data)
{
    CdfFunction *Cdf = (CdfFunction*)data;
    return Cdf->cdf(x);
}

ChiSquaredTest:: ChiSquaredTest (int nObservations, int nCategories)
{
    m_nObservations = nObservations;
    m_nCategories = nCategories;
}

float ChiSquaredTest::test(CdfFunction *Cdf, float *x)
{
    float  result;
```

```

    result = imsls_f_chi_squared_test(
        NULL, m_nObservations, m_nCategories, x,
        IMSLS_FCN_W_DATA, local_function, Cdf,
        IMSLS_CHI_SQUARED, &m_chi_squared,
        0);
    if (imsls_error_type() >= 3)
    {
        throw imsls_error_message();
    }
    return result;
}

```

To use `ChiSquaredTest` the user defined CDF function must be defined as the method `cdf` in a class that extends `FcnCdfFunction`. The following class, `NormalCdf`, defines this as the normal cdf:

```

class NormalCdf : public CdfFunction
{
public:
    NormalCdf();
    float cdf(float x);
};

NormalCdf::NormalCdf()
{
}

float NormalCdf::cdf(float x)
{
    return imsls_f_normal_cdf(x);
}

```

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

```

int main()
{
    imsls_error_options(
        IMSLS_SET_PRINT, IMSLS_FATAL, 0,
        IMSLS_SET_PRINT, IMSLS_TERMINAL, 0,
        IMSLS_SET_STOP, IMSLS_FATAL, 0,
        IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
        0);
    imsls_omp_options(IMLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    int nCategories = 10;
    int nObservations = 1000;
    imsls_random_seed_set(123457);
    float *x = imsls_f_random_normal(nObservations, 0);

    NormalCdf *normalCdf = new NormalCdf();
    ChiSquaredTest *chiSquaredTest =
        new ChiSquaredTest(nObservations, nCategories);
    float p_value = chiSquaredTest->test(normalCdf, x);
}

```

```
printf("p-value    = %g\n", p_value);  
printf("chi-squared = %g\n", chiSquaredTest->m_chi_squared);  
}
```

Output

```
p-value =0.154603  
chi-squared =13.1806
```

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};
```

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*.

Basic Statistics

Functions

Simple Summary Statistics

Univariate summary statistics	simple_statistics	25
Computes empirical quantiles	empirical_quantiles	35
Mean and variance inference for a single normal population	normal_one_sample	39
Inferences for two normal populations	normal_two_sample	44

Tabulate, Sort, and Rank

Tally observations into a one-way frequency table	table_oneway	56
Tally observations into a two-way frequency table	table_twoway	61
Sort data with options to tally cases into a multi-way frequency table	sort_data	67
Ranks, normal scores, or exponential scores	ranks	76

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 [ims1s_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 $14 \times n_variables$. The columns of this matrix correspond to the columns of `x`, and the rows contain the following statistics:

	Statistic
0	mean
1	variance
2	standard deviation
3	coefficient of skewness
4	coefficient of excess (kurtosis)

	Statistic
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_IDO, int ido,
    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_IDO, *int ido* (Input)

Processing option.
The argument `ido` must be one of 0, 1, 2, or 3. If `ido = 0` (the default), all of the observations are input during one invocation. If `ido = 1, 2, or 3`, blocks of rows of the data can be processed sequentially in separate invocations of `imsls_f_simple_statistics`; with this option, it is not a requirement that all observations be memory resident, thus enabling one to handle large data sets.

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 <code>n_observations</code> observations of <code>x</code> will be performed.
2	This is an intermediate invocation; updating for the <code>n_observations</code> observations of <code>x</code> will be performed.
3	This is the final invocation of this function. Updating for the data in <code>x</code> and wrap-up computations are performed. Workspace is released. No further invocations of <code>imsls_f_simple_statistics</code> with <code>ido</code> greater than 1 should be made without first invoking <code>imsls_f_simple_statistics</code> with <code>ido = 1</code> .

Default: `ido = 0`

`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_MEDIAN` or `IMSLS_MEDIAN_AND_SCALE` can be specified only when `ido` is equal to 0.

`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 \mathbf{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 $14 \times n_variables$. If `IMSLS_MEDIAN` is specified, the matrix is $15 \times n_variables$. If `IMSLS_MEDIAN_AND_SCALE` is specified, the matrix is $17 \times n_variables$.

Description

For the data in each column of \mathbf{x} , `ims1s_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 \mathbf{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

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

Skewness

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

Excess or Kurtosis

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

Minimum

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

Maximum

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

Range

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

Coefficient of Variation

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

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

$$\frac{\text{MAD}}{\phi^{-1}(3/4)}$$

where $\phi^{-1}(3/4) \approx 0.6745$ is the inverse of the standard norm

istribution 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).

Examples

Example 1

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

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

    simple_statistics = 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

Example 2

Continuing with [Example 1](#) data, the example below invokes the `imsls_f_simple_statistics` function using values of `IDO` greater than 0.

```
#include <imsls.h>

#define N_VARIABLES          5
#define N_OBSERVATIONS_BLOCK_1  2
#define N_OBSERVATIONS_BLOCK_2  8
#define N_OBSERVATIONS_BLOCK_3  3

int main()
{
    float      *simple_statistics;
    float      x1[] = { 7., 26., 6., 60., 78.5,
                       1., 29., 15., 52., 74.3};
    float      x2[] = {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};
    float      x3[] = { 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_BLOCK_1,
        N_VARIABLES, x1,
        IMSLS_IDO, 1, 0);

    simple_statistics = imsls_f_simple_statistics(N_OBSERVATIONS_BLOCK_2,
        N_VARIABLES, x2,
        IMSLS_IDO, 2, 0);

    simple_statistics = imsls_f_simple_statistics(N_OBSERVATIONS_BLOCK_3,
        N_VARIABLES, x3,
        IMSLS_IDO, 3, 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	659.816	111.792	763.434	616.688

Warning Errors

IMSL5_ROW_OF_X_CONTAINED_NAN	At least one row of "x" contained NaN (a missing value).
IMSL5_VAR_IN_X_CONTAINED_NAN	At least one observation for a variable in "x" contained NaN (a missing value). Missing observations were excluded from calculations for those variables.
IMSL5_CONSTANT_OBSERVATIONS	The observations on variable(s) are constant.
IMSL5_LESS_THAN_TWO_VALID_OBS	Fewer than two valid observations are present. The corresponding statistics are set to NaN (not a number), (except for the mean, which is not correct if no valid observations).
IMSL5_VARIANCE_UNDERFLOW	The variance for this variable underflows. Therefore, the variance and standard deviation are set to 0, and the skewness and kurtosis are set to NaN (not a number)
IMSL5_NEGATIVE_VARIANCE	The variance is negative for the variable. The corresponding confidence limits for the variance are set to NaN (not a number).
IMSL5_NOT_ENOUGH_OBSERVATIONS	Fewer than two valid observations are present for the variable. The corresponding statistics are set to NaN (not a number), (except for the mean, which is not correct if no valid observations are present, or is correct if one observation is present)
IMSL5_MIN_GREATER_THAN_MAX	The maximum value is less than the minimum value. The corresponding statistics are set to NaN (not a number).
IMSL5_MAX_LESS_THAN_MIN	The maximum value is less than the minimum value. The corresponding statistics are set to NaN (not a number).

IMSLS_SUM_OF_WEIGHTS_ZERO	The sum of the weights for variable is zero. The statistics, except for the <code>minima</code> , <code>maxima</code> , <code>ranges</code> and <code>counts</code> , are set to NaN (not a number).
IMSLS_ZERO_SUM_OF_WEIGHTS	The sum of the weights is zero. The statistics, except for the <code>minima</code> , <code>maxima</code> , <code>ranges</code> and <code>counts</code> , are set to NaN (not a number).
IMSLS_LESS_THAN_TWO_VALID_OBS	Fewer than two valid observations are present. The corresponding statistics are set to NaN (not a number), (except for the mean, which is not correct if no valid observations).
IMSLS_FOURTH_ORDER_UNDERFLOW	Since the range of variable is very small, the fourth order moment for this variable underflows. Therefore, the kurtosis is set to NaN (not a number).
IMSLS_HIGH_ORDER_UNDERFLOW	Since the range of variable <code>%(11)</code> is very small, the higher order moments for this variable underflow. Therefore, the skewness and kurtosis are set to NaN (not a number).
IMSLS_CHI_SQUARED_STAT_ERROR	An error occurred in determining the chi-squared statistic. The lower confidence limit for the variance is set to NaN (not a number).

Fatal Errors

IMSLS_BAD_IDO_6	<code>"ido" = #</code> . Initial allocations must be performed by invoking the function with <code>"ido" = 1</code> .
IMSLS_BAD_IDO_7	<code>"ido" = #</code> . A new analysis may not begin until the previous analysis is terminated by invoking the function with <code>"ido" = 3</code> .
IMSLS_BAD_N_VARIABLES	<code>"n_variables" = #</code> . The number of variables must be the same in separate function invocations.

empirical_quantiles

Computes empirical quantiles.

Synopsis

```
#include <imsls.h>

float *imsls_f_empirical_quantiles(int n_observations, float x[], int n_qprop,
    float qprop[], ..., 0)
```

The type *double* function is `imsls_d_empirical_quantiles`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `x[]` (Input)
An array of length `n_observations` containing the data.

int `n_qprop` (Input)
Number of empirical quantiles requested.

float `qprop[]` (Input)
An array of length `n_qprop` containing the desired quantile proportions. Each value must lie in the interval (0,1).

Return Value

The function `imsls_f_empirical_quantiles` returns an array of length `n_qprop` containing the empirical quantiles corresponding to the input proportions in `qprop`.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_empirical_quantiles(int n_observations, float x[], int n_qprop,
    float qprop[],
```

```
IMSLS_N_MISSING, int *n_miss,  
IMSLS_XLO, float **xlo,  
IMSLS_XLO_USER, float xlo[ ],  
IMSLS_XHI, float **xhi,  
IMSLS_XHI_USER, float xhi[ ],  
IMSLS_RETURN_USER, float p_q[ ],  
0)
```

Optional Arguments

`IMSLS_N_MISSING, int *n_miss` (Output)

The number of missing values, if any, in `x`.

`IMSLS_XLO, float **xlo` (Output)

An array of length `n_qprop` containing the largest element of `x` less than or equal to the desired quantile.

`IMSLS_XLO_USER, float xlo[]` (Output)

Storage for `xlo` provided by the user. See `IMSLS_XLO` above.

`IMSLS_XHI, float **xhi` (Output)

An array of length `n_qprop` containing the smallest element of `x` greater than or equal to the desired quantile.

`IMSLS_XHI_USER, float xhi[]` (Output)

Storage for `xhi` provided by the user. See `IMSLS_XHI` above.

`IMSLS_RETURN_USER, float p_q[]` (Output)

A user-allocated array of length `n_qprop`. Upon completion `p_q` contains the empirical quantiles corresponding to the input proportions in `qprop`.

Description

The function `imsls_f_empirical_quantiles` determines the empirical quantiles, as indicated in the vector `qprop`, from the data in `x`. `imsls_f_empirical_quantiles` first checks to see if `x` is sorted; if `x` is not sorted, the routine does either a complete or partial sort, depending on how many order statistics are required to compute the quantiles requested.

This function returns the empirical quantiles and, for each quantile, the two order statistics from the sample that are at least as large and at least as small as the quantile. For a sample of size n , the quantile corresponding to the proportion p is defined as

$$Q(p) = (1 - f)x_{(j)} + fx_{(j+1)}$$

where $j = \lfloor p(n + 1) \rfloor$, $f = p(n + 1) - j$, and $x_{(j)}$ is the j -th order statistic, if $1 \leq j < n$; otherwise, the empirical quantile is the smallest or largest order statistic.

Example

In this example, five empirical quantiles from a sample of size 30 are obtained. Notice that the 0.5 quantile corresponds to the sample median. The data are from Hinkley (1977) and Velleman and Hoaglin (1981). They are the measurements (in inches) of precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
#include <imsls.h>
#include <stdio.h>
int main(){
    float x[30] = {
        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 qprop[5] = {
        0.01, 0.5, 0.9, 0.95, 0.99};
    float *p_xlo, *p_xhi, *p_q;
    int i;

    p_q = imsls_f_empirical_quantiles(30, x, 5, qprop,
        IMSLS_XLO, &p_xlo,
        IMSLS_XHI, &p_xhi,
        0);
    printf("        Smaller  Empirical  Larger\n");
    printf("Quantile  Datum    Quantile  Datum\n");
    for(i=0; i<5; i++){
        printf("  %4.2f   %7.2f   %7.2f   %7.2f\n",
            qprop[i], p_xlo[i], p_q[i], p_xhi[i]);
    }
}
```

Output

	Smaller	Empirical	Larger
Quantile	Datum	Quantile	Datum
0.01	0.32	0.32	0.32

0.50	1.43	1.47	1.51
0.90	3.00	3.08	3.09
0.95	3.37	3.99	4.75
0.99	4.75	4.75	4.75

normal_one_sample

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

Synopsis

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

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

The type *double* function is `imsls_d_normal_one_sample`.

Required Arguments

int `n_observations` (Input)
Number of observations.

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

Return Value

The mean of the sample.

Synopsis with Optional Arguments

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

```
float imsls_f_normal_one_sample (int n_observations, float x[ ],
```

```
    IMSLS_CONFIDENCE_MEAN, float confidence_mean,
```

```
    IMSLS_CI_MEAN, float *lower_limit, float *upper_limit,
```

```
    IMSLS_STD_DEV, float *std_dev,
```

```
    IMSLS_T_TEST, int *df, float *t, float *p_value,
```

```
    IMSLS_T_TEST_NULL, float mean_hypothesis_value,
```

```
    IMSLS_CONFIDENCE_VARIANCE, float confidence_variance,
```

```

IMSLS_CI_VARIANCE, float *lower_limit, float *upper_limit,
IMSLS_CHI_SQUARED_TEST, int *df, float *chi_squared, float *p_value,
IMSLS_CHI_SQUARED_TEST_NULL, float variance_hypothesis_value,
0)

```

Optional Arguments

IMSLS_CONFIDENCE_MEAN, *float confidence_mean* (Input)

Confidence level (in percent) for two-sided interval estimate of the mean. Argument **confidence_mean** must be between 0.0 and 100.0 and is often 90.0, 95.0, or 99.0. For a one-sided confidence interval with confidence level c (at least 50 percent), set **confidence_mean** = $100.0 - 2.0 \times (100.0 - c)$. If **IMSLS_CONFIDENCE_MEAN** is not specified, a 95-percent confidence interval is computed.

IMSLS_CI_MEAN, *float *lower_limit, float *upper_limit* (Output)

Argument **lower_limit** contains the lower confidence limit for the mean, and argument **upper_limit** contains the upper confidence limit for the mean.

IMSLS_STD_DEV, *float *std_dev* (Output)

Standard deviation of the sample.

IMSLS_T_TEST, *int *df, float *t, float *p_value* (Output)

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

IMSLS_T_TEST_NULL, *float mean_hypothesis_value* (Input)

Null hypothesis value for t test for the mean.
Default: **mean_hypothesis_value** = 0.0

IMSLS_CONFIDENCE_VARIANCE, *float confidence_variance* (Input)

Confidence level (in percent) for two-sided interval estimate of the variances. Argument **confidence_variance** must be between 0.0 and 100.0 and is often 90.0, 95.0, 99.0. For a one-sided confidence interval with confidence level c (at least 50 percent), set **confidence_variance** = $100.0 - 2.0 \times (100.0 - c)$. 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

$$\sigma_0^2$$

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>
#include <stdio.h>

int 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\n", 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>
#include <stdio.h>

int 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_INTERMEDIATE_RESULTS, float stats[ ],
IMSLS_UNION, float stats1[ ], float stats2[ ],
IMSLS_FINAL_RESULTS, float final_stats[ ],
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

NOTE: The following three optional arguments allow the analysis to be applied to subsets of the original data sets and then later combined for final results. These optional arguments may be useful when analyzing data sets too large to fit into memory, and also allow subsets of the data to be analyzed in separate threads and later combined for final results.

IMSLS_INTERMEDIATE_RESULTS, *float stats[]* (Input/Output)

Array of length 25 containing intermediate results. On input, **stats** contains intermediate statistics about a previous function invocation. When invoking the function the first time, set all **stats** elements to 0.0. On output, **imsls_f_normal_two_sample** combines the results on the current data sets and the intermediate statistics in **stats**.

This optional argument can be applied to separate blocks of data when physical memory cannot hold the entire data sets.

Note that when **IMSLS_INTERMEDIATE_RESULTS** optional argument is used, **imsls_f_normal_two_sample** function outputs are valid only if **IMSLS_FINAL_RESULTS** optional argument is specified to calculate the final statistics. See [Example 3](#).

Default: **stats** = NULL.

IMSLS_UNION, *float stats1[], float stats2[]* (Input)

stats1 and **stats2** are arrays of length 25 containing the intermediate results about previous computations. **stats1** and **stats2** are the variables provided to the **IMSLS_INTERMEDIATE_RESULTS** optional argument in previous function invocations.

imsls_f_normal_two_sample combines the results on the current data sets and the intermediate statistics in **stats1** and **stats2**. **stats1** and **stats2** can be NULL. See [Example 3](#).

This option would typically be used in conjunction with the **IMSLS_INTERMEDIATE_RESULTS** option to process a large data set using separate threads or compute nodes. For example, a data set could be split into two subsets, where each subset of data is passed into a separate thread or compute node and processed through **imsls_f_normal_two_sample** with the **IMSLS_INTERMEDIATE_RESULTS** option. The output from each thread is then saved and input to a final call of **imsls_f_normal_two_sample** using option **IMSLS_UNION** and **IMSLS_FINAL_RESULTS**.

Default: **stats1** = NULL and **stats2** = NULL.

IMSLS_FINAL_RESULTS, *float final_stats[]* (Output)

Array of length 25 containing the final statistics. See [Example 3](#).

Elements of `final_stats` are:

<code>index</code>	<code>final_stats[i]</code>
0	Mean of the first sample.
1	Mean of the second sample.
2	Variance of the first sample.
3	Variance of the second sample.
4	Number of observations in the first sample.
5	Number of observations in the second sample.
Note: <code>final_stats[6]</code> through <code>final_stats[13]</code> depend on the assumption of equal variances.	
6	Pooled variance.
7	t value, assuming equal variances.
8	Probability of a larger t in absolute value, assuming normality, equal means, and equal variances.
9	Degrees of freedom assuming equal variances.
10	Lower confidence limit for the mean of the first population minus the mean of the second, assuming equal variances.
11	Upper confidence limit for the mean of the first population minus the mean of the second, assuming equal variances.
12	Lower confidence limit for the common variance.
13	Upper confidence limit for the common variance.
Note: <code>final_stats[14]</code> through <code>final_stats[18]</code> use approximations that do not depend on an assumption of equal variances.	
14	t value, assuming unequal variances.
15	Approximate probability of a larger t in absolute value, assuming normality, equal means, and unequal variances.
16	Degrees of freedom assuming unequal variances, for Satterthwaite's approximation.
17	Approximate lower confidence limit for the mean of the first population minus the mean of the second, assuming equal variances.
18	Approximate upper confidence limit for the mean of the first population minus the mean of the second, assuming equal variances.
19	F value (greater than or equal to 1.0).

<code>index</code>	<code>final_stats[i]</code>
20	Probability of a larger F in absolute value, assuming normality and equal variances.
21	Lower confidence limit for the ratio of the variance of the first population to the second.
22	Upper confidence limit for the ratio of the variance of the first population to the second.
23	Number of missing values of first sample.
24	Number of missing values of second sample.

`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

$$\sigma^2 = \sigma_0^2 \text{ where } \sigma^2$$

is the common, or pooled, variance, and

$$\sigma_0^2$$

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 and denominator 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_f_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:

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

and

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

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:

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

The t statistic is as follows:

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

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

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

The test statistic is

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

where

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

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

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

Inferences about Variances

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

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

Examples

Example 1

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

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

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

int 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>
#include <stdio.h>

int 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
```

Example 3

The same data is used for this example as for the initial example. This example illustrates the use of the `IMSL_S_INTERMEDIATE_RESULTS`, `IMSL_S_UNION`, and `IMSL_S_FINAL_RESULTS` optional arguments with "x1" and "x2" divided into three sub-groups. Since there are more "x2" values than "x1" values, `n1_observations` is set to zero on later calls to the function.

This example demonstrates how the analysis can be applied to subsets of the original data sets and then later combined for final results. These techniques may be useful when analyzing data sets too large to fit into memory, and also allow subsets of the data to be analyzed in separate threads (though this example does not show the use of separate threads) and later combined for final results.

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

#define N1_OBSERVATIONS 7
#define N2_OBSERVATIONS 9

int main()
{
    int n1, n2, i;
    float diff_means, pooled_variance;
    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
    };
    float stats1[25], stats2[25], final_stats[25];

    /* Initialize variables. */
    for (i = 0; i < 25; i++) {
        stats1[i] = 0.0;
        stats2[i] = 0.0;
        final_stats[i] = 0.0;
    }

    /*
    ** Bring in first group of observations on x1 and x2.
    ** Save intermediate results into variable, stat1.
    **
    ** Total second group:
    **     n1_observations = 3, n2_observations = 3
    **
    */

    /* First call using:  n1_observations = 2, n2_observations = 2 */
    n1 = 2;
    n2 = 2;
    imsls_f_normal_two_sample(n1, x1, n2, x2,
        IMSLS_INTERMEDIATE_RESULTS, stats1,
        0);
```

```
/* Second call using: n1_observations = 1, n2_observations = 1 */
n1 = 1;
n2 = 1;
imsls_f_normal_two_sample(n1, &x1[2], n2, &x2[2],
    IMSLS_INTERMEDIATE_RESULTS, stats1,
    0);

/*
** Bring in second group of observations on x1 and x2.
** Save intermediate results into variable, stat2.
**
** Total second group:
**     n1_observations = 4, n2_observations = 4.
*/
n1 = 4;
n2 = 4;
imsls_f_normal_two_sample(n1, &x1[3], n2, &x2[3],
    IMSLS_INTERMEDIATE_RESULTS, stats2,
    0);

/*
** Bring in third group of observations on x1 and x2
** and combine the results in variables, stats1 and stats2,
** from the first and second groups.
**
** Total third group:
**     n1_observations = 0, n2_observations = 2.
*/
n1 = 0;
n2 = 2;
diff_means = imsls_f_normal_two_sample(n1, x1, n2, &x2[7],
    IMSLS_UNION, stats1, stats2,
    IMSLS_FINAL_RESULTS, final_stats,
    IMSLS_POOLED_VARIANCE, &pooled_variance,
    0);

/* Print results */
printf("x1_mean - x2_mean = %5.2f\n", diff_means);
printf("pooled variance   = %5.2f\n", pooled_variance);
}
```

Output

```
x1_mean - x2_mean = -50.48
pooled variance   = 434.63
```

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[],
```

```
IMSL_S_RETURN_USER, float table[ ],
0)
```

Optional Arguments

IMSL_S_DATA_BOUNDS, *float *minimum, float *maximum* (Output)

If none is specified or if **IMSL_S_DATA_BOUNDS** is specified, **n_intervals** intervals of equal length are used with the initial interval starting with the minimum value in **x** and the last interval ending with the maximum value in **x**. The initial interval is closed on the left and right. The remaining intervals are open on the left and closed on the right. When **IMSL_S_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 $(\text{maximum} - \text{minimum})/\text{n_intervals}$.

or

IMSL_S_KNOWN_BOUNDS, *float lower_bound, float upper_bound* (Input)

If **IMSL_S_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

$$\frac{\text{upper_bound} - \text{lower_bound}}{\text{n_intervals} - 2}$$

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

or

IMSL_S_CUTPOINTS, *float cutpoints[]* (Input)

If **IMSL_S_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

IMSL_S_CLASS_MARKS, *float class_marks[]* (Input)

If **IMSL_S_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>

int main()
{
    int    n_intervals=10;
    int    n_observations=30;
    float  *table;
    float  x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
                 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
                 0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
                 1.89, 0.90, 2.05};
    table = 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 $(-\infty, 0.5]$, $(0.5, 1.0]$, ..., $(4.0, 4.5]$, and $(4.5, \infty]$.

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

```

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

int main()
{
    int      n_intervals=10;
    int      n_observations=30;
    double   *table;
    double   x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47,
                  1.43, 3.37, 2.20, 3.00, 3.09, 1.51, 2.10,
                  0.52, 1.62, 1.31, 0.32, 0.59, 0.81, 2.81,
                  1.87, 1.18, 1.35, 4.75, 2.48, 0.96, 1.89,
                  0.90, 2.05};
    double   cutpoints[] = {0.5, 1.0, 1.5, 2.0, 2.5,
                          3.0, 3.5, 4.0, 4.5};
    table = 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 **ny** (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 `IMSL_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 `IMSL_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 `IMSL_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 `IMSL_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 `IMSL_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 $\mathbf{cx}[1] - \mathbf{cx}[0]$ in the \mathbf{x} direction and $\mathbf{cy}[1] - \mathbf{cy}[0]$ in the \mathbf{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 \mathbf{x} in this example are the same as those used in the examples for `imsls_f_table_oneway`. The data for \mathbf{y} were created by adding small integers to the data in \mathbf{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>

int 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>

int 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>

int 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>
int 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 ($i = 0, 1, \dots, n_keys - 1$), 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).

`IMSLI_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.

`IMSLI_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 `IMSLI_LIST_CELLS`.

`IMSLI_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`.

`IMSLI_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 `IMSLI_N`.

Description

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

Sorting

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

- For $i = 0, 1, \dots, n_{\text{observations}} - 2$,
 $x[i][\text{indices_keys}[0]] \leq x[i+1][\text{indices_keys}[0]]$
- For $k = 1, \dots, n_{\text{keys}} - 1$, if $x[j][\text{indices_keys}[j]] = x[i+1][\text{indices_keys}[j]]$ for $j = 0, 1, \dots, k - 1$, then $x[j][\text{indices_keys}[k]] = x[i+1][\text{indices_keys}[k]]$

The observations also can be sorted in descending order.

The rows of \mathbf{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 \mathbf{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 \mathbf{x} , but performs computations only for the variables (columns) in the first `n_keys` columns of \mathbf{x} (Exception: see optional argument `IMSLI_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 `IMSLI_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 `IMSLI_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 `list_cells` array 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 \mathbf{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
```

```

int 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[0][0], 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
int 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>

int 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 $\text{abs}(x[i] - x[j])$ 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 \mathbf{x} . If $\mathbf{x}[i]$ has the smallest value among the values in \mathbf{x} and there is no other element in \mathbf{x} with this value, then `ranks[i] = 1`. If both $\mathbf{x}[i]$ and $\mathbf{x}[j]$ have the same smallest value, the output value depends on the option used to break ties.

Argument	Result
IMSLS_AVERAGE_TIE	<code>ranks[i] = ranks[j] = 1.5</code>
IMSLS_HIGHEST	<code>ranks[i] = ranks[j] = 2.0</code>

Argument	Result
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 ($1 \leq r_i \leq n$, 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 $\mathbf{x}[i]$ equals $\mathbf{x}[j]$ (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:

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

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

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

    ranks = 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

int main()
{
    float      fuzz_value=0.0, score[4][N_OBSERVATIONS], *ranks;
    float      x[] = {
        0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
        3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
        1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
        4.75, 2.48, 0.96, 1.89, 0.90, 2.05
    };

    char      *row_labels[] = {"Blom", "Tukey", "Van der Waerden",
        "Expected Value"};

    /* Blom scores using largest ranks */
    /* for ties */
    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

Regression

Functions

Multivariate Linear Regression—Model Fitting

Generates regressors for a general linear model	regressors_for_glm	98
Fits a multivariate linear regression model	regression	108

Multivariate Linear Regression—Statistical Inference and Diagnostics

Produces summary statistics for a regression model	regression_summary	124
Computes predicted values, confidence intervals, and diagnostics	regression_prediction	133
Construction of a completely testable hypothesis	hypothesis_partial	145
Sums of cross products for a multivariate hypothesis	hypothesis_scph	152
Tests for the multivariate linear hypothesis	hypothesis_test	157

Variable Selection

All best regressions	regression_selection	165
Stepwise regression	regression_stepwise	176

Polynomial and Nonlinear Regression

Fits a polynomial regression model	poly_regression	187
Computes predicted values, confidence intervals, and diagnostics	poly_prediction	196
Fits a nonlinear regression model	nonlinear_regression	206
Fits a nonlinear regression model using Powell's algorithm	nonlinear_optimization	219

Alternatives to Least Squares Regression

LAV, Lpnorm, and LMV criteria regression	Lnorm_regression	230
Performs partial least squares (PLS) regression	pls_regression	249

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

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \quad i = 1, 2, \dots, n$$

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

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

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; $\beta_0, \beta_1, \dots, \beta_k$ 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_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 \mathbf{y} , and the independent variables are input in array \mathbf{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 $(x_1, x_2, \dots, x_k, y)^T (x_1, x_2, \dots, x_k, y)$.

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

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

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; $\beta_0, \beta_1, \dots, \beta_k$ 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:

	INTCEP	
<code>IMSLS_NO_INTERCEPT</code>	0	An intercept is not in the model.
<code>IMSLS_INTERCEPT (default)</code>	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:

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
$\beta_0 + \beta_1x_1$	1	0	
$\beta_0 + \beta_1x_1 + \beta_2x_1^2$	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
$\beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_1x_2$	1	0	
$\mu + \alpha_i + \beta x_{1i} + \beta_1 x_{1i}$	1	1	2

	n_effects	n_var_effects	Indices_effects
$\beta_0 + \beta_1x_1$	1	1	0
$\beta_0 + \beta_1x_1 + \beta_2x_1^2$	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
$\beta_0 + \beta_1x_1 + \beta_2x_2 + \beta_3x_1x_2$	3	1, 1, 2	0, 1, 0, 1
$\mu + \alpha_i + \beta x_{1i} + \beta_1 x_{1i}$	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

$$1 - R_j^2(1, 2, \dots, j-1)$$

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

$$R_j(1, 2, \dots, j-1)$$

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

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

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 's 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 $y = X\beta + \epsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors with $\text{rank}(X) = r$, 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`):

Regression	DFR = AOV[0]	SSR = AOV[3]	MSR = AOV[6]	AOV[8]	AOV[9]
Error	DFE = AOV[1]	SSE = AOV[4]	$s^2 = \text{AOV}[7]$		
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:

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

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:

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

The error sum of squares is given as follows:

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

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, $H_0: \beta_1 = \beta_2 = \dots = \beta_k = 0$, 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. Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

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

$$R_a^2 = (\text{anova_table}[11])$$

are expressed as a percentage and are defined as follows:

$$R^2 = 100(SSR/SST) = 100(1 - SSE/SST)$$

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

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

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

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

`coef_t_tests`

Two-dimensional matrix containing the regression coefficient vector β 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:

`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 $y = X\beta + \epsilon$, 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

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

where $W = \text{diag}(w_1, w_2, \dots, w_n)$ and $(X^T W X)^{-}$ denotes a generalized inverse of $X^T W X$. 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

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

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

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

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

The jackknife residual is defined as

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

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

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

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

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

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

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

Hoaglin and Welsch (1978) suggest that DFFITS greater than

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

is large.

Transformations

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

$$(x_1, x_2, x_1^2, x_2^2, x_1x_2)$$

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 '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, L_p 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 `IMSL_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 L_p norm ($p \leq 1$), is given by option `IMSLS_METHOD_LLQ`. 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_LLQ` allows the user to try several choices for $p \geq 1$ by simply changing the input value of p in the calling program. The CPU time decreases as p gets larger. Generally, choices of p between 1 and 2 are of interest. However, the L_p 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.

Function `imsls_f_pls_regression` provides an alternative method which is useful when there are many inter-related regression variables and relatively few observations. `imsls_f_pls_regression` finds linear combinations of the predictor variables that have highest covariance with Y .

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 `IMSLX_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, or

    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:

<code>dummy_method</code>	Method
<code>IMSLS_ALL</code>	The n indicator variables are the dummy variables (default).
<code>IMSLS_LEAVE_OUT_LAST</code>	The dummies are the first $n - 1$ indicator variables.
<code>IMSLS_SUM_TO_ZERO</code>	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, \dots, a_n . From this classification variable, `imsls_f_regressors_for_glm` creates n indicator variables. For $k = 1, 2, \dots, n$, we have

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

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 $A_k = I_k (k = 1, 2, \dots, n)$. For `dummy_method = IMSLS_LEAVE_OUT_LAST`, the dummy variables are $A_k = I_k (k = 1, 2, \dots, n - 1)$. For `dummy_method = IMSLS_SUM_TO_ZERO`, the dummy variables are $A_k = I_k - I_n (k = 1, 2, \dots, n - 1)$. 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

$$A_1, A_2, \dots, A_{m_1}$$

and

$$B_1, B_2, \dots, B_{m_2}$$

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

$$\begin{aligned} A \otimes B &= (A_1, A_2, \dots, A_{m_1}) \otimes (B_1, B_2, \dots, B_{m_2}) \\ &= (A_1 B_1, A_1 B_2, \dots, A_1 B_{m_2}, A_2 B_1, A_2 B_2, \dots, \\ &\quad A_2 B_{m_2}, \dots, A_{m_1} B_1, A_{m_1} B_2, \dots, A_{m_1} B_{m_2}) \end{aligned}$$

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 $A \otimes B \otimes X_1 X_2$.

Remarks

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

```
n_class = 2
n_continuous = 1
n_effects = 7
n_var_effects = (1, 1, 2, 1, 2, 2, 3)
indices_effects = (0, 1, 0, 1, 2, 0, 2, 1, 2, 0, 1, 2)
```

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

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

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 \mathbf{x} , the second effect corresponds to the second column of \mathbf{x} , etc. A second order model corresponding to the columns (variables) of \mathbf{x} is generated if `IMSLS_MODEL_ORDER` with `model_order = 2` is specified.

There are

$$NEF = n_class + 2 * n_continuous + \binom{NVAR}{2}$$

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

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

effects correspond to the two-variable interactions.

- Let the data matrix $\mathbf{x} = (A, B, X_1)$, 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 $\mathbf{x} = (A, X_1, X_2)$, 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 $\mathbf{x} = (X_1, A, X_2)$ (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>
#include <stdio.h>

int 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

$$\mu + \alpha_i + \beta_j + \gamma_{ij} + \delta x_{ij} + \zeta_i x_{ij} + \eta_j x_{ij} + \theta_{ij} x_{ij} \quad i = 1, 2; j = 1, 2, 3$$

where

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

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

int 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", "Gamma1", "Gamma2",
        "Delta", "Zeta1", "Eta1", "Eta2", "Theta1", "Theta2"
    };

    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	Gammal1	Gammal2	Delta
1	1.00	1.00	0.00	1.00	0.00	1.11
2	1.00	1.00	0.00	1.00	0.00	2.22
3	1.00	1.00	0.00	1.00	0.00	3.33
4	1.00	0.00	1.00	0.00	1.00	1.11
5	1.00	0.00	1.00	0.00	1.00	2.22
6	1.00	0.00	1.00	0.00	1.00	3.33
7	1.00	0.00	0.00	0.00	0.00	1.11
8	1.00	0.00	0.00	0.00	0.00	2.22
9	1.00	0.00	0.00	0.00	0.00	3.33
10	0.00	1.00	0.00	0.00	0.00	1.11
11	0.00	1.00	0.00	0.00	0.00	2.22
12	0.00	1.00	0.00	0.00	0.00	3.33
13	0.00	0.00	1.00	0.00	0.00	1.11
14	0.00	0.00	1.00	0.00	0.00	2.22
15	0.00	0.00	1.00	0.00	0.00	3.33
16	0.00	0.00	0.00	0.00	0.00	1.11
17	0.00	0.00	0.00	0.00	0.00	2.22
18	0.00	0.00	0.00	0.00	0.00	3.33
	Zeta1	Eta1	Eta2	Theta11	Theta12	

1	1.11	1.11	0.00	1.11	0.00
2	2.22	2.22	0.00	2.22	0.00
3	3.33	3.33	0.00	3.33	0.00
4	1.11	0.00	1.11	0.00	1.11
5	2.22	0.00	2.22	0.00	2.22
6	3.33	0.00	3.33	0.00	3.33
7	1.11	0.00	0.00	0.00	0.00
8	2.22	0.00	0.00	0.00	0.00
9	3.33	0.00	0.00	0.00	0.00
10	0.00	1.11	0.00	0.00	0.00
11	0.00	2.22	0.00	0.00	0.00
12	0.00	3.33	0.00	0.00	0.00
13	0.00	0.00	1.11	0.00	0.00
14	0.00	0.00	2.22	0.00	0.00
15	0.00	0.00	3.33	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00
17	0.00	0.00	0.00	0.00	0.00
18	0.00	0.00	0.00	0.00	0.00
* * * Analysis of Variance * * *					
degrees of freedom for the model					11.0000
degrees of freedom for error					6.0000
total (corrected) degrees of freedom					17.0000
sum of squares for the model					43.9028
sum of squares for error					0.8333
total (corrected) sum of squares					44.7361
model mean square					3.9912
error mean square					0.1389
F-statistic					28.7364
p-value					0.0003
R-squared (in percent)					98.1372
adjusted R-squared (in percent)					94.7221
est. standard deviation of the model error					0.3727
overall mean of y					3.9722
coefficient of variation (in percent)					9.3821

regression

Fits a multivariate linear regression model using least squares.

Synopsis

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

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

The type *double* function is `imsls_d_regression`.

Required Arguments

int **n_rows** (Input)

Number of rows in **x**.

int **n_independent** (Input)

Number of independent (explanatory) variables.

float **x[]** (Input)

Array of size **n_rows** × **n_independent** containing the independent (explanatory) variables(s).

The *i*-th column of **x** contains the *i*-th independent variable.

float **y[]** (Input)

Array of size **n_rows** × **n_dependent** containing the dependent (response) variables(s). The *i*-th column of **y** contains the *i*-th dependent variable. See optional argument **IMSLS_N_DEPENDENT**

to set the value of **n_dependent**.

Return Value

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

Synopsis with Optional Arguments

```
#include <imsls.h>

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

```

    IMSLS_WEIGHTS, float weights[ ],
    IMSLS_REGRESSION_INFO, lmsls_f_regression **regression_info,
    IMSLS_RETURN_USER, float coefficients[ ],
    0)

```

Optional Arguments

IMSLX_COL_DIM, *int x_col_dim* (Input)

Column dimension of **x**.

Default: **x_col_dim** = **n_independent**

IMSLY_COL_DIM, *int y_col_dim* (Input)

Column dimension of **y**.

Default: **y_col_dim** = **n_dependent**

IMSLN_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

IMSLX_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 **IMSLF_FREQUENCIES** and **IMSLW_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.

IMSL_IDO, *int ido* (Input)

Processing option.

The argument **ido** must be one of 0, 1, 2, or 3. If **ido** = 0 (the default), all of the observations are

input during one invocation. If `ido = 1, 2, or 3`, blocks of rows of the data can be processed sequentially in separate invocations of `imsls_f_regression`; with this option, it is not a requirement that all observations be memory resident, thus enabling one to handle large data sets.

<code>ido</code>	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 <code>n_rows</code> observations of <code>x</code> will be performed.
2	This is an intermediate invocation; updating for the <code>n_rows</code> observations of <code>x</code> will be performed.
3	This is the final invocation of this function. Updating for the data in <code>x</code> and wrap-up computations are performed. Workspace is released. No further invocations of <code>imsls_f_regression</code> with <code>ido</code> greater than 1 should be made without first invoking <code>imsls_f_regression</code> with <code>ido = 1</code> .

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

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

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 $m × m$ elements contain the matrix for the first dependent variable, the next $m × m$ 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

$15 \times \mathbf{n_dependent}$ 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:

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`. Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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 $\mathbf{n_dependent} \times \mathbf{n_dependent}$ 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**. To release this space, use **imsls_free**.

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** × *nunits* of memory, where $n = \mathbf{n_independent}$; otherwise, $n = \mathbf{n_independent} + 1$.

Description

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

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

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; $\beta_0, \beta_1, \dots, \beta_k$ 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.,

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

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:

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

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

$$\bar{y}$$

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

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

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 \bar{x}_j represent the mean vector for the independent variables given the data for rows 1, 2, ..., i .

The current mean vector is defined as follows:

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

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

$$\bar{x}_i$$

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:

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

An orthogonal reduction on the centered matrix is computed. When the final computations are performed, the intercept estimate and the first row and column of the estimated covariance matrix of the estimated coefficients are updated (if `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.

$$\sqrt{1 - R_{j \cdot 1, 2, \dots, i-1}^2}$$

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

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

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

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

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

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

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

Output

Least-Squares Coefficients			
0	1	2	3
7.733	-0.200	2.333	-1.667

Example 2

A weighted least-squares fit is computed using the model

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

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

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

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

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

```
#include <imsls.h>

#define N_INDEPENDENT 2
#define N_COEFFICIENTS N_INDEPENDENT + 1
#define N_OBSERVATIONS 4

int 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 `IMSLX_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>
#include <stdio.h>

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

int main()
{
    float coefficients[N_DEPENDENT*N_COEFFICIENTS];
    float scpe[N_DEPENDENT*N_DEPENDENT];
    float anova_table[15*N_DEPENDENT];
    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;
    int indind[N_INDEPENDENT] = {0, 1, 2};
    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, NULL,
        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
cted)                8.00      8.00
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
percent)             97.44     33.73
adj. R-squared
(in percent)         95.90      0.00
est. s.t.d. of
model error          0.89      4.69
overall mean of y    3.00      2.00
coefficient of
variation (in
percent)             29.81     234.52

```

Example 4

Continuing with [Example 1](#) data, the example below invokes the regression function using values of `IDO` greater than 0. Also, usage of the keywords `IMSLS_COEF_COVARIANCES` and `IMSLS_X_MEAN` is demonstrated.

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

#define N_INDEPENDENT 3
#define N_OBSERVATIONS_BLOCK_1 3
#define N_OBSERVATIONS_BLOCK_2 3
#define N_OBSERVATIONS_BLOCK_3 3
#define N_COEFFICIENTS 4

int main()
{
    float coefficients[N_COEFFICIENTS], *coef_covariance=NULL;
    float *anova_table=NULL;
    float *residual=NULL, *x_mean=NULL;

    float x1[][N_INDEPENDENT] = {
        7.0, 5.0, 6.0,
        2.0,-1.0, 6.0,
        7.0, 3.0, 5.0
    };

    float x2[][N_INDEPENDENT] = {
        -3.0, 1.0, 4.0,
        2.0,-1.0, 0.0,
        2.0, 1.0, 7.0
    };

    float x3[][N_INDEPENDENT] = {
        -3.0,-1.0, 3.0,
        2.0, 1.0, 1.0,
        2.0, 1.0, 4.0
    };

    float y1[] = {7.0,-5.0, 6.0};
    float y2[] = {5.0, 5.0,-2.0};
    float y3[] = {0.0, 8.0, 3.0};

    imsls_f_regression(N_OBSERVATIONS_BLOCK_1, N_INDEPENDENT, &x1[0][0], y1,
        IMSLS_RETURN_USER, coefficients,
        IMSLS_IDO, 1,
        0);

    imsls_f_regression(N_OBSERVATIONS_BLOCK_2, N_INDEPENDENT, &x2[0][0], y2,
        IMSLS_RETURN_USER, coefficients,
        IMSLS_IDO, 2,
        0);

    imsls_f_regression(N_OBSERVATIONS_BLOCK_3, N_INDEPENDENT, &x3[0][0], y3,
        IMSLS_RETURN_USER, coefficients,
        IMSLS_COEF_COVARIANCES, &coef_covariance,
        IMSLS_X_MEAN, &x_mean,
        IMSLS_IDO, 3,
        0);
}
```

```

imsls_f_write_matrix("\nLeast Squares Coefficients", 1,
    N_COEFFICIENTS, coefficients, 0);

if (coef_covariance){
    imsls_f_write_matrix("\nCoefficient Covariance",
        N_COEFFICIENTS, N_COEFFICIENTS, coef_covariance,
        IMSLS_PRINT_UPPER,
        0);
    imsls_free(coef_covariance);
}

if (x_mean){
    imsls_f_write_matrix("\nx means", 1, N_INDEPENDENT, x_mean, 0);
    imsls_free(x_mean);
}
}

```

Output

Least Squares Coefficients			
1	2	3	4
7.733	-0.200	2.333	-1.667

Coefficient Covariance				
	1	2	3	4
1	0.3951	-0.0120	0.0289	-0.0778
2		0.0160	-0.0200	-0.0000
3			0.0556	-0.0111
4				0.0222

x means		
1	2	3
	1	4

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 invoking the function with "ido" = 1.

IMSLS_BAD_IDO_7

"ido" = #. A new analysis may not begin until the previous analysis is terminated by invoking the function 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,
```

```
IMSL5_ANOVA_TABLE_USER, float anova_table[ ],
0)
```

Optional Arguments

IMSL5_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

IMSL5_COEF_T_TESTS, *float **coef_t_tests* (Output)

Address of a pointer to the $npar \times 4$ 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 $i = 0, 1, 2, \dots, npar - 1$.

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

IMSL5_COEF_T_TESTS_USER, *float coef_t_tests[]* (Output)

Storage for array **coef_t_tests** is provided by the user. See **IMSL5_COEF_T_TESTS**.

IMSL5_COEF_COL_DIM, *int* **coef_col_dim** (Input)

Column dimension of **coef_t_tests**.

Default: **coef_col_dim** = 4

IMSL5_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 $i = 0, 1, 2, \dots, npar - 1$, 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

$$1.0 - \frac{1.0}{\text{coef_vif}[i]}$$

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

IMSL5_ANOVA_TABLE, *float **anova_table* (Output)

Address of a pointer to the array of size 15 containing the analysis of variance table.

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. Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

IMSL5_ANOVA_TABLE_USER, *float anova_table[]* (Output)

Storage for array `anova_table` is provided by the user. See `IMSL5_ANOVA_TABLE`.

Description

Function `imsls_f_regression_summary` computes summary statistics from a fitted general linear model. The model is $y = X\beta + \epsilon$, 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 $\boldsymbol{\varepsilon}$ 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 $\boldsymbol{\varepsilon}$ 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 $\text{rank}(X) < p$, 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 $j \leq i$) 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 ($\{j : r_{ij} = 0\}$) 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

$$R^{g_3}R^{g_3T}$$

is a symmetric g_2 inverse of $R^TR = X^TX$. See Sallas and Lioni 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>

int 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,
```


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, where SSE is sum of squares error and DFE is degrees of freedom error.

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 $50.0 \leq onecl < 100.0$, set

`confidence = 100.0 - 2.0 * (100.0 - onecl)`.

Default: `confidence = 95.0`

IMSLS_SCHEFFFE_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_SCHEFFFE_CI_USER, *float lower_limit[], float upper_limit[]* (Output)

Storage for arrays `lower_limit` and `upper_limit` is provided by the user. See `IMSLS_SCHEFFFE_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.

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 . The leverage is defined by

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

where $W = \text{diag}(w_1, w_2, \dots, w_n)$ and $(X^T W X)^{-}$ denotes a generalized inverse of $X^T W X$.

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

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

is given by the following:

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

where

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

The computation of the remainder of the case statistics follow easily from their definitions. See [Diagnostics for Individual Cases](#) for the definition of the case diagnostics.

Informational errors can occur if the input matrix \mathbf{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 \mathbf{x} contains a row not in the space spanned by the rows of R . An examination of the model that was fitted and the \mathbf{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 \mathbf{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 \mathbf{x} .

Examples

Example 1

```
#include <imsls.h>

int 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>

int 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

IMSL5_NONESTIMABLE

Within the preset tolerance, the linear combination of regression coefficients is nonestimable.

IMSL5_LEVERAGE_GT_1

A leverage (= #) much greater than 1.0 is computed. It is set to 1.0.

IMSL5_DEL_MSE_LT_0

A deleted residual mean square (= #) much less than 0 is computed. It is set to 0.

Fatal Errors

IMSL5_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_parameters* 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_parameters* 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 $n_{dependent}$, the number of dependent variables in the fitted regression model.

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. The actual size of H is nh by $n_{parameters}$, where nh is the number of rows in the completely testable hypothesis returned by `imsls_f_hypothesis_partial`. Note that nh may be less than or equal to **nhp**.

`IMSL_S_H_MATRIX_USER`, *float h[]* (Output)

Storage for array `h` is provided by the user. See `IMSL_S_H`.

`IMSL_S_G`, *float **g* (Output)

Address of a pointer to the internally allocated array of size `nhp` by `nu` containing the G matrix. The elements of `g` contain the null hypothesis values for the completely testable hypothesis. The actual size of G is nh by `nu`, where nh is the number of rows in the completely testable hypothesis returned by `imsls_f_hypothesis_partial`. Note that nh may be less than or equal to `nhp`.

`IMSL_S_G_USER`, *float g[]* (Output)

Storage for array `g` is provided by the user. See `IMSL_S_G`.

Description

Once a general linear model $y = X\beta + \epsilon$ 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 `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 $c^T = a^T X$, 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 $\mathfrak{R}(H)$) and the row space of X ($\mathfrak{R}(X)$) is not essentially empty and is a proper subset of $\mathfrak{R}(H)$, i.e., $\{0\} \subset \mathfrak{R}(H) \cap \mathfrak{R}(X) \subset \mathfrak{R}(H)$. A hypothesis $H\beta = g$ is completely testable if $\{0\} \subset \mathfrak{R}(H) \cap \mathfrak{R}(X) \subset \mathfrak{R}(X)$. 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 `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 $Y = X\beta + \epsilon$ 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 $\mathfrak{R}(X)$ is replaced by $\mathfrak{R}(R)$ where R is the upper triangular matrix based on the linear equality restrictions. The nonzero rows of R form a basis for the row space 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 columns of Q_1 . Set `rank_hp` = n_1 .

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

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

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

$$W^T A V = \text{diag}(\sigma_1, \dots, \sigma_{\min(n_1, n_2)})$$

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 \mathbf{nhp} by \mathbf{nhp} matrix related to R_1 as follows: If $\mathbf{nhp} < n_parameters$, R_{11} equals the first \mathbf{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 declared inconsistent, an error message with error code equal to 2 is issued.

7. Partition

$$Z^T = (Z_1^T, Z_2^T)$$

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 ($0 < nh < \mathbf{rank_hp}$), or completely testable ($0 < nh = \mathbf{rank_hp}$).

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

Example

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

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

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>
#include <stdio.h>

#define N_ROWS 3
#define N_INDEPENDENT 1
#define N_DEPENDENT 1
#define N_PARAMETERS 3
#define NHP 2

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

    imsls_free(coefficients);
    imsls_free(info);
}

```

```
    imsls_free(x);  
    imsls_free(h);  
    imsls_free(g);  
}
```

Output

```
*** WARNING Error IMSLS_RANK_DEFICIENT from imsls_f_regression. The model *** is not  
full rank. There is not a unique least squares solution. *** The rank of the matrix of  
regressors is 2.
```

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 (lmsls_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 $Y = X\beta + \epsilon$.

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 Salas and Lioni (1988) for multivariate 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 $\varepsilon = 10.0 \times \text{imsls_f_machine}(4)$ ($10.0 \times \text{imsls_d_machine}(4)$ 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

$$\begin{aligned} H\beta U &= G \\ A\beta &= Z \end{aligned}$$

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:

$$\begin{aligned} Y^1 &= X\beta^1 + \varepsilon^1 & (1) \\ A\beta^1 &= Z^1 \\ H\beta^1 &= G \end{aligned}$$

and

$$Y^1 = X\beta^1 + \epsilon^1 \quad (2)$$

$$A\beta^1 = Z^1$$

where $Y^1 = YU$, $\beta^1 = \beta U$, $\epsilon^1 = \epsilon U$, and $Z^1 = ZU$. 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>
#include <stdio.h>

int 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_c (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 $R^T C = H$, $(C^T D C)^-$ denotes the generalized inverse of $C^T D C$, and D is a diagonal matrix with diagonal elements

$$d_{ii} = \begin{cases} 1 & \text{if } r_{ii} > 0 \\ 0 & \text{otherwise} \end{cases}$$

For a detailed discussion, see [Linear Dependence and the R Matrix](#) in the Usage Notes.

The error sum of squares and crossproducts matrix for the model $Y = X\beta + \epsilon$ is

$$(Y - X\hat{\beta})^T (Y - X\hat{\beta})$$

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

$$S_E = U^T (Y - X\hat{\beta})^T (Y - X\hat{\beta}) U$$

Let p equal the order of the matrices S_E and S_H , i.e.,

$$p = \begin{cases} \text{nu} & \text{if nu} > 0 \\ n_dependent & \text{otherwise} \end{cases}$$

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 ($i = 1, 2, \dots, p$) of the generalized eigenvalue problem $S_H x = \lambda S_E x$. These test statistics are as follows:

Wilk's lambda

$$A = \frac{\det(S_E)}{\det(S_H + S_E)} = \prod_{i=1}^p \frac{1}{1 + \lambda_i}$$

The associated p -value is based on an approximation discussed by Rao (1973, p. 556). The statistic

$$F = \frac{ms - pq / 2 + 1}{pq} \frac{1 - A^{1/s}}{A^{1/s}}$$

has an approximate F distribution with pq and $ms - pq / 2 + 1$ numerator and denominator degrees of freedom, respectively, where

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

and

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

The F test is exact if $\min(p, q) \leq 2$ (Kshirsagar, 1972, Theorem 4, p. 299–300).

Roy's maximum root

$$c = \max \lambda_i \quad \text{over all } i$$

where c is output as `value`. The p -value is based on the approximation

$$F = \frac{v + q - s}{s} c$$

where $s = \max(p, q)$ has an approximate F distribution with s and $v + q - s$ numerator and denominator degrees of freedom, respectively. The F test is exact if $s = 1$; the p -value is also exact. In general, the value output in `p_value` is lower bound on the actual p -value.

Hotelling's trace

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

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

$$b = 4 + \frac{pq + 2}{\frac{(v + q - p - 1)(v - 1)}{(v - p - 3)(v - p)}}$$

the p -value is based on the result that

$$F = \frac{b(v - p - 1)}{(b - 2)pq} U$$

has an approximate F distribution with pq and b degrees of freedom. The test is exact if $\min(p, q) = 1$. For $v \leq p + 1$, 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 $\mathbf{v} \geq p$. 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 $\mathbf{v} \geq p$ can be a serious drawback, `imsls_f_hypothesis_test` computes a fourth test statistic based on eigenvalues θ_i ($i = 1, 2, \dots, p$) 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 $\mathbf{v} + q \geq p$. 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

$$V = \text{tr} \left[S_H (S_H + S_E)^{-1} \right] = \sum_{i=1}^p \theta_i$$

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

$$\begin{aligned} s &= \min(p, q) \\ m &= \frac{1}{2}(|p - q| - 1) \\ n &= \frac{1}{2}(\mathbf{v} - p - 1) \end{aligned}$$

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>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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,
IMSL_S_U, nu, u,
IMSL_S_WILK_LAMBDA, &v1, &p1,
IMSL_S_ROY_MAX_ROOT, &v2, &p2,
IMSL_S_HOTELLING_TRACE, &v3, &p3,
IMSL_S_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

IMSL_S_SINGULAR_1

"u"*"scpe"*"u" is singular. Only Pillai's trace can be computed. Other statistics are set to NaN.

Fatal Errors

IMSL_S_NO_STAT_1

"scpe" + "scph" is singular. No tests can be computed.

IMSL_S_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.

IMSL_S_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.

IMSL_S_SINGULAR_2

"u"*"scpe"*"u" + "scph" is singular. No tests can be computed.

IMSL_S_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_criteria, float **criteria,
IMSLS_CRITERIONS_USER, int index_criteria[ ], float criteria[ ],
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,
IMSLS_COEF_STATISTICS_USER, int index_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_criteria, *float ***criteria (Output)

Argument `index_criteria` 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 `criteria` of the first element for each subset size. For `I = 0, 1, ..., nsize - 1`, element numbers `index_criteria[I]`, `index_criteria[I] + 1, ..., index_criteria[I + 1] - 1` of `criteria` correspond to the $(I + 1)$ -st subset size. Argument `criteria` is the address of a pointer to the internally allocated array of length `max(index_criteria[nsize] - 1, n_candidate)` containing in its first `index_criteria[nsize] - 1` elements the criterion values for each subset considered, in increasing subset size order.

IMSLS_CRITERIONS_USER, *int* index_criteria[], *float* criteria[] (Output)

Storage for arrays `index_criteria` and `criteria` is provided by the user. An upper bound on the length of `criteria` is $\max(\max_n_good_saved \times nsize, n_candidate)$. 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 $I = 0, 1, \dots, nsize - 1$, 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 `criteria`.

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:

$$\frac{\max_n_good_saved \times nsize \times (nsize + 1)}{2}$$

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 $\max_subset_size \times \max_n_best$ 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 $\max_n_best \times n_candidate$, otherwise. For $I = 0, 1, \dots, ntbest - 1$, 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 $(\text{index_coefficients}[ntbest] - 1) \times 5$ 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	<i>t</i> -statistic for the test that the coefficient is 0
4	<i>p</i> -value for the two-sided <i>t</i> 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)

$$R^2 = 100 \left(1 - \frac{\text{SSE}_p}{\text{SST}} \right)$$

- R_a^2 (adjusted R^2 in percent)

$$R_a^2 = 100 \left[1 - \left(\frac{n-1}{n-p} \right) \frac{\text{SSE}_p}{\text{SST}} \right]$$

Note that maximizing the criterion is equivalent to minimizing the residual mean square:

$$\frac{\text{SSE}_p}{(n-p)}$$

- Mallows' C_p statistic

$$C_p = \frac{\text{SSE}_p}{s_{\text{n_candidate}}^2} + 2p - n$$

Here, n is equal to the sum of the frequencies (or `n_rows` if `IMLSL5_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

$$s_{\text{n_candidate}}^2$$

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 `IMSL5_INPUT_COV`. Three situations in which the user may want to do this are as follows:

1. The intercept is not in the model. A raw (uncorrected) sum of squares and crossproducts matrix for the independent and dependent variables is required. Argument `n_observations` must be set to 1 greater than the number of observations. Form $A^T A$, where $A = [A, Y]$, to compute the raw sum of squares and crossproducts matrix.

2. An intercept is a candidate variable. A raw (uncorrected) sum of squares and crossproducts matrix for the constant regressor (= 1.0), independent, and dependent variables is required for `cov`. In this case, `cov` contains one additional row and column corresponding to the constant regressor. This row/column contains the sum of squares and crossproducts of the constant regressor with the independent and dependent variables. The remaining elements in `cov` are the same as in the previous case. Argument `n_observations` must be set to 1 greater than the number of observations.
3. There are m variables to be forced into the models. A sum of squares and crossproducts matrix adjusted for the m variables is required (calculated by regressing the candidate variables on the variables to be forced into the model). Argument `n_observations` must be set to m less than the number of observations.

Programming Notes

Function `imsls_f_regression_selection` can save considerable CPU time over explicitly computing all possible regressions. However, the function has some limitations that can cause unexpected results for users who are unaware of the limitations of the software.

1. For $n_candidate + 1 > -\log_2(\epsilon)$, where ϵ is `imsls_f_machine(4)` and `(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, ..., $2^{n_candidate}$) 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 `IMSLD_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 `IMSLD_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

int 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[0][0], 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 Mallows' C_p statistic is used as the criterion rather than R^2 . Note that when Mallows' 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

int 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

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 <code>p_value_in</code> . 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 <code>p_value_out</code> . 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` = $100 * \text{eps}$, 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:

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	<i>F</i> -statistic
9	<i>p</i> -value
10	R^2 (in percent)
11	adjusted R^2 (in percent)
12	estimate of the standard deviation

Note that the *p*-value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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 invocation. 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

Column	Description
2	<i>t</i> -statistic for the test that the coefficient is 0
3	<i>p</i> -value for the two-sided <i>t</i> 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:

$$1.0 - \frac{1.0}{\text{coef_vif}[i]}$$

`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>i</i>-th Variable
-1	Variable <i>i</i> is not in model.
1	Variable <i>i</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>i</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 <i>k</i> -th step.
$k < 0.0$	Variable was deleted from model during the <i>k</i> -th step.

IMSLS_COV_SWEPT_USER, *float* *covs (Output)

User-allocated array of length $(n_candidate + 1) \times (n_candidate + 1)$ 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 vari-

able. 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 Efroymsen (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

int 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[0][0], 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);
}

```

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, lmsls_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 , $i = 0, \dots, \text{degree} - 1$, and the columns are described as follows:

Column	Description
0	degrees of freedom
1	Sums of squares

Column	Description
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 , $i = 0, \dots, \text{degree} - 1$, 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>i</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

Column	Description
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)

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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,

$$R^2 = \frac{\sum w_i (\hat{y}_i - \bar{y})^2}{\sum w_i (y_i - \bar{y})^2} 100\%$$

where

$$\hat{y}_i$$

is the fitted y value at x_i and \bar{y} is the mean of y . This statistic is useful in assessing the overall fit of the curve to the data. R^2 must be between 0 and 100 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
```

```

int main()
{
    float    *coefficients;
    float    x[] = {0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 4.0,
                   4.0, 5.0, 5.0, 6.0, 6.0, 7.0, 7.0};
    float    y[] = {508.1, 498.4, 568.2, 577.3, 651.7, 657.0, 755.3,
                   758.9, 787.6, 792.1, 841.4, 831.8, 854.7, 871.4};

    coefficients = 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

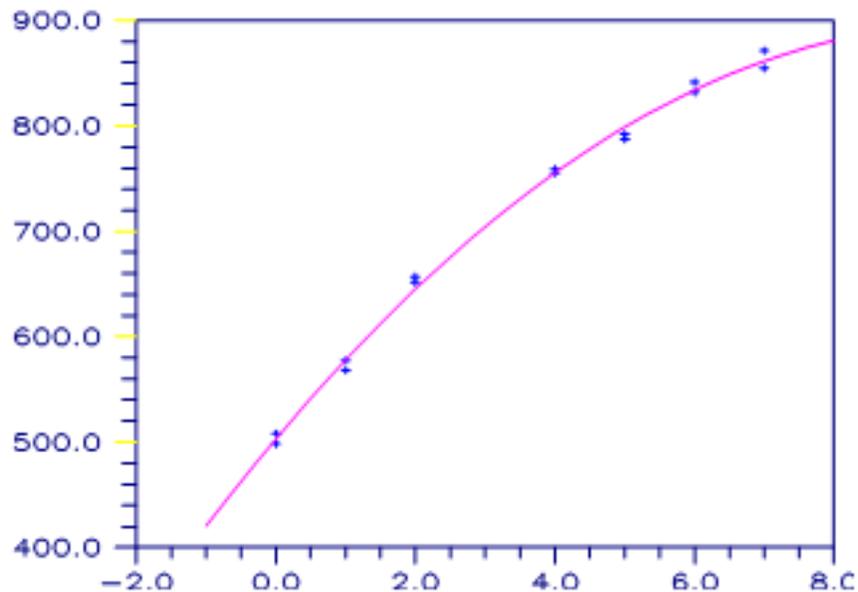


Figure 1, A Polynomial Fit

Example 2

This example is a continuation of the initial example. Here, many optional arguments are used.

```

#include <imsls.h>

#define DEGREE      2
#define NOBS        14

int 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_SCHEFFÉ_CI, float **lower_limit, float **upper_limit,
IMSLS_SCHEFFÉ_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[],
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_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 inter-

vals with confidence level `onecl`, where $50.0 \leq \text{onecl} < 100.0$, set `confidence = 100.0 - 2.0 * (100.0 - \text{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 $\text{SSE}/(\text{DFE} * \text{weights}[i])$ for the estimated variance of a future response, where SSE is sum of squares error and DFE is degrees of freedom error.

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.

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.

IMSL5_COOKSD_USER, *float* `cooks`d[] (Output)

Storage for array `cooks`d is provided by the user. See `IMSL5_COOKSD`.

IMSL5_DFFITS, *float ***`dffits` (Output)

Address of a pointer to an internally allocated array of length `n_predict` containing the DFFITS statistics, where DFFITS is the change in the predicted value of a point in the absence of the empirical value.

IMSL5_DFFITS_USER, *float* `dffits`[] (Output)

Storage for array `dffits` is provided by the user. See `IMSL5_DFFITS`.

Description

Function `imsls_f_poly_prediction` assumes a polynomial model

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

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 \mathbf{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_regression`, which produces the fit using orthogonal polynomials, are used for input by the structure `poly_info`. The fitted model from `imsls_f_poly_regression` is

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

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 $X^T X$ matrix for this model is a diagonal matrix with elements d_j . The case statistics are easily computed from this model and are equal to those from the original polynomial model with β_j 's as the regression coefficients.

The leverage is computed as follows:

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

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>

int 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);

imsls_free(coefficients);
imsls_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>

int 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], cooks_d[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, cooks,
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, cooks, 0);
imsls_f_write_matrix("DFFITS", 1, N_PREDICT, dffits, 0);

imsls_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



[more...](#)

Fits a multivariate 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 `imsls_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_VARIANCE_COVARIANCE_MATRIX, float **var_covar,
    IMSLS_VARIANCE_COVARIANCE_MATRIX_USER, float var_covar[ ],
    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 $j = 1, 2, \dots, n_parameters$.

IMSLS_THETA_SCALE, *float theta_scale[]* (Input)

Array with **n_parameters** components containing the scaling array for **theta**. 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 details.

Default: `theta_scale[] = 1`.

IMSLS_GRADIENT_EPS, *float gradient_eps* (Input)

Scaled gradient tolerance. The j -th component of the scaled gradient at **theta** is calculated as

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

where $g = \nabla F(\theta)$, $t = \text{theta_scale}$, and

$$\|F(\theta)\|_2^2 = \sum_{i=1}^n (y_i - f(x_i; \theta))^2$$

The value $F(\theta)$ is the sum of the squared residuals, SSE, at the point θ .

Convergence is declared if $|g_i| * \max\{|\theta_i|, 1.0/t_i\}/\text{SSE}$ is less than `gradient_eps` for $i = 0, 1, 2, \dots, n_parameters$, where g_i is the i -th component of an internal intermediate gradient vector.

Default:

$$\text{gradient_eps} = \sqrt{\epsilon}$$

($\sqrt[3]{\epsilon}$ in double, where ϵ is the machine precision)

`IMSLS_STEP_EPS`, *float step_eps* (Input)

Scaled step tolerance. The j -th component of the scaled step from points θ and θ' is computed as

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

where $t = \text{theta_scale}$

Convergence is declared if $|g_{n+i}| / \max\{|\theta_i|, 1.0/t_i\}$ is less than `step_eps` for $i = 0, 1, 2, \dots, n$, where g_{n+i} is the i -th component of the last step and $n = n_parameters$.

Default: `step_eps` = $\epsilon^{2/3}$, where ϵ is the machine precision

`IMSLS_SSE_REL_EPS`, *float sse_rel_eps* (Input)

Relative SSE function tolerance.

Convergence is declared if the change in SSE is less than or equal to `sse_rel_eps` * SSE in absolute value.

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

`IMSLS_SSE_ABS_EPS`, *float sse_abs_eps* (Input)

Absolute SSE function tolerance.

Convergence is declared if SSE is less than `sse_abs_eps`.

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

IMSL5_MAX_STEP, *float* `max_step` (Input)

Maximum allowable step size.

Default: `max_step` = 1000 $\max(\epsilon_1, \epsilon_2)$, where $\epsilon_1 = (t^T \theta_0)^{1/2}$, $\epsilon_2 = \|t\|_2$, $t = \text{theta_scale}$, and $\theta_0 = \text{theta_guess}$

IMSL5_INITIAL_TRUST_REGION, *float* `trust_region` (Input)

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

IMSL5_GOOD_DIGIT, *int* `ndigit` (Input)

Number of good digits in the function.

Default: machine dependent

IMSL5_MAX_ITERATIONS, *int* `max_itn` (Input)

Maximum number of iterations.

Default: `max_itn` = 100

IMSL5_MAX_SSE_EVALUATIONS, *int* `max_sse_eval` (Input)

Maximum number of SSE function evaluations.

Default: `max_sse_eval` = 400

IMSL5_MAX_JACOBIAN_EVALUATIONS, *int* `max_jacobian` (Input)

Maximum number of Jacobian evaluations.

Default: `max_jacobian` = 400

IMSL5_TOLERANCE, *float* `tolerance` (Input)

False convergence tolerance.

Default: `tolerance` = 100* `eps`, where `eps` = `imsl5_f_machine(4)` if single precision and `eps` = `imsl5_d_machine(4)` if double precision

IMSL5_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.

IMSL5_PREDICTED_USER, *float* `predicted[]` (Output)

Storage for array `predicted` is provided by the user. See `IMSL5_PREDICTED`.

IMSL5_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.

IMSL5_RESIDUAL_USER, *float* `residual[]` (Output)

Storage for array `residual` is provided by the user. See `IMSL5_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_VARIANCE_COVARIANCE_MATRIX`, *float **var_covar* (Output)

Address of a pointer to an internally allocated array of size `n_parameters × n_parameters` containing the variance/covariance matrix of the coefficients.

`IMSLS_VARIANCE_COVARIANCE_MATRIX_USER`, *float var_covar []* (Output)

Storage for array `var_covar` is provided by the user. See

`IMSLS_VARIANCE_COVARIANCE_MATRIX`.

`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 [Passing Data to User-Supplied Functions](#) at the beginning of this manual for more details.

IMSL_S_JACOBIAN_W_DATA, void jacobian (int n_independent, float xi[], int n_parameters, float theta[], float f_jac[]), 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 [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

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

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:

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

A value of θ that minimizes

$$\sum_{i=1}^n [e_i(\theta)]^2$$

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:

$$\left(J(\hat{\theta}_c)^T J(\hat{\theta}_c) + \mu_c I \right) s_c = J(\hat{\theta}_c)^T e(\hat{\theta}_c)$$

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.

The first stopping criterion for `imsls_f_nonlinear_regression` occurs when SSE is less than the absolute function tolerance. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance. The third stopping criterion occurs when the scaled distance between the last two steps is less than the step tolerance. The third stopping criterion also generates error `IMSLS_LITTLE_FCN_CHANGE`. The fourth stopping criterion occurs when the relative change in SSE is less than `sse_rel_eps`. The fourth stopping criterion also generates error code `IMSLS_FALSE_CONVERGENCE`. See Dennis and Schnabel (1983, pages 159–161, 278–280, and 347–348) for a discussion of stopping criteria and choice of tolerances.

On some platforms, `imsls_f_nonlinear_regression` can evaluate the user-supplied functions `fcn` and `jacobian` in parallel. This is done only if the function `imsls_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

Programming Notes

Nonlinear regression allows substantial flexibility over linear regression because the user can specify the functional form of the model. This added flexibility can cause unexpected convergence problems for users that are unaware of the limitations of the software. Also, in many cases, there are possible remedies that may not be immediately obvious. The following is a list of possible convergence problems and some remedies. There is not a one-to-one correspondence between the problems and the remedies. Remedies for some problems also may be relevant for the other problems.

1. A local minimum is found. Try a different starting value. Good starting values often can be obtained by fitting simpler models. For example, for a nonlinear function.

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

good starting values can be obtained from the estimated linear regression coefficients

$$\hat{\beta}_0$$

and

$$\hat{\beta}_1$$

from a simple linear regression of $\ln y$ on $\ln x$. The starting values for the nonlinear regression in this case would be

$$\theta_1 = e^{\hat{\beta}_0} \text{ and } \theta_2 = \hat{\beta}_1$$

If an approximate linear model is not clear, then simplify the model by reducing the number of nonlinear regression parameters. For example, some nonlinear parameters for which good starting values are known could be set to these values in order to simplify the model for computing starting values for the remaining parameters.

2. The estimate of θ is incorrectly returned as the same or very close to the initial estimate. This occurs often because of poor scaling of the problem, which might result in the residual sum of squares being either very large or very small relative to the precision of the computer. The optional arguments allow control of the scaling.
3. The model is discontinuous as a function of θ . (The function $f(\mathbf{x}; \theta)$ 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:

$$Y = \alpha + (0.49 - \alpha)e^{-\beta(X-8)} + \varepsilon$$

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

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

int 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};

  imsls_omp_options(IMSL_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

                          /* 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);
}

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)));
}
```

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

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

int main()
{
    int df;
    int n_independent = 1;
    int n_parameters = 2;
    int n_obs = 15;
    float *theta_hat, *r, *y_hat, *var_covar;
    float grad_eps = 1.0e-9;
    float theta_guess[2] = {60.0, -0.03};
    float a, dfe, normalValue;

    float y[15] = {
        54.0, 50.0, 45.0, 37.0, 35.0,
        25.0, 20.0, 16.0, 18.0, 13.0,
        8.0, 11.0, 8.0, 4.0, 6.0
    };

    float x[15] = {
        2.0, 5.0, 7.0, 10.0, 14.0,
        19.0, 26.0, 31.0, 34.0, 38.0,
        45.0, 52.0, 53.0, 60.0, 65.0
    };

    char *fmt="%15.3f";
    char *dashes=
        "-----";

    /* Nonlinear regression */
    theta_hat = imsls_f_nonlinear_regression(fcn, n_parameters, n_obs,
        n_independent, x, y,
        IMSLS_THETA_GUESS, theta_guess,
        IMSLS_GRADIENT_EPS, grad_eps,
        IMSLS_DF, &df,
        IMSLS_R, &r,
        IMSLS_PREDICTED, &y_hat,
        IMSLS_VARIANCE_COVARIANCE_MATRIX, &var_covar,
        IMSLS_JACOBIAN, jacobian,
        0);

    /* Print results */
    /* Calculate and Print Coefficients & their 95% Confidence Limits */
    printf("\n          ESTIMATED COEFFICIENTS \n");
}
```

```

printf("%s\n", dashes);
printf(" Coefficient | Lower 95% Limit |");
printf(" Estimate | Upper 95% Limit \n");

dfe = (float) df;
normalValue = imsls_f_t_inverse_cdf(0.975, dfe);
a = normalValue * sqrt(var_covar[0]);

printf("    Theta_1 | %10.3f      | %7.3f | %12.3f\n",
       theta_hat[0] - a, theta_hat[0], theta_hat[0] + a);

a = normalValue * sqrt(var_covar[3]);

printf("    Theta_2 | %10.3f      | %7.3f | %12.3f\n",
       theta_hat[1] - a, theta_hat[1], theta_hat[1] + a);

printf("%s\n", dashes);

imsls_f_write_matrix("Var/Covar matrix", n_parameters, n_parameters,
                    var_covar,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);

imsls_f_write_matrix("Predicted values", 1, n_obs, y_hat,
                    IMSLS_WRITE_FORMAT, "%7.2f",
                    0);
}

static float fcn(int n_independent, float x[], int n_parameters,
                float theta[])
{
    return (theta[0] * exp(x[0] * theta[1]));
} /* End of fcn */

static 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
-----
Coefficient | Lower 95% Limit | Estimate | Upper 95% Limit
Theta_1    | 55.426          | 58.607  | 61.787
Theta_2    | -0.043          | -0.040  | -0.036
-----

Var/Covar matrix
      1      2
1     2.167   -0.002
2     -0.002   0.000

Predicted values
      1      2      3      4      5      6      7      8
54.15  48.08  44.42  39.45  33.67  27.62  20.94  17.18

```

	9	10	11	12	13	14	15
	15.26	13.02	9.87	7.48	7.19	5.45	4.40

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.

IMSLS_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

nonlinear_optimization



[more...](#)

Fits data to a nonlinear model (possibly with linear constraints) using the successive quadratic programming algorithm (applied to the sum of squared errors, $sse = \sum(y_i - f(x_i; \theta))^2$) 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 `imsls_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,
```

```

IMSL_S_PREDICTED_USER, float predicted[ ],
IMSL_S_RESIDUAL, float **residual,
IMSL_S_RESIDUAL_USER, float residual[ ],
IMSL_S_SSE, float *sse,
IMSL_S_RETURN_USER, float theta_hat[ ],
IMSL_S_FCN_W_DATA, float fcn( ), void *data,
IMSL_S_JACOBIAN_W_DATA, float jacobian( ), void *data,
0)

```

Optional Arguments

IMSL_S_THETA_GUESS, *float theta_guess[]* (Input)

Array with **n_parameters** components containing an initial guess.

Default: **theta_guess[] = 0**

IMSL_S_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 $j = 1, 2, \dots, n_parameters$. 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.

IMSL_S_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 .

IMSL_S_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:

$$a_{i1} \theta_1 + \dots + a_{ij} \theta_j = b_i \text{ for } i = 1, n_equality \text{ and } j = 1, n_parameter, \text{ and}$$

$$a_{k1} \theta_1 + \dots + a_{kj} \theta_j \leq b_k \text{ for } k = n_equality + 1, n_constraints \text{ and } j = 1, n_parameter.$$

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:

<code>stop_info</code>	Reason for leaving routine
1	θ is feasible, and the condition that depends on <code>acc</code> is satisfied.
2	θ is feasible, and rounding errors are preventing further progress.
3	θ is feasible, but <code>sse</code> fails to decrease although a decrease is predicted by the current gradient vector.
4	The calculation cannot begin because <code>a</code> contains fewer than <code>n_constraints</code> 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 <code>theta_lb[i]</code> equal to <code>theta_ub[i]</code> .
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 <code>sse</code> evaluations (<code>max_sse_eval</code>) 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 [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 [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 $\min f(\boldsymbol{\theta})$, $\boldsymbol{\theta} \in \mathfrak{X}$, subject to

$$\begin{aligned} A_1\boldsymbol{\theta} &= b_1 \\ A_2\boldsymbol{\theta} &\leq b_2 \\ \boldsymbol{\theta}_l &\leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_u \end{aligned}$$

given the vectors b_1 , b_2 , $\boldsymbol{\theta}_l$, and $\boldsymbol{\theta}_u$ and the matrices A_1 and A_2 .

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise $\boldsymbol{\theta}^0$, the initial guess provided by the user, to satisfy

$$A_1\boldsymbol{\theta} = b_1$$

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

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

subject to

$$\begin{aligned} a_j d &= 0 & j \in I_k \\ a_j d &\leq 0 & j \in J_k \end{aligned}$$

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 $\theta_i \leq (\theta_{U})_i$ and $a_j = -e_i$ for the constraint $\theta_i \leq (\theta_l)_i$. 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

$$f(\theta^k + \alpha^k d^k) \leq f(\theta^k) + 0.1 \alpha^k (d^k)^T \nabla f(\theta^k)$$

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

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

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

float fcn(int n_independent, float x[], int n_parameters, float theta[]);

int 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[11] = {0.05, 0.21, 0.67, 0.72, 0.98, 0.94,
                  1.00, 0.73, 0.44, 0.36, 0.02};

    imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    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);

    imsls_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.

$$y_i = \theta_0 + (0.49 - \theta_0)e^{-\theta_1(x_i - 8)} + \varepsilon_i$$

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[]);

int 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;

imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

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);

imsls_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.

IMSL5_BAD_CONSTRAINTS_4

The variables are determined by the equality constraints.

IMSL5_TOO_MANY_ITERATIONS_1

Number of function evaluations exceeded "maxfcn" = #.

IMSL5_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

Lnorm_regression

Fits a multiple linear regression model using either the Least Absolute Value (L_1), Least L_p norm (L_p), or Least Maximum Value (Minimax or L_∞) method of multiple linear regression.

Synopsis

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

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

The type *double* function is `imsls_d_Lnorm_regression`.

Required Arguments

int n_rows (Input)

Number of rows in **x**.

int n_independent (Input)

Number of independent (explanatory) variables.

float x[] (Input)

Array of size **n_rows** × **n_independent** containing the independent (explanatory) variables(s).

The *i*-th column of **x** contains the *i*-th independent variable.

float y[] (Input)

Array of size **n_rows** containing the dependent (response) variable.

Return Value

`imsls_f_Lnorm_regression` returns a pointer to an array of length **n_independent** + 1 containing a least absolute value solution for the regression coefficients. The estimated intercept is the initial component of the array, where the *i*-th component contains the regression coefficients for the *i*-th dependent variable. If the optional argument `IMSL_NO_INTERCEPT` is used then the (*i*-1)-st 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, or
```

```
    IMSLS_METHOD_LLQ, float p, or
```

```
    IMSLS_METHOD_LMV,
```

```
    IMSLS_X_COL_DIM, int x_col_dim,
```

```
    IMSLS_INTERCEPT, or
```

```
    IMSLS_NO_INTERCEPT,
```

```
    IMSLS_TOLERANCE, float tolerance,
```

```
    IMSLS_WEIGHTS, float weights[],
```

```
    IMSLS_FREQUENCIES, float frequencies[],
```

```
    IMSLS_MAX_ITERATIONS
```

```
    IMSLS_RANK, int *rank,
```

```
    IMSLS_ITERATIONS, int *iterations,
```

```
    IMSLS_N_ROWS_MISSING, int *n_rows_missing,
```

```
    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_SCALE, float *square_of_scale,
```

```
    IMSLS_RESIDUALS_LP_NORM, float *Lp_norm_residual,
```

```
    IMSLS_EPS, float epsilon,
```

```
    IMSLS_RETURN_USER, float coefficients[],
```

```
    0)
```

Optional Arguments

IMSLS_METHOD_LAV (Input)

or

IMSLS_METHOD_LL p , float p (Input)

or

IMSLS_METHOD_LMV, (Input)

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_LL p 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_X_COL_DIM, int x_col_dim (Input)

Leading dimension of \mathbf{x} exactly as specified in the dimension statement in the calling program.

IMSLS_INTERCEPT (Input)

or

IMSLS_NO_INTERCEPT, (Input)

IMSLS_INTERCEPT is the default where the fitted value for observation i is

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

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_TOLERANCE, float $tolerance$ (Input)

Tolerance used in determining linear dependence.

$Tolerance = 100 \times imsls_f_machine(4)$ is the default.

For more details see Chapter 15, [Utilities](#) function [imsls_f_machine](#).

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_MAX_ITERATIONS**, *int *iterations* (Input)
Maximum number of iterations allowed when using the multiple linear regression method.
IMSLS_MAX_ITERATIONS is only applicable if **IMSLS_METHOD_LL_P** is specified.
Default = 100
- 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_SEA**, *float sum_lav_error* (Output)
Sum of the absolute value of the errors. **IMSLS_SEA** is only applicable if **IMSLS_METHOD_LAV** is also specified.
- IMSLS_MAX_RESIDUAL**, *float max_residual* (Output)
Magnitude of the largest residual. **IMSLS_MAX_RESIDUAL** is only applicable if **IMSLS_METHOD_LMV** is specified.
- IMSLS_R**, *float **R_matrix* (Output)
Upper triangular matrix of dimension (number of coefficients by number of coefficients) containing the R matrix from a QR decomposition of the matrix of regressors. **IMSLS_R** is only applicable if **IMSLS_METHOD_LL_P** is specified.
- 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_DEGREES_OF_FREEDOM** is only applicable if **IMSLS_METHOD_LL_P** is specified.
- IMSLS_RESIDUALS**, *float **residual* (Output)
Address of a pointer to an array of length **n_rows** (equal to the number of observations) containing the residuals. **IMSLS_RESIDUALS** is only applicable if **IMSLS_METHOD_LL_P** is specified.
- IMSLS_RESIDUALS_USER**, *float residual[]* (Output)
Storage for array **residual** is provided by the user. See **IMSLS_RESIDUALS**.

IMSL_SSCALE, *float *square_of_scale* (Output)

Square of the scale constant used in an L_p analysis. An estimated asymptotic variance-covariance matrix of the regression coefficients is `square_of_scale` $\times (R^T R)^{-1}$. **IMSL_SSCALE** is only applicable if **IMSL_METHOD_LL**P is specified.

IMSL_RESIDUALS_LP_NORM, *float *Lp_norm_residual* (Output)

L_p norm of the residuals. **IMSL_RESIDUALS_LP_NORM** is only applicable if **IMSL_METHOD_LL**P is specified.

IMSL_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.

Default: `Epsilon = 100 \times machine(4)`. **IMSL_EPS** is only applicable if **IMSL_METHOD_LL**P is specified.

IMSL_RETURN_USER, *float coefficients[]* (Output)

Storage for array `coefficients` is provided by the user. See Return Value.

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 **IMSL_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

$$\sum_{i=0}^{n-1} |y_i - \hat{y}_i|$$

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 **IMSL_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 `IMLSL_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 `IMLSL_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 `IMLSL_LLP` computes estimates of the regression coefficients in a multiple linear regression model $y = X\beta + \epsilon$ under the criterion of minimizing the L_p norm of the deviations for $i = 0, \dots, n-1$ 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 `IMSL_WEIGHTS` and `IMSL_FREQUENCIES` are not supplied, the estimated regression coefficient vector,

$$\hat{\beta}$$

(output in `coefficients[]`) minimizes the L_p norm

$$\left(\sum_{i=0}^{n-1} |y_i - \hat{y}_i|^p \right)^{1/p}$$

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

$$\left(\sum_{i=0}^{n-1} f_i \left| \sqrt{w_i} (y_i - \hat{y}_i) \right|^p \right)^{1/p}$$

The asymptotic variance-covariance matrix of the estimated regression coefficients is given by

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

where R is from the QR decomposition of the matrix of regressors (output in `R-Matrix`). An estimate of λ^2 is output in `square_of_scale`.

In the discussion that follows, we will first present the algorithm with frequencies and weights all taken to be one. Later, we will present the modifications to handle frequencies and weights different from one.

Option call `IMSLs_LLp` uses Newton's method with a line search for $p > 1.25$ and, for $p \leq 1.25$, uses a modification due to Ekblom (1973, 1987) in which a series of perturbed problems are solved in order to guarantee convergence and increase the convergence rate. The cutoff value of 1.25 as well as some of the other implementation details given in the remaining discussion were investigated by Sallas (1990) for their effect on CPU times.

In each case, for the first iteration a least-squares solution for the regression coefficients is computed using routine `imsls_f_regression`. 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

$$z_i^{(k+1)} = |e_i^{(k)}|^{p-1} \text{sign}(e_i^{(k)})$$

and

$$v_i^{(k+1)} = |e_i^{(k)}|^{p-2}$$

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:

$$v_i^{(k+1)} = \begin{cases} \tau^{p-2} & \text{if } p < 2 \text{ and } |e_i^{(k)}| < \tau \\ |e_i^{(k)}|^{p-2} & \text{otherwise} \end{cases}$$

where τ equals $100 \times \text{imsls_f_machine}(4)$ (or $100.0 \times \text{imsls_d_machine}(4)$ 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

$$\left[V^{(k+1)} \right]^{1/2} X$$

is computed using fast Givens transformations. Let

$$R^{(k+1)}$$

denote the upper triangular matrix from the QR decomposition. The linear system

$$\left[R^{(k+1)} \right]^T R^{(k+1)} d^{(k+1)} = X^T z^{(k+1)}$$

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 $\alpha^{(k+1)}$. 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 ρ . The value of ρ is constrained to be no less than 0.1.

An approximate upper bound for ρ is 0.5. If after 10 successive backtrack attempts, $\alpha^{(k)} = \rho_1 \rho_2 \dots \rho_{10}$ 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:

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

where s is the square root of the residual mean square from the least-squares fit on the first iteration.

For the case $1 \leq p \leq 1.25$, 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

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

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 $[R^{(k+1)}]^T R^{(k+1)} d^{(k+1)} = \chi^T z^{(k+1)}$ is solved for $d^{(k+1)}$ where $R^{(k+1)}$ is from the QR decomposition of $[V^{(k+1)}]^{1/2} \chi$. 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 $\max(\text{epsilon}, 10^{-j})$ where $j = 1, 2, 3, \dots$ 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

$$t_i = \frac{(p-2)v_i^{(k)}}{\left(e_i^{(k)}\right)^2 + c^2}$$

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

$$\left[R^{(k)} \right]^T R^{(k)} d = X^T t$$

Convergence of the sequence of problems is declared when the maximum relative difference in residuals from the solution of successive problems is less than `epsilon`.

The preceding discussion was limited to the case for which `weights[i] = 1` and `frequencies[i] = 1`, i.e., the weights and frequencies are all taken equal to one. The necessary modifications to the preceding algorithm to handle weights and frequencies not all equal to one are as follows:

1. Replace

$$e_i^{(k)} \text{ by } \sqrt{w_i} e_i^{(k)}$$

in the definitions of

$$z_i^{(k+1)}, v_i^{(k+1)}, \delta_i^{(k+1)}$$

and t_i .

2. Replace

$$z_i^{(k+1)} \text{ by } f_i(w_i)^{1/2} z_i^{(k+1)}, v_i^{(k+1)} \text{ by } f_i w_i v_i^{(k+1)}, \text{ and } t_i^{(k+1)} \text{ by } f_i(w_i)^{1/2} t_i^{(k+1)}$$

These replacements have the same effect as multiplying the i -th row of X and y by

$$\sqrt{w_i}$$

and repeating the row f_i times except for the fact that the residuals returned by `imsls_f_Lnorm_regression` are in terms of the original y and X .

Finally, R and an estimate of λ^2 are computed. Actually, R is recomputed because on output it corresponds to the R from the initial QR decomposition for least squares. The formula for the estimate of λ^2 depends on p .

For $p = 1$, the estimator for λ^2 is given by (McKean and Schrader 1987)

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

with

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

where $z_{0.975}$ is the 97.5 percentile of the standard normal distribution, and where

$$\tilde{e}_{(m)} (m = 1, 2, \dots, DFE)$$

are the ordered residuals where rank zero residuals are excluded. Note that

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

For $p = 2$, the estimator of λ^2 is the customary least-squares estimator given by

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

For $1 < p < 2$ and for $p > 2$, the estimator for λ^2 is given by (Gonin and Money 1989)

$$\hat{\omega}_p^2 = \frac{m_{2p-2}}{[(p-1)m_{p-2}]^2}$$

with

$$m_r = \frac{\sum_{i=0}^{n-1} f_i |\sqrt{w_i} (y_i - \hat{y}_i)|^r}{\sum_{i=0}^{n-1} f_i}$$

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

$$\max_{0 \leq i \leq n-1} |y_i - \hat{y}_i|$$

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.

Examples

Example 1

A straight line fit to a data set is computed under the LAV criterion.

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

int 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

```

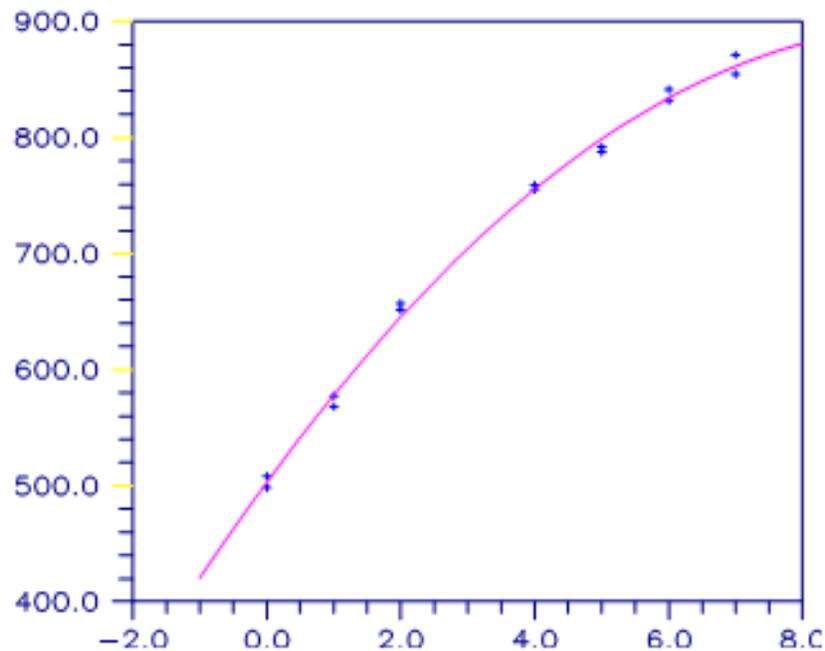


Figure 2, Least Squares and Least Absolute Value Fitted Lines

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>

int 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;
    float Lp_norm_residual;
    float R_matrix[4], residuals[8];
    float *coefficients = NULL;
    int i, irank, iter, nrmiss;
    int n_row=2;
    int n_col=2;
    char *dashes=
        "-----";

    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_LLQ, 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\n",
            residuals[0], residuals[1], residuals[2], residuals[3]);
        printf("\t%6.2f\t%6.2f\t%6.2f\t%6.2f\n\n",
            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("%s\n\n", dashes);
        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 = 1.000
Lp norm of the residuals = 6.002
Rank of Regressors Matrix = 2
Degrees of Freedom Error = 6.000
Number of Iterations = 8
Number of Missing Values = 0
Square of Scale Constant = 6.248
    
```

R Matrix

```

          1          2
1      2.828      8.485
2      0.000      3.464
-----
    
```

```

Coefficients =    0.39    0.56
Residuals =    0.06    2.39   -1.50    0.50
              -0.55    0.45   -0.61   -0.16
    
```

```

P = 1.500
Lp norm of the residuals = 3.712
Rank of Regressors Matrix = 2
Degrees of Freedom Error = 6.000
Number of Iterations = 6
Number of Missing Values = 0
Square of Scale Constant = 1.059
    
```

R Matrix

```

          1          2
1      2.828      8.485
2      0.000      3.464
-----
    
```

```

Coefficients =   -0.13    0.75
Residuals =    0.38    2.13   -1.38    0.63
              -0.63    0.38   -0.88   -0.63
    
```

```

P = 2.000
Lp norm of the residuals = 2.937
Rank of Regressors Matrix = 2
Degrees of Freedom Error = 6.000
Number of Iterations = 1
Number of Missing Values = 0
Square of Scale Constant = 1.438
    
```

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	= 2.500			
Lp norm of the residuals	= 2.540			
Rank of Regressors Matrix	= 2			
Degrees of Freedom Error	= 6.000			
Number of Iterations	= 4			
Number of Missing Values	= 0			
Square of Scale Constant	= 0.789			
R Matrix				
	1	2		
1	2.828	8.485		
2	0.000	3.464		

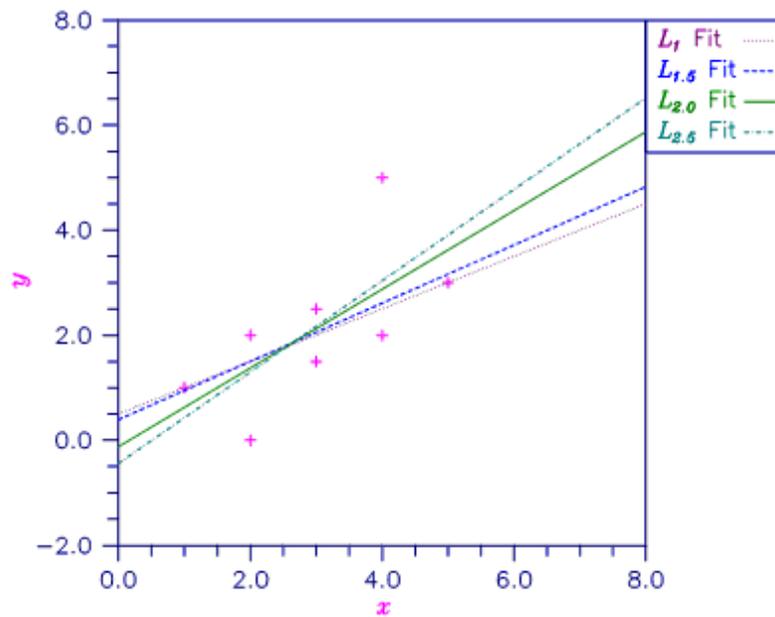


Figure 3, Various LP Fitted Lines

Example 3

A straight line fit to a data set is computed under the LMV criterion.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    float xx[] = {0.0, 1.0, 2.0, 3.0, 4.0, 4.0, 5.0};
    float yy[] = {0.0, 2.5, 2.5, 4.5, 4.5, 6.0, 5.0};
    float max_residual;
    int irank, iter, nrmiss;

    float *coefficients = NULL;

    coefficients = imsls_f_Lnorm_regression(7, 1, xx, yy,
        IMSLS_METHOD_LMV,
        IMSLS_MAX_RESIDUAL, &max_residual,
        IMSLS_RANK, &irank,
        IMSLS_ITERATIONS, &iter,
        IMSLS_N_ROWS_MISSING, &nrmiss,
        0);

    printf("B = %6.2f\t%6.2f\n\n", coefficients[0], coefficients[1]);
    printf("Rank of Regressors Matrix    = %3d\n", irank);
    printf("Magnitude of Largest Residual = %8.4f\n", max_residual);
    printf("Number of Iterations           = %3d\n", iter);
    printf("Number of Rows Missing            = %3d\n", nrmiss);

}
```

Output

```
B =   1.00    1.00
Rank of Regressors Matrix    = 2
Magnitude of Largest Residual = 1.00000
Number of Iterations         = 3
Number of Rows Missing       = 0
```

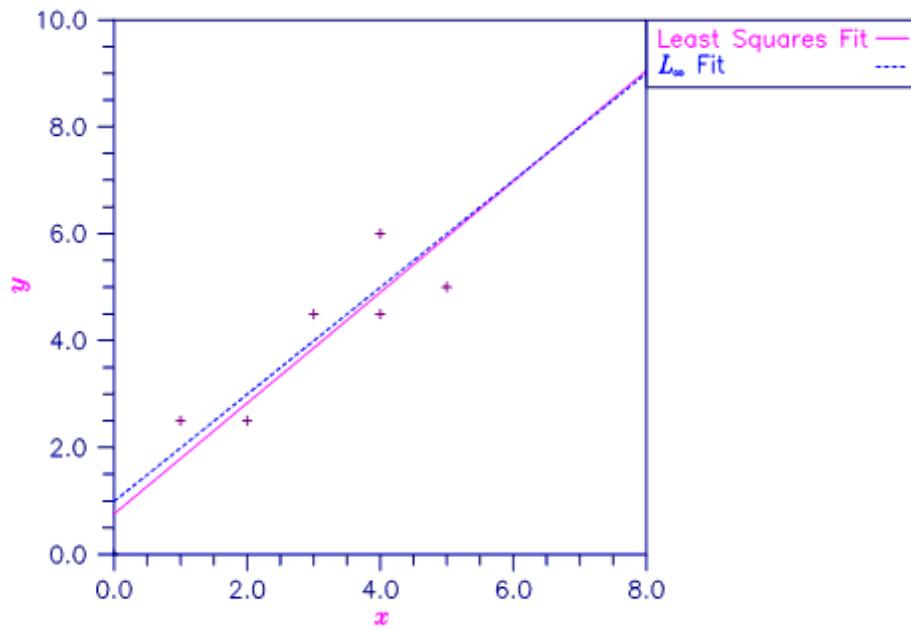


Figure 4, Least Squares and Least Maximum Value

pls_regression

Performs partial least squares (PLS) regression for one or more response variables and one or more predictor variables.

Synopsis

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

```
float *imsls_f_pls_regression (int ny, int h, float y[ ], int nx, int p, float x[ ], ..., 0)
```

The type *double* function is `imsls_d_pls_regression`.

Required Arguments

int ny (Input)

The number of rows of **y**.

int h (Input)

The number of response variables.

float y[] (Input)

Array of length $ny \times h$ containing the values of the responses.

int nx (Input)

The number of rows of **x**.

int p (Input)

The number of predictor variables.

float x[] (Input)

Array of length $nx \times p$ containing the values of the predictor variables.

Return Value

A pointer to the array of length $ix \times iy$ containing the final PLS regression coefficient estimates for the mean-centered variables, where $ix \leq p$ is the number of predictor variables in the model, and $iy \leq h$ is the number of response variables. To release this space, use `imsls_free`. If the estimates cannot be computed, NULL is returned.

Synopsis with Optional Arguments

```
#include <imspls.h>
```

```
float *imspls_f_pls_regression (int ny, int h, float y[ ], int nx, int p, float x[ ],  
    IMSLS_N_OBSERVATIONS, int nobs,  
    IMSLS_Y_INDICES, int iy, int iyind[ ],  
    IMSLS_X_INDICES, int ix, int ixind[ ],  
    IMSLS_N_COMPONENTS, int ncomps,  
    IMSLS_CROSS_VALIDATION, int cv,  
    IMSLS_N_FOLD, int k,  
    IMSLS_SCALE, int scale,  
    IMSLS_PRINT_LEVEL, int iprint,  
    IMSLS_OPT_N_COMPONENTS, int *optcomps,  
    IMSLS_PREDICTED, float **yhat,  
    IMSLS_PREDICTED_USER, float yhat[ ],  
    IMSLS_RESIDUALS, float **resids,  
    IMSLS_RESIDUALS_USER, float resids[ ],  
    IMSLS_STD_ERRORS, float **se,  
    IMSLS_STD_ERRORS_USER, float se[ ],  
    IMSLS_PRESS, float **press,  
    IMSLS_PRESS_USER, float press[ ],  
    IMSLS_X_SCORES, float **xscrs,  
    IMSLS_X_SCORES_USER, float xscrs[ ],  
    IMSLS_Y_SCORES, float **yscrs,  
    IMSLS_Y_SCORES_USER, float yscrs[ ],  
    IMSLS_X_LOADINGS, float **xldgs,  
    IMSLS_X_LOADINGS_USER, float xldgs[ ],
```

```
IMSLS_Y_LOADINGS, float **yldgs,  
IMSLS_Y_LOADINGS_USER, float yldgs[ ],  
IMSLS_WEIGHTS, float **wts,  
IMSLS_WEIGHTS_USER, float wts[ ],  
IMSLS_STANDARD_COEF, float **standard_coef,  
IMSLS_STANDARD_COEF_USER, float standard_coef[ ],  
IMSLS_INTERCEPT_TERMS, float **intercepts,  
IMSLS_INTERCEPT_TERMS_USER, float intercepts[ ],  
IMSLS_PCT_VAR, float **pctvar,  
IMSLS_PCT_VAR_USER, float pctvar[ ],  
IMSLS_RETURN_USER, float coef[ ],  
0)
```

Optional Arguments

IMSLS_N_OBSERVATIONS, *int nob*s (Input)

Positive integer specifying the number of observations to be used in the analysis.

Default: `nobs = min(ny, nx)`.

IMSLS_Y_INDICES, *int iy*, *int iyind*[] (Input)

Argument `iyind` is an array of length `iy` containing column indices of `y` specifying which response variables to use in the analysis. Each element in `iyind` must be less than or equal to `h-1`.

Default: `iy = h`, `iyind = 0, 1, ..., h-1`.

IMSLS_X_INDICES, *int ix*, *int ixind*[] (Input)

Argument `ixind` is an array of length `ix` containing column indices of `x` specifying which predictor variables to use in the analysis. Each element in `ixind` must be less than or equal to `p-1`.

Default: `ix = p`, `ixind = 0, 1, ..., p-1`.

IMSLS_N_COMPONENTS, *int ncomps* (Input)

The number of PLS components to fit. `ncomps ≤ ix`.

Default: `ncomps = ix`.

If `cv = 1` is used, models with 1 up to `ncomps` components are tested using cross-validation. The model with the lowest predicted residual sum of squares is reported.

`IMSLS_CROSS_VALIDATION`, *int cv* (Input)

If `cv = 0`, the function fits only the model specified by `ncomps`. If `cv = 1`, the function performs *K*-fold cross validation to select the number of components.

Default: `cv = 1`.

`IMSLS_N_FOLD`, *int k* (Input)

The number of folds to use in *K*-fold cross validation. `k` must be between 2 and `nobs`, inclusive. `k` is ignored if `cv = 0` is used.

Default: `k = 5`.

`IMSLS_SCALE`, *int scale* (Input)

If `scale = 1`, `y` and `x` are centered and scaled to have mean 0 and standard deviation of 1. If `scale = 0`, `y` and `x` are centered to have mean 0 but are not scaled.

Default: `scale = 0`.

`IMSLS_PRINT_LEVEL`, *int iprint* (Input)

Printing option.

<code>iprint</code>	Action
0	No Printing.
1	Prints final results only.
2	Prints intermediate and final results.

Default: `iprint = 0`.

`IMSLS_OPT_N_COMPONENTS`, *int *optcomps* (Output)

The number of components of the optimal model. The value is identical with `ncomps`, if `cv = 0` is used.

`IMSLS_PREDICTED`, *float **yhat* (Output)

Argument `yhat` is the address of an array of length `nobs × iy`, containing the predicted values for the response variables using the final values of the coefficients.

`IMSLS_PREDICTED_USER`, *float yhat []* (Output)

Storage for array `yhat` is provided by the user. See `IMSLS_PREDICTED`.

IMSLS_RESIDUALS, *float **resids* (Output)

Argument **resids** is the address of an array of length **nobs** × **iy**, containing residuals of the final fit for each response variable.

IMSLS_RESIDUALS_USER, *float resids[]* (Output)

Storage for array **resids** is provided by the user. See **IMSLS_RESIDUALS**.

IMSLS_STD_ERRORS, *float **se* (Output)

Argument **se** is the address of an array of length **ix** × **iy**, containing the standard errors of the PLS coefficients.

IMSLS_STD_ERRORS_USER, *float se[]* (Output)

Storage for array **se** is provided by the user. See **IMSLS_STD_ERRORS**.

IMSLS_PRESS, *float **press* (Output)

Argument **press** is the address of an array of length **ncomps** × **iy**, containing the predicted residual error sum of squares obtained by cross-validation for each model of size $j= 1, \dots, \mathbf{ncomps}$ components. The argument **press** is ignored if **cv** = 0 is used for **IMSLS_CROSS_VALIDATION**.

IMSLS_PRESS_USER, *float press[]* (Output)

Storage for array **press** is provided by the user. See **IMSLS_PRESS**.

IMSLS_X_SCORES, *float **xscrs* (Output)

Argument **xscrs** is the address of an array of length **nobs** × **ncomps** containing *X*-scores.

IMSLS_X_SCORES_USER, *float xscrs[]* (Output)

Storage for array **xscrs** is provided by the user. See **IMSLS_X_SCORES**.

IMSLS_Y_SCORES, *float **yscrls* (Output)

Argument **yscrls** is the address of an array of length **nobs** × **ncomps** containing *Y*-scores.

IMSLS_Y_SCORES_USER, *float yscrls[]* (Output)

Storage for array **yscrls** is provided by the user. See **IMSLS_Y_SCORES**.

IMSLS_X_LOADINGS, *float **xldgs* (Output)

Argument **xldgs** is the address of an array of length **ix** × **ncomps**, containing *X*-loadings.

IMSLS_X_LOADINGS_USER, *float xldgs[]* (Output)

Storage for array **xldgs** is provided by the user. See **IMSLS_X_LOADINGS**.

IMSLS_Y_LOADINGS, *float **yldgs* (Output)

Argument **yldgs** is the address of an array of length **iy** × **ncomps**, containing *Y*-loadings.

IMSLS_Y_LOADINGS_USER, *float yldgs[]* (Output)

Storage for array **yldgs** is provided by the user. See **IMSLS_Y_LOADINGS**.

IMSLS_WEIGHTS, *float **wts* (Output)

Argument **wts** is the address of an array of length $ix \times ncomps$, containing the weight vectors.

IMSLS_WEIGHTS_USER, *float wts []* (Output)

Storage for array **wts** is provided by the user. See **IMSLS_WEIGHTS**.

IMSLS_STANDARD_COEF, *float **standard_coef* (Output)

Argument **standard_coef** is the address of an array of length $ix \times iy$, containing the final PLS regression coefficient estimates for the centered (if **scale** = 0) or standardized variables (if **scale** = 1). The contents of **standard_coef** and **coef** are identical if **scale** = 0 is used.

IMSLS_STANDARD_COEF_USER, *float standard_coef []* (Output)

Storage for array **standard_coef** is provided by the user. See **IMSLS_STANDARD_COEF**.

IMSLS_INTERCEPT_TERMS, *float **intercepts* (Output)

Argument **intercepts** is the address of an array of length iy , containing the intercept terms of the PLS regression.

IMSLS_INTERCEPT_TERMS_USER, *float intercepts []* (Output)

Storage for array **intercepts** is provided by the user. See **IMSLS_INTERCEPT_TERMS**.

IMSLS_PCT_VAR, *float **pctvar* (Output)

Argument **pctvar** is the address of an array of length $2 \times ncomps$, containing the percentage of variance explained by the model in its first **optcomps** columns. The first row contains the percentage of variance of **x** explained by each component, the second row the percentage of variance of **y** explained by each component.

IMSLS_PCT_VAR_USER, *float pctvar []* (Output)

Storage for array **pctvar** is provided by the user. See **IMSLS_PCT_VAR**.

IMSLS_RETURN_USER, *float coef []* (Output)

If specified, the final PLS regression coefficient estimates are stored in array **coef** provided by the user.

Description

Function **imsls_f_pls_regression** performs partial least squares regression for a response matrix $Y(n_y \times h)$ and a set of p explanatory variables, $X(n_x \times p)$. **imsls_f_pls_regression** finds linear combinations of the predictor variables that have highest covariance with Y . In so doing, **imsls_f_pls_regression** produces a predictive model for Y using components (linear combinations) of the individual predictors. Other names for these linear combinations are scores, factors, or latent variables. Partial least squares regression is an alternative method to ordinary least squares for problems with many, highly collinear predictor variables. For further discussion see, for example, Abdi (2010), and Frank and Friedman (1993).

In Partial Least Squares (PLS), a score, or component matrix, T , is selected to represent both X and Y as in,

$$X = TP^T + E_x$$

and

$$Y = TQ^T + E_y$$

The matrices P and Q are the least squares solutions of X and Y regressed on T .

That is,

$$P^T = (T^T T)^{-1} T^T X$$

and

$$Q^T = (T^T T)^{-1} T^T Y$$

The columns of T in the above relations are often called X -scores, while the columns of P are the X -loadings. The columns of the matrix U in $Y = UQ^T + G$ are the corresponding Y scores, where G is a residual matrix and Q , as defined above, contains the Y -loadings.

Restricting T to be linear in X , the problem is to find a set of weight vectors (columns of W) such that $T = XW$ predicts both X and Y reasonably well.

Formally, $W = [w_1, \dots, w_{m-1}, w_m, \dots, w_M]$ where each w_j is a column vector of length p , $M \leq p$ is the number of components, and where the m -th partial least squares (PLS) component w_m solves:

$$\begin{cases} \max_{\alpha} \text{Corr}^2(Y, X\alpha) \text{Var}(X\alpha) \\ \text{s.t.} \\ \|\alpha\| = 1 \\ \alpha^T S w_l = 0, \quad l = 1, \dots, m-1 \end{cases}$$

where $S = X^T X$ and $\|\alpha\| = \sqrt{\alpha^T \alpha}$ is the Euclidean norm. For further details see Hastie, et. al., pages 80-82 (2001).

That is, w_m is the vector which maximizes the product of the squared correlation between Y and $X\alpha$ and the variance of $X\alpha$, subject to being orthogonal to each previous weight vector left multiplied by S . The PLS regression

coefficients $\hat{\beta}_{PLS}$ arise from

$$Y = X\hat{\beta}_{PLS} + E_y = TQ^T + E_y = XWQ^T + E_y, \text{ or } \hat{\beta}_{PLS} = WQ^T$$

Algorithms to solve the above optimization problem include NIPALS (nonlinear iterative partial least squares) developed by Herman Wold (1966, 1985) and numerous variations, including the SIMPLS algorithm of de Jong (1993). `imspls_f_pls_regression` implements the SIMPLS method. SIMPLS is appealing because it finds a solution in terms of the original predictor variables, whereas NIPALS reduces the matrices at each step. For univariate Y it has been shown that SIMPLS and NIPALS are equivalent (the score, loading, and weights matrices will be proportional between the two methods).

By default, `imspls_f_pls_regression` searches for the best number of PLS components using K -fold cross-validation. That is, for each $M = 1, 2, \dots, p$, `imspls_f_pls_regression` estimates a PLS model with M components using all of the data except a hold-out set of size roughly equal to `nobs/k`. Using the resulting model estimates, `imspls_f_pls_regression` predicts the outcomes in the hold-out set and calculates the predicted residual sum of squares (PRESS). The procedure then selects the next hold-out sample and repeats for a total of K times (i.e., *folds*). For further details see Hastie, et. al., pages 241-245 (2001).

For each response variable, `imspls_f_pls_regression` returns results for the model with lowest PRESS. The best model (the number of components giving lowest PRESS), generally will be different for different response variables.

When requested via the optional argument `IMSLS_STD_ERRORS`, `imspls_f_pls_regression` calculates modified jackknife estimates of the standard errors as described in Martens and Martens (2000).

Comments

1. `imspls_f_pls_regression` defaults to leave-one-out cross-validation when there are too few observations to form K folds in the data. The user is cautioned that there may be too few observations to make strong inferences from the results.
2. This implementation of `imspls_f_pls_regression` does not handle missing values. The user should remove missing values or NaN's from the input data.

Examples

Example 1

The following artificial data set is provided in de Jong (1993).

$$X = \begin{bmatrix} -4 & 2 & 1 \\ -4 & -2 & -1 \\ 4 & 2 & -1 \\ 4 & -2 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} 430 & -94 \\ -436 & 12 \\ -361 & -22 \\ 367 & 104 \end{bmatrix}$$

The first call to `imsls_f_pls_regression` fixes the number of components to 3 for both response variables, and the second call performs K -fold cross validation. Note that because the number of folds is equal to n , `imsls_f_pls_regression` performs leave-one-out (LOO) cross-validation.

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

#define H 2
#define N 4
#define P 3

int main() {
    int iprint=1, ncomps=3;
    float x[N][P] = {
        -4.0, 2.0, 1.0,
        -4.0, -2.0, -1.0,
        4.0, 2.0, -1.0,
        4.0, -2.0, 1.0
    };
    float y[N][H] = {
        430.0, -94.0,
        -436.0, 12.0,
        -361.0, -22.0,
        367.0, 104.0
    };
    float *coef=NULL, *yhat=NULL, *se=NULL;
    float *coef2=NULL, *yhat2=NULL, *se2=NULL;

    /* Print out informational error. */
    imsls_error_options(IMSLS_SET_PRINT, IMSLS_ALERT, 1, 0);

    printf("Example 1a: no cross-validation, request %d components.\n",
           ncomps);
    coef = imsls_f_pls_regression(N, H, &y[0][0], N, P, &x[0][0],
                                 IMSLS_N_COMPONENTS, ncomps,
                                 IMSLS_CROSS_VALIDATION, 0,
                                 IMSLS_PRINT_LEVEL, iprint,
                                 IMSLS_PREDICTED, &yhat,
                                 IMSLS_STD_ERRORS, &se,
                                 0);

    printf("\nExample 1b: cross-validation\n");
    coef2 = imsls_f_pls_regression(N, H, &y[0][0], N, P, &x[0][0],
                                   IMSLS_N_FOLD, N,
```

```

    IMSLS_PRINT_LEVEL, iprint,
    IMSLS_PREDICTED, &yhat2,
    IMSLS_STD_ERRORS, &se2,
    0);
}

```

Output

Example 1a: no cross-validation, request 3 components.

PLS Coeff		
	1	2
1	0.8	10.3
2	17.3	-29.0
3	398.5	5.0

Predicted Y		
	1	2
1	430	-94
2	-436	12
3	-361	-22
4	367	104

Std. Errors		
	1	2
1	131.5	5.1
2	263.0	10.3
3	526.0	20.5

*** WARNING Error IMSLS_PLS_REGRESSION_CONVERGED from imsls_f_pls_regression.

*** The PLS regression algorithm converged in 2 iterations, but the
 *** number of requested PLS components is 3. The number of computed
 *** PLS components is reduced to 2.

Example 1b: cross-validation

Cross-validated results for response 1:

Comp	PRESS
1	3860649
2	5902575
3	5902575

The best model has 1 component(s).

Cross-validated results for response 2:

Comp	PRESS
1	36121
2	8984
3	8984

The best model has 2 component(s).

PLS Coeff		
	1	2
1	6.0	-0.2
2	66.1	-2.2

3	361.4	-11.8
Predicted Y		
	1	2
1	469.5	-15.4
2	-517.6	17.0
3	-205.3	6.7
4	253.4	-8.3
Std. Errors		
	1	2
1	131.2	18.5
2	114.8	10.1
3	561.5	22.5
*** WARNING Error IMSLS_PLS_REGRESSION_CONVERGED from imsls_f_pls_regression.		
***	The PLS regression algorithm converged in 2 iterations, but the	
***	number of requested PLS components is 3. The number of computed	
***	PLS components is reduced to 2.	

Example 2

The data, as appears in S. Wold et al. (2001), is a single response variable, the “free energy of the unfolding of a protein”, while the predictor variables are 7 different, highly correlated measurements taken on 19 amino acids.

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

#define H 1
#define N 19
#define P 7

int main() {
    int iprint=2, ncomps=7;
    float x[N][P] = {
        0.23, 0.31, -0.55, 254.2, 2.126, -0.02, 82.2,
        -0.48, -0.6, 0.51, 303.6, 2.994, -1.24, 112.3,
        -0.61, -0.77, 1.2, 287.9, 2.994, -1.08, 103.7,
        0.45, 1.54, -1.4, 282.9, 2.933, -0.11, 99.1,
        -0.11, -0.22, 0.29, 335.0, 3.458, -1.19, 127.5,
        -0.51, -0.64, 0.76, 311.6, 3.243, -1.43, 120.5,
        0.0, 0.0, 0.0, 224.9, 1.662, 0.03, 65.0,
        0.15, 0.13, -0.25, 337.2, 3.856, -1.06, 140.6,
        1.2, 1.8, -2.1, 322.6, 3.35, 0.04, 131.7,
        1.28, 1.7, -2.0, 324.0, 3.518, 0.12, 131.5,
        -0.77, -0.99, 0.78, 336.6, 2.933, -2.26, 144.3,
        0.9, 1.23, -1.6, 336.3, 3.86, -0.33, 132.3,
        1.56, 1.79, -2.6, 366.1, 4.638, -0.05, 155.8,
        0.38, 0.49, -1.5, 288.5, 2.876, -0.31, 106.7,
        0.0, -0.04, 0.09, 266.7, 2.279, -0.4, 88.5,
        0.17, 0.26, -0.58, 283.9, 2.743, -0.53, 105.3,
        1.85, 2.25, -2.7, 401.8, 5.755, -0.31, 185.9,
        0.89, 0.96, -1.7, 377.8, 4.791, -0.84, 162.7,
        0.71, 1.22, -1.6, 295.1, 3.054, -0.13, 115.6
    };
    float y[N][H] = {8.5, 8.2, 8.5, 11.0, 6.3, 8.8, 7.1, 10.1,
        16.8, 15.0, 7.9, 13.3, 11.2, 8.2, 7.4, 8.8, 9.9, 8.8, 12.0};
```

```

float *coef=NULL, *yhat=NULL, *se=NULL;
float *coef2=NULL, *yhat2=NULL, *se2=NULL;

printf("Example 2a: no cross-validation, request %d components.\n",
      ncomps);
coef = imsls_f_pls_regression(N, H, &y[0][0], N, P, &x[0][0],
    IMSLS_N_COMPONENTS, ncomps,
    IMSLS_CROSS_VALIDATION, 0,
    IMSLS_SCALE, 1,
    IMSLS_PRINT_LEVEL, iprint,
    IMSLS_PREDICTED, &yhat,
    IMSLS_STD_ERRORS, &se,
    0);

printf("\nExample 2b: cross-validation\n");
coef2 = imsls_f_pls_regression(N, H, &y[0][0], N, P, &x[0][0],
    IMSLS_SCALE, 1,
    IMSLS_PRINT_LEVEL, iprint,
    IMSLS_PREDICTED, &yhat2,
    IMSLS_STD_ERRORS, &se2,
    0);
}

```

Output

Example 2a: no cross-validation, request 7 components.

Standard PLS Coefficients

```

1
-5.468
1.668
0.624
1.424
-2.550
4.870
4.871

```

PLS Coeff

```

1
-20.07
4.63
1.42
0.09
-7.27
20.93
0.46

```

Predicted Y

```

1
9.37
7.30
8.10
12.02
8.79
6.76
7.24
10.45
15.79
14.36

```

8.41
 9.94
 11.52
 8.64
 8.22
 8.40
 11.13
 8.97
 12.39

Variance Analysis

=====

Pctge of Y variance explained

Component	Cum. Pctge
1	42.3
2	45.5
3	61.2
4	68.5
5	71.6
6	78.7
7	78.8

=====

Pctge of X variance explained

Component	Cum. Pctge
1	64.2
2	97.7
3	99.0
4	99.5
5	99.8
6	99.9
7	100.0

Std. Errors

1
 13.13
 6.72
 1.84
 0.20
 4.68
 14.30
 0.33

Example 2b: cross-validation

Cross-validated results for response 1:

Comp	PRESS
1	167.5
2	162.9
3	166.5
4	168.8
5	264.6
6	221.1
7	184.7

The best model has 2 component(s).

Standard PLS Coefficients

1
 0.1598

0.2163
 -0.1673
 0.0095
 -0.0136
 0.1649
 0.0294

PLS Coeff

1
 0.5867
 0.6000
 -0.3797
 0.0006
 -0.0388
 0.7089
 0.0028

Predicted Y

1
 9.86
 7.71
 7.35
 11.02
 8.32
 7.46
 9.32
 9.00
 12.09
 12.09
 6.59
 11.11
 12.46
 10.27
 9.02
 9.51
 12.82
 10.69
 11.09

Variance Analysis

=====

Pctge of Y variance explained

Component	Cum. Pctge
1	42.3
2	45.5

=====

Pctge of X variance explained

Component	Cum. Pctge
1	64.2
2	97.7

Std. Errors

1
 0.2615
 0.2029
 0.1302
 0.0041
 0.2078
 0.4279
 0.0064

Warning Errors

IMSL5_PLS_REGRESSION_CONVERGED

The PLS regression algorithm converged in # iterations, but the number of requested PLS components is #. The number of computed PLS components is reduced to #.

Correlation and Covariance

Functions

Variations, Covariances, and Correlations

Variance-covariance or correlation matrix	covariances	266
Partial correlations and covariances	partial_covariances	275
Pooled covariance matrix	pooled_covariances	281
Robust estimate of covariance matrix	robust_covariances	288

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 function `imsls_f_covariances`.
- For data grouped by some auxiliary variable, function `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_covariances`.
- Function `imsls_f_robust_covariances` computes robust M-estimates of the mean and covariance matrix from a matrix of observations.

covariances



[more...](#)

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 <code>missing_value_method = 1</code> . 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 <code>missing_value_method = 2</code> . 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` \times `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 missing 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:

$$x_{k0} = 0.0 \text{ for } k = 1, \dots, p$$

$$c_{jk0} = 0.0 \text{ for } j, k = 1, \dots, p$$

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} :

$$r_{i+1} = \frac{f_{i+1}w_{i+1}}{\sum_{l=1} f_l w_l}$$

$$\bar{x}_{k, i+1} = \bar{x}_{ki} + (x_{k, i+1} - \bar{x}_{ki})r_{i+1}$$

$$c_{jk, i+1} = c_{jki} + f_{i+1}w_{i+1} (x_{j, i+1} - \bar{x}_{ji}) (x_{k, i+1} - \bar{x}_{ki}) (1 - r_{i+1})$$

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

$$\bar{x}_k = \frac{\sum_{i=1}^n f_i w_i x_{ki}}{\sum_{i=1}^n f_i w_i}$$

where n is the number of observations.

The following formula defines the sample covariance, s_{jk} , between variables j and k :

$$s_{jk} = \frac{\sum_{i=1}^n f_i w_i (x_{ji} - \bar{x}_j) (x_{ki} - \bar{x}_k)}{\sum_{i=1}^n f_i - 1}$$

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

int 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

int main()
{
    char *title;
    float *means, *correlations;
    float x[] = {
        1.0, 5.1, 3.5, 1.4, .2, 1.0, 4.9, 3.0, 1.4, .2,
        1.0, 4.7, 3.2, 1.3, .2, 1.0, 4.6, 3.1, 1.5, .2,
        1.0, 5.0, 3.6, 1.4, .2, 1.0, 5.4, 3.9, 1.7, .4,
        1.0, 4.6, 3.4, 1.4, .3, 1.0, 5.0, 3.4, 1.5, .2,
        1.0, 4.4, 2.9, 1.4, .2, 1.0, 4.9, 3.1, 1.5, .1,
        1.0, 5.4, 3.7, 1.5, .2, 1.0, 4.8, 3.4, 1.6, .2,
        1.0, 4.8, 3.0, 1.4, .1, 1.0, 4.3, 3.0, 1.1, .1,
        1.0, 5.8, 4.0, 1.2, .2, 1.0, 5.7, 4.4, 1.5, .4,

```

```

1.0, 5.4, 3.9, 1.3, .4, 1.0, 5.1, 3.5, 1.4, .3,
1.0, 5.7, 3.8, 1.7, .3, 1.0, 5.1, 3.8, 1.5, .3,
1.0, 5.4, 3.4, 1.7, .2, 1.0, 5.1, 3.7, 1.5, .4,
1.0, 4.6, 3.6, 1.0, .2, 1.0, 5.1, 3.3, 1.7, .5,
1.0, 4.8, 3.4, 1.9, .2, 1.0, 5.0, 3.0, 1.6, .2,
1.0, 5.0, 3.4, 1.6, .4, 1.0, 5.2, 3.5, 1.5, .2,
1.0, 5.2, 3.4, 1.4, .2, 1.0, 4.7, 3.2, 1.6, .2,
1.0, 4.8, 3.1, 1.6, .2, 1.0, 5.4, 3.4, 1.5, .4,
1.0, 5.2, 4.1, 1.5, .1, 1.0, 5.5, 4.2, 1.4, .2,
1.0, 4.9, 3.1, 1.5, .2, 1.0, 5.0, 3.2, 1.2, .2,
1.0, 5.5, 3.5, 1.3, .2, 1.0, 4.9, 3.6, 1.4, .1,
1.0, 4.4, 3.0, 1.3, .2, 1.0, 5.1, 3.4, 1.5, .2,
1.0, 5.0, 3.5, 1.3, .3, 1.0, 4.5, 2.3, 1.3, .3,
1.0, 4.4, 3.2, 1.3, .2, 1.0, 5.0, 3.5, 1.6, .6,
1.0, 5.1, 3.8, 1.9, .4, 1.0, 4.8, 3.0, 1.4, .3,
1.0, 5.1, 3.8, 1.6, .2, 1.0, 4.6, 3.2, 1.4, .2,
1.0, 5.3, 3.7, 1.5, .2, 1.0, 5.0, 3.3, 1.4, .2};

```

```

/* 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



[more...](#)

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 $n = \mathbf{n_independent} + \mathbf{n_dependent}$. 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 `IMSL_S_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:

	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 (Input)

or

`IMSLS_PARTIAL_CORR` (Input)

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 the 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 follows 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

$$P_{22|1} = \left[\text{diag}(\Sigma_{22|1}) \right]^{-1/2} \Sigma_{22|1} \left[\text{diag}(\Sigma_{22|1}) \right]^{-1/2}$$

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 \mathbf{x} , `df`, is not known, the resulting *p*-values may be useful for comparison, but they should not be 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>

int 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][0], 0);

imsls_f_write_matrix("Partial Covariances", 6, 6, pcov, 0);

imsls_free(pcov);
}

```

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>
#include <stdio.h>

int 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);

imsls_free(pcorr);
imsls_free(pval);
}

```

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 $n_rows \times n_variables + 1$ 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, or
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, float **means,
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 <code>n_rows</code> observations of <code>x</code> will be performed.
2	This is an intermediate invocation; updating for the <code>n_rows</code> observations of <code>x</code> will be performed.
3	All statistics are updated for the <code>n_rows</code> observations. The covariance matrix computed.

Default: `ido = 0`

IMSLS_ROWS_ADD (Input)

or

IMSLS_ROWS_DELETE (Input)

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 errors.

If **n_rows** ≠ 0 (for any value of **ido**), the group observation totals, T_i , for $i = 1, \dots, g$, 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

$$\bar{x}_i = \frac{T_i}{\sum_j w_{ij} f_{ij}}$$

$$S = \frac{1}{\sum_{ij} f_{ij} - g} \sum_{i,j} w_{ij} f_{ij} (x_{ij} - \bar{x}_i)(x_{ij} - \bar{x}_i)^T$$

Examples

Example 1

The following example computes a pooled variance-covariance matrix. The last column of the data set is the group indicator.

```
#include <imsls.h>

int 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);
    imsls_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 permutation of the matrix or the use of the `IMSLS_X_INDICES` optional keyword. This example chooses the keyword method.

```
#include <imsls.h>

int 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);

    imsls_free(means);
    imsls_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` (Input)

or

`IMSLS_INITIAL_EST_MEDIAN` (Input)

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 Used
0	Huber's conjugate-gradient algorithm is used.
1	Stahel's algorithm is used.

`IMSLS_PERCENTAGE`, *float percentage* (Input)

Percentage of gross errors expected in the data. Argument `percentage` must be in the range 0.0 to 100.0 and contains the percentage of outliers expected in the data. If the percentage of gross errors expected in the data is not known, a reasonable strategy is to choose a value of `percentage` that is such that larger values do not result in significant changes in the estimates.

Default: `percentage = 5.0`

`IMSLS_MAX_ITERATIONS`, *int maxit* (Input)

Maximum number of iterations.

Default: `maxit = 30`

`IMSLS_TOLERANCE`, *float tolerance* (Input)

Convergence criterion. When the maximum absolute change in a location or covariance estimate is less than `tolerance`, convergence is assumed.

Default: `tolerance` = 10^{-4}

`IMSLS_MINIMAX_WEIGHTS`, *float *a, float *b, float *c* (Output)

Arguments `a`, `b`, and `c` contain the values for the parameters of the weighting function. See the [Description](#) section.

`IMSLS_GROUP_COUNTS`, *int **gcounts* (Output)

Address of a pointer to an integer array of length `n_groups` containing the number of observations in each group.

`IMSLS_GROUP_COUNTS_USER`, *int gcounts[]* (Output)

Storage for integer array `gcounts` is provided by the user. See `IMSLS_GROUP_COUNTS`.

`IMSLS_SUM_WEIGHTS`, *float **sum_weights* (Output)

Address of a pointer to an array of length `n_groups` containing the sum of the weights times the frequencies in the groups.

`IMSLS_SUM_WEIGHTS_USER`, *float sum_weights[]* (Output)

Storage for array `sum_weights` is provided by the user. See `IMSLS_SUM_WEIGHTS`.

`IMSLS_MEANS`, *float **means* (Output)

Address of a pointer to an array of size `n_groups` by `n_variables`. The i -th row of `means` contains the group i variable means.

`IMSLS_MEANS_USER`, *float means[]* (Output)

Storage for array `means` is provided by the user. See `IMSLS_MEANS`.

`IMSLS_U`, *float **u* (Output)

Address of a pointer to an array of size `n_variables` by `n_variables` containing the lower matrix U , the lower triangular for the robust sample cross-products matrix. U is computed from the robust sample covariance matrix, S (see the [Description](#) section), as $S = U^T U$.

`IMSLS_U_USER`, *float u[]* (Output)

Storage for array `u` is provided by the user. See `IMSLS_U`.

`IMSLS_BETA`, *float *beta* (Output)

Argument `beta` contains the constant used to ensure that the estimated covariance matrix has unbiased expectation (for a given mean vector) for a multivariate normal density.

`IMSLS_N_ROWS_MISSING`, *int *nrmiss* (Output)

Number of rows of data encountered in calls to `robust_covariances` containing missing values (NaN) for any of the variables used.

IMSLs_RETURN_USER, float c[] (Output)

If specified, c returns the covariance matrix. Storage for array c is provided by the user.

Description

Function `imsls_f_robust_covariances` computes robust M-estimates of the mean and covariance matrix from a matrix of observations. A pooled estimate of the covariance matrix is computed when multiple groups are present in the input data. M-estimate weights are obtained using the “minimax” weights of Huber (1981, pp. 231-235), with `percentage` expected gross errors. Huber’s (1981) weighting equations are given by:

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

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

User specified observation weights and frequencies may be given for each row in \mathbf{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 \mathbf{x} in population (group) i with mean vector μ_i , for $i = 1, \dots, \tau$. Let the covariance matrix Σ be such that $\Sigma = R^T R$. If

$$\mathbf{y} = R \bullet^T (\mathbf{x} - \mu_i)$$

then

$$g(\mathbf{y}) = |\Sigma|^{1/2} f(R^T \mathbf{y} + \mu_i; \mu_i, \Sigma)$$

It is assumed that $g(\mathbf{y})$ 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

$$\frac{1}{n} \sum_{j=1}^{n_i} f_{ig} w_{ij} w(r_{ij}) y_{ij} = 0$$

and

$$\frac{1}{n} \sum_{i=1}^{\tau} \sum_{j=1}^{n_i} f_{ij} w_{ij} \left[u(r_{ij}) y_{ij} y_{ij}^T - \beta I_p \right] = 0$$

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 `iw` 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 $\epsilon = \text{tolerance}$, 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 spaced 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 estimates 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>

int 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);

    imsls_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>

int 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);

    imsls_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.

Analysis of Variance and Designed Experiments

Functions

General Analysis of Variance

One-way analysis of variance	anova_oneway	313
Analyzes a one-way classification model with covariates	ancovar	325
Analysis of variance for fixed effects balanced factorial designs.	anova_factorial	339
Nested random effects analysis of variance	anova_nested	349
Analysis of variance for balanced fixed, random, or mixed models.	anova_balanced	363

Designed Experiments

Analysis of balanced and unbalanced completely randomized factorial experiments	crd_factorial	375
Analysis of balanced and unbalanced randomized complete block factorial experiments	rcbd_factorial	386
Analysis of latin-square experiments	latin_square	396
Analysis of balanced and partially-balanced data from lattice experiments	lattice	405
Analysis of split-plot experiments	split_plot	422
Analysis of split-split-plot experiments.	split_split_plot	436
Analysis of strip-plot experiments	strip_plot	452
Analysis of strip-split-plot experiments	strip_split_plot	463

Utilities

Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.	homogeneity	484
Multiple comparisons of means	multiple_comparisons	491
False discovery rates.	false_discovery_rates	497
Yates' method for estimating missing observations in designed experiments.	yates	505

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 functions 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.

Table 1 – 2x2 Factorial Experiment

Treatment	Additive A	Additive B
1	No	No
2	Yes	No
3	No	Yes
4	Yes	Yes

This is referred to as a 2x2 or 2² 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. Function `imsls_f_rcbd_factorial` 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 3×3 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 $b = n_locations \cdot r \cdot (k - 1) + 1$. For a balanced-lattice, $b = r \cdot k = (k + 1) \cdot k = (\sqrt{t} + 1) \cdot \sqrt{t} = 4 \cdot 3 = 12$.

Table 2 – A 3x3 Balanced-Lattice for Nine Treatments in Four Replicates

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 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)

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.

Table 3 – The Anova Table for a Lattice Experiment at One Location

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	$r \cdot t - 1$	SSR	

Table 4 – The Anova Table for a Lattice Experiment at Multiple Locations

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
TOTAL	$p(k - 1)(r \cdot k - k - 1)$	SSTot	

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.

Table 5 – A Latin Square Design for t=4 Treatments

		Test Vehicle			
		1	2	3	4
Test	1	A	C	B	D
	2	D	B	A	C
Bay	3	C	A	D	B
	4	B	D	C	A

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.

Table 6 – Latin Square Anova Table for Octane Experiment

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			

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 test-

ing 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](#).

Table 7 – Completely Randomized Experiments – Both Factors Randomized

W3F2	W1F3	W4F1	W2F1
W2F3	W1F1	W1F3	W1F2
W2F2	W3F1	W2F1	W4F2
W3F2	W1F1	W2F3	W1F2
W4F1	W3F2	W3F2	W4F3
W4F3	W3F1	W2F2	W4F2

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.

Table 8 – Randomized Complete Block Experiments – Both Factors Randomized Within a Block

Block 1	W3F3	W1F3	W4F1	W4F3
	W2F3	W1F1	W3F2	W1F2
	W2F2	W3F1	W2F1	W4F2
Block 2	W3F2	W1F1	W2F3	W1F2
	W4F1	W1F3	W3F2	W4F3

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.

Table 9 – A Split-Plot Experiment for Wheat (W) and Fertilizer (F)

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

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.

Table 10 – Strip-Plot Experiments – Strip-Plots Completely Crossed

		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 11 – Split-Plot Experiments – Split-Plots Nested within Strip-Plots

Whole-Plot Factor			
A2	A1	A4	A3
A2B1	A1B3	A4B1	A3B3
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

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 5, “Split-Plot Randomization.”](#) 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.

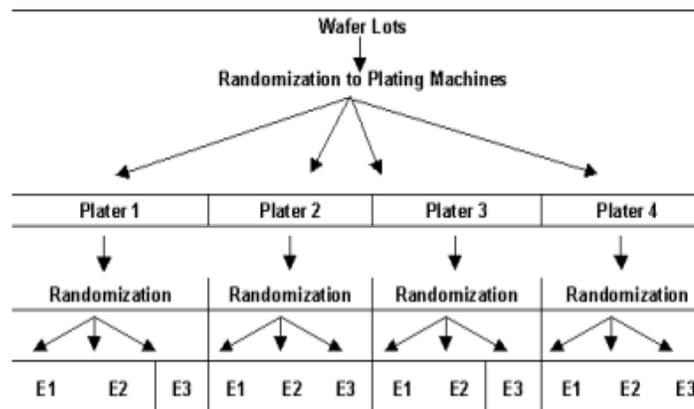


Figure 5, Split-Plot Randomization

Notice that although there are only three 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 6, "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.

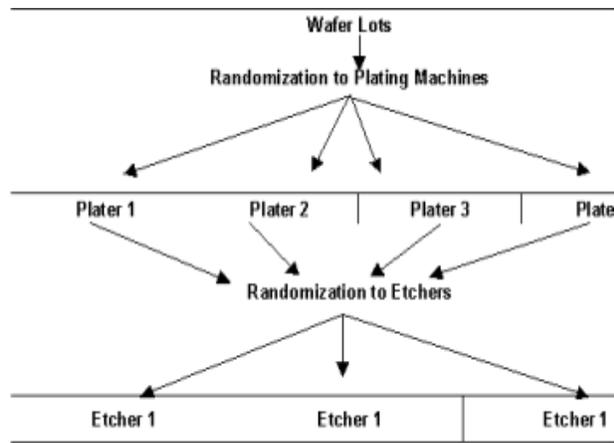


Figure 6, Strip-Plot Semiconductor Experiment

Strip-plot experiments can be analyzed using `imspls_f_strip_plot`. Function `imspls_f_strip_plot` returns a strip-plot Anova table with the following general structure:

Table 12 – Strip-Plot Anova Table for Semiconductor Experiment

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

Table 12 – Strip-Plot Anova Table for Semiconductor Experiment (Continued)

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			

Split-Split Plot 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 13](#), 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.

Table 13 – Split-Plot Experiment – Split-Plot B Nested within Whole-Plot A

Whole-Plot Factor			
A2	A1	A4	A3
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B3

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.

Table 14 – Split-Split Plot Experiment

Whole-Plot Factor			
A2	A1	A4	A3
A2B3C2	A1B2C1	A4B1C2	A3B3C2
A2B3C1	A1B2C2	A4B1C1	A3B3C1

Table 14 – Split-Split Plot Experiment (Continued)

A2B1C1	A1B1C1	A4B3C2	A3B2C2
A2B1C2	A1B1C2	A4B3C1	A3B2C1
A2B2C2	A1B3C1	A4B2C1	A3B1C2
A2B2C1	A1B3C2	A4B2C2	A3B1C1

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 a 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.

Table 15 – Strip-Split Plot Experiment, Split-Plots Nested Within Strip-Plot Factors A and B

		A2	A1	A4	A3
Factor B Strip Plot	B3	A2B3C2	A1B3C1	A4B3C2	A3B3C2
		A2B3C1	A1B3C2	A4B3C1	A3B3C1
	B1	A2B1C1	A1B1C1	A4B1C2	A3B1C2
		A2B1C2	A1B1C2	A4B1C1	A3B1C1
	B2	A2B2C2	A1B2C1	A4B2C1	A3B2C2
		A2B2C1	A1B2C2	A4B2C2	A3B2C1

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, functions `imsls_f_shapiro_wilk_normality_test`, `imsls_f_lilliefors_normality_test`, and `imsls_f_cgi_squared_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 $\sigma_1 = \sigma_2 = \dots = \sigma_i$, where σ_i is the standard deviation of the observational error for the i -th 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.

Multiple Testing

It is well known that the risk of committing a Type I error (i.e., a false positive or false discovery) increases with the number of tests being performed, even if all the tests use the same cut-off value for significance. The purpose of a multiple testing correction is to control the risk of false positives when performing multiple tests of significance. Two functions are included in this chapter to correct for multiple tests: `imsls_f_multiple_comparisons` performs multiple comparisons of treatment means using one of four methods to control the overall Type I error rate, and `imsls_f_false_discovery_rates` calculates false discovery rates and associated q -values, given a set of p -values resulting from multiple independent tests.

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 `ims1s_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_groups - 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_SCHEFFFE, 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_SCHEFFFE_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:

	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.

	Analysis of Variance Statistics
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).

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`IMSL_S_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)

Storage for array `anova_table` is provided by the user. See `IMSL_S_ANOVA_TABLE`.

`IMSL_S_GROUP_MEANS`, *float ***`means` (Output)

Address of a pointer to an internally allocated array of length `n_groups` containing the group means.

`IMSL_S_GROUP_MEANS_USER`, *float* `means[]` (Output)

Storage for array `means` is provided by the user. See `IMSL_S_GROUP_MEANS`.

`IMSL_S_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.

`IMSL_S_GROUP_STD_DEVS_USER`, *float* `std_devs[]` (Output)

Storage for array `std_devs` is provided by the user. See `IMSL_S_STD_DEVS`.

`IMSL_S_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.

`IMSL_S_GROUP_COUNTS_USER`, *int* `counts[]` (Output)

Storage for array `counts` is provided by the user. See `IMSL_S_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

$$\left(\frac{\text{ngroups}}{2} \right) \times 5$$

internally allocated array containing the statistics relating to the difference of means.

	Description
0	Group number for the <i>i</i> -th mean.
1	Group number for the <i>j</i> -th mean.
2	Difference of means (<i>i</i> -th mean) – (<i>j</i> -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

`IMSL_S_DUNN_SIDAK_USER, float ci_diff_means[]` (Output)

or

`IMSL_S_BONFERRONI_USER, float ci_diff_means[]` (Output)

or

`IMSL_S_SCHEFFE_USER, float ci_diff_means[]` (Output)

or

`IMSL_S_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_oweway` performs an analysis of variance of responses from a oneway classification design. The model is

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

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_oweway` 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_oweway` computes simultaneous confidence intervals on all

$$k^* = \frac{k(k-1)}{2}$$

pairwise comparisons of k means $\mu_1, \mu_2, \dots, \mu_k$ 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

$$\alpha = 1 - \frac{\text{confidence}}{100.0}$$

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 ($n_1 = n_2 = \dots = n_k = n$) 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 $(1 - \alpha)$ 100. Hayter's proof gave further support to earlier recommendations for its use (Stoline 1981). (Methods that are currently better are restricted to special cases and only offer improvement in severely unbalanced cases; see, for example, Spurrier and Isham 1985.) The formula for the difference $\mu_i - \mu_j$ is given by the following:

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

Dunn-Šidák method: The Dunn-Šidák method is a conservative method. The method gives wider intervals than the Tukey-Kramer method. (For large v and small α and k , the difference is only slight.) The method is slightly better than the Bonferroni method and is based on an improved Bonferroni (multiplicative) inequality (Miller 1980, pp. 101, 254–255). The method uses the t distribution (see function `imsls_f_t_inverse_cdf`, Chapter 11, [Probability Distribution Functions and Inverses](#)). The formula for the difference $\mu_i - \mu_j$ is given by

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

where $t_{f, v}$ is the 100 f percentage point of the t distribution with v 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:

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

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, [Probability Distribution Functions and Inverses](#)). The formula for the difference $\mu_i - \mu_j$ is given by

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

where $F_{1-\alpha; (k-1), \nu}$ is the $(1 - \alpha)$ 100 percentage point of the F distribution with $k - 1$ and ν degrees of freedom.

One-at-a-Time t method (Fisher's LSD): The One-at-a-Time t method is appropriate for constructing a single confidence interval. The confidence percentage input is appropriate for one interval at a time. The method has been used widely in conjunction with the overall test of the null hypothesis $\mu_1 = \mu_2 = \dots = \mu_k$ 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:

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

Examples

Example 1

This example computes a one-way analysis of variance for data discussed by Searle (1971, Table 5.1, pp. 165–179). The responses are plant weights for six plants of three different types—three normal, two off-types, and one aberrant. The responses are given by type of plant in the following table:

101	84	32
105	88	
94		

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

int 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\n", 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>

int 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_owenay (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, "%11.4f",
    0);
}

```

Output

```

    * * * Analysis of Variance * * *
degrees of freedom for among groups      2.0000
degrees of freedom for within groups     3.0000
total (corrected) degrees of freedom     5.0000
sum of squares for among groups          3480.0000
sum of squares for within groups          70.0000
total (corrected) sum of squares         3550.0000
among mean square                        1740.0000
within mean square                       23.3333
F-statistic                              74.5714
p-value                                  0.0028
R-squared (in percent)                   98.0282
adjusted R-squared (in percent)          96.7136
est. standard deviation of within error   4.8305
overall mean of y                        84.0000
coefficient of variation (in percent)     5.7505

```

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.

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>

int 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, "%11.4g",
    0);
}

```

Output

```

 * * * Analysis of Variance * * *
degrees of freedom for among groups          4
degrees of freedom for within groups        15
total (corrected) degrees of freedom        19
sum of squares for among groups             1242
sum of squares for within groups            150.78
total (corrected) sum of squares            1393
among mean square                           310.56
within mean square                           10.05
F-statistic                                  30.90
p-value                                      4.398e-007
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)        8.332

 * * * 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

ancovar

Analyzes a one-way classification model with covariates.

Synopsis

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

```
float *imsls_f_ancovar (int ngroup, int ncov, int ni[ ], float y[ ], float x[ ],..., 0)
```

The type *double* function is `imsls_d_ancovar`.

Required Arguments

int `ngroup` (Input)

The number of treatment groups.

int `ncov` (Input)

The number of covariates.

int `ni[]` (Input)

Array of length `ngroup` containing the number of responses for each group.

float `y[]` (Input)

Array of length `n` containing the data for the response variable where
 $n = ni[0] + ni[1] + \dots + ni[ngroup-1]$.

float `x[]` (Input)

Array of size `n` by `ncov` containing the data for the covariates.

Return Value

Pointer to an array of length 15 containing the one-way analysis of covariance assuming parallelism, organized as follows:

	Anova Table Value
0	Degrees of freedom for model (groups + covariates).
1	Degrees of freedom for error.

	Anova Table Value
2	Total (corrected) degrees of freedom.
3	Sum of squares for model (groups and covariates combined).
4	Sum of squares for error.
5	Total (corrected) sum of squares.
6	Model mean square (groups and covariates combined).
7	Error mean square.
8	<i>F</i> -statistic.
9	<i>p</i> -value.
10	R^2 (in percent).
11	Adjusted R^2 (in percent).
12	Estimate of the standard deviation.
13	Overall response mean.
14	Coefficient of variation (in percent).

Note that the *p*-value is returned as 0.0 when the value is so small that all significant digits have been lost.

Synopsis with Optional Arguments

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

```
float *imsls_f_ancovar (int ngroup, int ncov, int ni [ ], float y [ ], float x [ ],
    IMSLS_N_MISSING, int *nmiss,
    IMSLS_ADJ_ANOVA, float **adj_aov,
    IMSLS_ADJ_ANOVA_USER, float adj_aov [ ],
    IMSLS_PARALLEL_TESTS, float **testpl,
    IMSLS_PARALLEL_TESTS_USER, float testpl [ ],
    IMSLS_XYMEAN, float **xymean,
    IMSLS_XYMEAN_USER, float xymean [ ],
    IMSLS_COEF, float **coef,
```

```

    IMSLS_COEF_USER, float coef[ ],
    IMSLS_COEF_TABLES, float **coef_tables,
    IMSLS_COEF_TABLES_USER, float coef_tables[ ],
    IMSLS_REG_ANOVA, float **aov_tables,
    IMSLS_REG_ANOVA_USER, float aov_tables[ ],
    IMSLS_R_MATRIX, float **r,
    IMSLS_R_MATRIX_USER, float r[ ],
    IMSLS_COV_MEANS, float **covm,
    IMSLS_COV_MEANS_USER, float covm[ ],
    IMSLS_COV_COEF, float **covb,
    IMSLS_COV_COEF_USER, float covb[ ],
    IMSLS_RETURN_USER, float aov[ ],
    0)

```

Optional Arguments

IMSLN_MISSING, *int *nmiss* (Output)

The number of cases with missing values in **x** or **y** is returned in ***nmiss**. Cases with any missing values are not used in the analysis.

IMSLN_ADJ_ANOVA, *float **adj_aov* (Output)

Address of a pointer to an internally allocated array of length 8 containing the partial sum of squares for the one-way analysis of covariance organized as follows:

i	adj_aov[i]
0	Degrees of freedom for groups after covariates.
1	Degrees of freedom for covariates after groups.
2	Sum of squares for groups after covariates.
3	Sum of squares for model (groups and covariates combined).
4	F-statistic for groups.

i	adj_aov[i]
5	<i>F</i> -statistic for covariates.
6	<i>p</i> -value for groups.
7	<i>p</i> -value for covariates.

Note that the *p*-values are returned as 0.0 when the values are so small that all significant digits have been lost.

IMSLS_ADJ_ANOVA_USER, *float* adj_aov[] (Output)

Storage for the array `adj_aov` provided by the user. See `IMSLS_ADJ_ANOVA` for a description.

IMSLS_PARALLEL_TESTS, *float ***testp1 (Output)

Address of a pointer to an internally allocated array of length 10 containing the parallelism tests for the one-way analysis of covariance organized as follows:

i	testp1[i]
0	Extra degrees of freedom for model not assuming parallelism.
1	Degrees of freedom for error for model not assuming parallelism.
2	Degrees of freedom for error for model assuming parallelism.
3	Extra sum of squares for model not assuming parallelism.
4	Sum of squares for error for model not assuming parallelism.
5	Sum of squares for error for model assuming parallelism.
6	Mean square for <code>testp1[0]</code> .
7	Mean square for <code>testp1[1]</code> .
8	<i>F</i> -statistic.
9	<i>p</i> -value.

IMSLS_PARALLEL_TESTS_USER, *float* testp1[] (Output)

Storage for the array `testp1` provided by the user. See `IMSLS_PARALLEL_TESTS` for a description.

IMSLS_XYMEAN, *float **xymean* (Output)

Address of a pointer to an internally allocated array of size **ngroup+1** by **ncov+3** containing the unadjusted means for the covariates and the response variate and the means for the response variate adjusted for the covariates. Each row for **i = 0, 1, ..., ngroup-1** corresponds to a group. Row **ngroup** contains overall statistics. The means are organized in **xymean** columns as follows:

0	Number of non-missing cases
1 thru ncov	Covariate means.
ncov + 1	Response mean.
ncov + 2	Response mean adjusted assuming parallelism.

IMSLS_XYMEAN_USER, *float xymean[]* (Output)

Storage for the array **xymean** provided by the user. See **IMSLS_XYMEAN** for a description.

IMSLS_COEF, *float **coef* (Output)

Address of a pointer to an internally allocated array of size **ngroup + ncov** by 4 containing statistics for the regression coefficients for the model assuming parallelism. Each row corresponds to a coefficient in the model. For **i = 0, 1, ..., ngroup-1**, row **i** is for the **y** intercept for the *i*-th group. The remaining **ncov** rows are for the covariate coefficients. The statistics in the columns are organized as follows:

0	Coefficient estimate.
1	Estimated standard error of the estimate.
2	<i>t</i> -statistic.
3	<i>p</i> -value.

IMSLS_COEF_USER, *float coef[]* (Output)

Storage for the array **coef** provided by the user. See **IMSLS_COEF** for a description.

IMSLS_COEF_TABLES, *float **coef_tables* (Output)

Address of a pointer to an internally allocated array of size **ngroup** by **ncov+1** by 4 containing statistics for a linear regression model fitted separately for each of the **ngroup** treatment groups. This array can be viewed as a 3 dimensional array with **ngroup** rows, **ncov+1** columns, and depth of 4. Each row corresponds to one of the **ngroup** treatment groups. Each column corresponds to the model coefficients.

For column = 0, the statistics relate to the intercept in the regression model. For column = 1, 2, ..., `ncov`, the statistics relate to the slopes for the covariates. The depth dimension corresponds to the columns described for `IMSLS_COEF` as follows:

0	Coefficient estimate.
1	Estimated standard error of the estimate.
2	<i>t</i> -statistic.
3	<i>p</i> -value.

`IMSLS_COEF_TABLES_USER`, *float coef_tables[]* (Output)

Storage for the array `coef_tables` provided by the user. See `IMSLS_COEF_TABLES` for a description.

`IMSLS_REG_ANOVA`, *float **aov_tables* (Output)

Address of a pointer to an internally allocated array of size `ngroup` by 15 containing the analysis of variance tables for each linear regression model fitted separately to each treatment group. The 15 values in the *i*-th row are for treatment group *i* organized as follows:

<i>j</i>	<code>aov_tables[i*15+j]</code>
0	Degrees of freedom for regression model (covariates).
1	Degrees of freedom for error.
2	Total (corrected) degrees of freedom.
3	Sum of squares for regression model.
4	Sum of squares for error.
5	Total (corrected) sum of squares.
6	Model mean square.
7	Error mean square.
8	<i>F</i> -statistic.
9	<i>p</i> -value.
10	R^2 (in percent).
11	Adjusted R^2 (in percent).
12	Error standard deviation.
13	Overall response mean.
14	Coefficient of variation (in percent).

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`IMSLS_REG_ANOVA_USER`, *float* `aov_tables[]` (Output)

Storage for the array `aov_tables` provided by the user. See `IMSLS_REG_ANOVA` for a description.

`IMSLS_R_MATRIX`, *float ***`r` (Output)

Address of a pointer to an internally allocated array of size `ngroup+ncov` by `ngroup + ncov` containing the R matrix from the QR decomposition. The R matrix is from the regression assuming parallelism.

`IMSLS_R_MATRIX_USER`, *float* `r[]` (Output)

Storage for the array `r` provided by the user. See `IMSLS_R_MATRIX` for a description.

`IMSLS_COV_MEANS`, *float ***`covm` (Output)

Address of a pointer to an internally allocated array of size `ngroup` by `ngroup` containing the estimated matrix of variances and covariances for the adjusted means assuming parallelism.

`IMSLS_COV_MEANS_USER`, *float* `covm[]` (Output)

Storage for the array `covm` provided by the user. See `IMSLS_COV_MEANS` for a description.

`IMSLS_COV_COEF`, *float ***`covb` (Output)

Address of a pointer to an internally allocated array of size `ngroup + ncov` by `ngroup+ncov` containing the estimated matrix of variances and covariances for the coefficients in `coef` returned using `IMSLS_COEF` or `IMSLS_COEF_USER`.

`IMSLS_COV_COEF_USER`, *float* `covb[]` (Output)

Storage for the array `covb` provided by the user. See `IMSLS_COV_COEF` for a description.

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

An array of length 15 provided by the user for the return value. See [Return Value](#) above for a description.

Description

Function `imsls_f_ancovar` performs analyses for models that combine the features of a one-way analysis of variance model with that of a multiple linear regression model. The basic one-way analysis of covariance model is

$$y_{ij} = \beta_{0i} + \beta_1 x_{ij1} + \beta_2 x_{ij2} + \dots + \beta_m x_{ijm} + \varepsilon_{ij} \quad i = 1, 2, \dots, ngroup; \quad j = 1, 2, \dots, n_i$$

where the observed value of y_{ij} constitutes the j -th response in the i -th group, β_{0i} denotes the y intercept for the regression function for the i -th group, $\beta_1, \beta_2, \dots, \beta_m$ are the regression coefficients for the covariates, and the ε_{ij} 's are independently distributed normal errors with mean zero and variance σ^2 . This model allows the regression function for each group to have different intercepts. However, the remaining m regression coefficients are the same for each group, i.e., the regression functions are parallel.

In practice, sometimes the regression functions are not parallel. In addition to estimates for the model assuming parallelism, `imsls_f_ancovar` computes estimates and summary statistics for the separate regressions for each group. These estimates can be examined using the optional arguments `IMSLS_COEF_TABLES` and `IMSLS_REG_ANOVA`.

Estimates for the β_{0i} 's and $\beta_1, \beta_2, \dots, \beta_m$ in the model assuming parallelism are returned using the optional argument `IMSLS_COEF`. Summary statistics are also computed for this model.

The adjusted group means, stored in the last column of `xymean`, are computed using the formula:

$$\hat{\beta}_{0i} + \hat{\beta}_1 \bar{x}_1 + \hat{\beta}_2 \bar{x}_2 + \dots + \hat{\beta}_{ncov} \bar{x}_{ncov}$$

The estimated covariance between the i_1 -th and i_2 -th adjusted group mean is given by

$$v_{i_1 i_2} + \sum_{r=1}^m \sum_{s=1}^m \bar{x}_r v_{k+r, k+s} \bar{x}_s + \sum_{r=1}^m \bar{x}_r v_{i_1, k+r} + \sum_{r=1}^m \bar{x}_r v_{i_2, k+r}$$

where v_{pq} is the entry in `covb[(p - 1)(ngroup + ncov) + q - 1]` and is the estimated covariance between the p -th and q -th estimated coefficients in the regression function.

A discussion of formulas and interpretations for the one-way analysis of covariance problem appears in most elementary statistics texts, e.g., Snedecor and Cochran (1967, *Chapter 14*).

Examples

Example 1

This example fits a one-way analysis of covariance model assuming parallelism using data discussed by Snedecor and Cochran (Table 14.6.1, pages 432–436). The responses are concentrations of cholesterol (in mg/100 ml) in the blood of two groups of women: women from Iowa and women from Nebraska. Age of a woman is the single

covariate. The cholesterol concentrations and ages of the women according to state are shown in the following table. (There are 11 Iowa women and 19 Nebraska women in the study. Only the first 5 women from each state are shown here.)

	Cholesterol		
46	181	18	137
52	228	44	173
39	182	33	177
65	249	78	241
54	259	51	225

There is no evidence from the data to indicate that the regression lines for cholesterol concentration as a function of age are not parallel for Iowa and Nebraska women (p -value is 0.5425). The parallel line model suggests that Nebraska women may have higher cholesterol concentrations than Iowa women. The cholesterol concentrations (adjusted for age) are 195.5 for Iowa women versus 224.2 for Nebraska women. The difference is 28.7 with an estimated standard error of

$$\sqrt{170.4 + 97.4 - 2(2.9)} = 16.1$$

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

int main()
{
  int ncov=1, ngroup=2;
  int ni[2] = {11, 19};
  float *testpl, *aov, *xymean, *covm;
  float y[30] = {
    181.0, 228.0, 182.0, 249.0, 259.0,
    201.0, 121.0, 339.0, 224.0, 112.0,
    189.0, 137.0, 173.0, 177.0, 241.0,
    225.0, 223.0, 190.0, 257.0, 337.0,
    189.0, 214.0, 140.0, 196.0, 262.0,
    261.0, 356.0, 159.0, 191.0, 197.0
  };
  float x[30] = {
    46.0, 52.0, 39.0, 65.0, 54.0,
    33.0, 49.0, 76.0, 71.0, 41.0,
    58.0, 18.0, 44.0, 33.0, 78.0,
    51.0, 43.0, 44.0, 58.0, 63.0,
    19.0, 42.0, 30.0, 47.0, 58.0,
    70.0, 67.0, 31.0, 21.0, 56.0
  };
  aov = imsls_f_ancovar(ngroup, ncov, ni, y, x,
    IMSLS_PARALLEL_TESTS, &testpl,
    IMSLS_XYMEAN, &xymean,
```

```

        IMSLS_COV_MEANS, &covm, 0);
printf("
        * * * ANALYSIS OF VARIANCE * * * \n");
printf("
        Sum of          Mean          Prob of\n");
printf("Source   DF      Squares      Square      Overall F  Larger F\n");
printf("Model   %3.0f  %10.2f  %9.2f  %2.2f  %8.6f\n",
        aov[0], aov[3], aov[6], aov[8], aov[9]);
printf("Error   %3.0f  %10.2f  %9.2f  \n", aov[1], aov[4], aov[7]);
printf("Total   %3.0f  %10.2f  \n", aov[2], aov[5]);
printf("\n");
printf(" * * * TEST FOR PARALLELISM * * * \n");
printf(" Sum of Mean F Prob of\n");
printf("SOURCE DF Squares Square TEST Larger F\n");
printf("Extra due to\n");
printf("Nonparallelism %3.0f %10.2f %7.2f %7.5f %5.4f\n",
        testpl[0], testpl[3], testpl[6], testpl[8], testpl[9]);
printf("Extra Assuming\n");
printf("Nonparallelism %3.0f %10.2f %7.2f \n", testpl[1], testpl[4],
        testpl[7]);
printf("Error Assuming\n");
printf("Parallelism %3.0f %10.2f \n", testpl[2], testpl[5]);

imsls_f_write_matrix("\nXY Mean Matrix\n", ngroup+1, 4, xymean, 0);
imsls_f_write_matrix("\nVar./Covar. Matrix of Adjusted Group Means\n",
        ngroup, ngroup, covm, 0);
}

```

Output

```

        * * * ANALYSIS OF VARIANCE * * *
Source   DF      Sum of Squares      Mean Square      Overall F  Prob of
Model    2      54432.77      27216.39      14.97      0.000042
Error    27      49103.91      1818.66
Total    29      103536.69

```

```

        * * * TEST FOR PARALLELISM * * *
SOURCE   DF      Sum of Squares      Mean Square      F TEST      Prob of
Extra due to
Nonparallelism 1      709.05      709.05      0.38094      0.5425
Extra Assuming
Nonparallelism 26     48394.86      1861.34
Error Assuming
Parallelism    27     49103.91

```

```

        XY Mean Matrix
           1      2      3      4
1      11.0     53.1    207.7    195.5
2      19.0     45.9    217.1    224.2
3      30.0     48.6    213.7    213.7

```

```

        Var./Covar. Matrix of Adjusted Group Means
           1      2
1      170.4     -2.9
2      -2.9      97.4

```

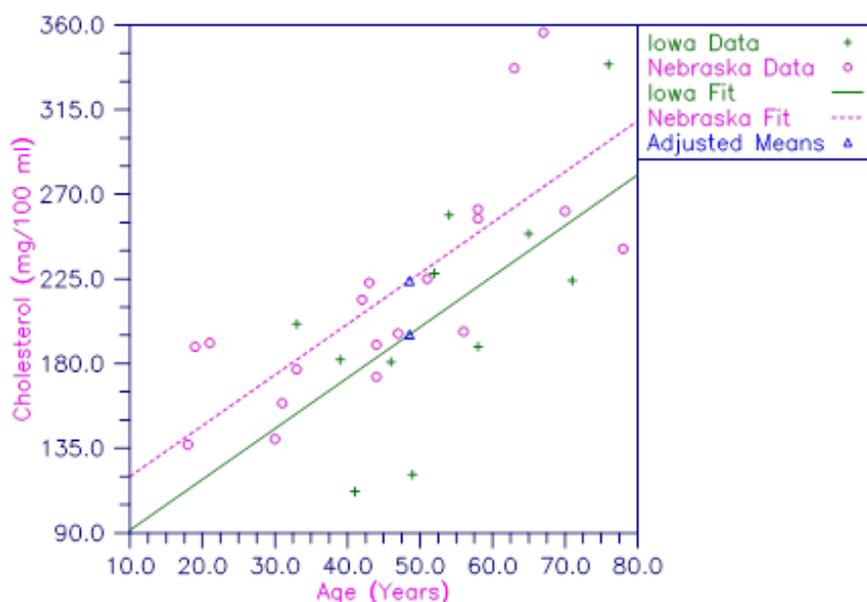


Figure 7, Plot of Cholesterol Concentrations and Fitted Parallel Lines by State

Example 2

This example fits a one-way analysis of covariance model and performs a test for parallelism using data discussed by [Snedecor and Cochran](#) (1967, Table 14.8.1, pages 438-443). The responses are weight gains (in pounds per day) of 40 pigs for four groups of pigs under varying treatments. Two covariates-initial age (in days) and initial weight (in pounds) are used. For each treatment, there are 10 pigs. Only the first five pigs from each treatment are shown here.

Treatment 1			Treatment 2			Treatment 3			Treatment 4		
Age	Wt.	Gain									
78	61	1.40	78	74	1.61	78	80	1.67	77	62	1.40
90	59	1.79	99	75	1.31	83	61	1.41	71	55	1.47
94	76	1.72	80	64	1.12	79	62	1.73	78	62	1.37
71	50	1.47	75	48	1.35	70	47	1.23	70	43	1.15
99	61	1.26	94	62	1.29	85	59	1.49	95	57	1.22

For these data, the test for non-parallelism is not statistically significant ($p = 0.901$). The one-way analysis of covariance test for the treatment means adjusted for the covariates, assuming parallel slopes, is statistically significant at a stated significance level of $\alpha = 0.05$, ($p = 0.04931$).

Multiple comparisons can be done using the least significant difference approach of comparing each pair of treatment groups with the two-sample student-t test. Since the adjusted means in the one-way analysis of covariance are correlated, the standard error for these comparisons must be computed using the variances and covariances in `covm`. The standard errors for these comparisons are fairly similar ranging from 0.0630 to 0.0638. The Student's t comparisons identify differences between groups 1 and 2, and 1 and 4 as being statistically significant with p -values of 0.01225 and 0.03854 respectively.

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

int main()
{
    int i, j;
    int ncov=2, ngroup=4, nob=40;
    int ni[4] = {10, 10, 10, 10};
    float aov[15], testpl[10], adj_aov[8], xyemean[5*5], covm[4*4];
    float x1[40] = {
        78.0, 90.0, 94.0, 71.0, 99.0, 80.0, 83.0, 75.0, 62.0, 67.0,
        78.0, 99.0, 80.0, 75.0, 94.0, 91.0, 75.0, 63.0, 62.0, 67.0,
        78.0, 83.0, 79.0, 70.0, 85.0, 83.0, 71.0, 66.0, 67.0, 67.0,
        77.0, 71.0, 78.0, 70.0, 95.0, 96.0, 71.0, 63.0, 62.0, 67.0
    };

    float x2[40] = {
        61.0, 59.0, 76.0, 50.0, 61.0, 54.0, 57.0, 45.0, 41.0, 40.0,
        74.0, 75.0, 64.0, 48.0, 62.0, 42.0, 52.0, 43.0, 50.0, 40.0,
        80.0, 61.0, 62.0, 47.0, 59.0, 42.0, 47.0, 42.0, 40.0, 40.0,
        62.0, 55.0, 62.0, 43.0, 57.0, 51.0, 41.0, 40.0, 45.0, 39.0,
    };

    float y[40] = {
        1.40, 1.79, 1.72, 1.47, 1.26, 1.28, 1.34, 1.55, 1.57, 1.26,
        1.61, 1.31, 1.12, 1.35, 1.29, 1.24, 1.29, 1.43, 1.29, 1.26,
        1.67, 1.41, 1.73, 1.23, 1.49, 1.22, 1.39, 1.39, 1.56, 1.36,
        1.40, 1.47, 1.37, 1.15, 1.22, 1.48, 1.31, 1.27, 1.22, 1.36
    };

    float x[40*2], stderror, delta, t, df, pvalue;

    /* setup covariate input matrix */
    for(i=0; i<nob; i++){
        x[i*ncov ] = x1[i];
        x[i*ncov+1] = x2[i];
    }

    imsls_f_ancovar(ngroup, ncov, ni, y, x,
        IMSLS_PARALLEL_TESTS_USER, testpl,
        IMSLS_ADJ_ANOVA_USER, adj_aov,
        IMSLS_XYMEAN_USER, xyemean,
        IMSLS_COV_MEANS_USER, covm,
```

```

    IMSLS_RETURN_USER, aov,
    0);

printf("\n");
printf("          * * * TEST FOR PARALLELISM * * * \n");
printf("                Sum of      Mean      F      Prob "
      "of\n");
printf("SOURCE          DF      Squares      Square      TEST      Larger "
      "F\n");
printf("Extra due to\n");
printf("Nonparallelism %3.0f %10.2f      %7.2f      %7.5f      %5.3f\n",
      testpl[0], testpl[3], testpl[6], testpl[8], testpl[9]);
printf("Extra Assuming\n");
printf("Nonparallelism %3.0f %10.2f      %7.2f \n",
      testpl[1], testpl[4], testpl[7]);
printf("Error Assuming\n");
printf("Parallelism      %3.0f %10.2f      \n", testpl[2], testpl[5]);

printf("\n");
printf("          * * * ANALYSIS OF VARIANCE * * * \n");
printf("                Sum of      Mean      Prob "
      "of\n");
printf("Source   DF      Squares      Square      Overall F      Larger"
      "F\n");
printf("Model    %3.0f      %f      %f      %f      %5.4f\n",
      aov[0], aov[3], aov[6], aov[8], aov[9]);
printf("Error   %3.0f      %f      %f      \n", aov[1], aov[4],
      aov[7]);
printf("Total   %3.0f      %f      \n", aov[2], aov[5]);

printf("\n");
printf("          * * * ADJUSTED ANALYSIS OF VARIANCE * * * \n");
printf("                Sum of      F      Prob "
      "of\n");
printf("Source          DF      Squares      TEST      Larger "
      "F\n");
printf("Groups after Covariates %3.0f      %10.2f      %5.2f      %7.5f\n",
      adj_aov[0], adj_aov[2], adj_aov[4], adj_aov[6]);
printf("Covariates after Groups %3.0f      %10.2f      %5.2f      %7.5f\n",
      adj_aov[1], adj_aov[3], adj_aov[5], adj_aov[7]);

printf("\n          * * * GROUP MEANS * * * \n");
printf("GROUP | Unadjusted | Adjusted | Std. Error\n");

for(i=0; i<ngroup; i++) {
    stderr = sqrt(covm[i*ngroup + i]);
    printf(" %d | %5.4f | %5.4f | %7.4f\n", i+1,
        xymean[i*(ngroup+1)+ngroup-1], xymean[i*(ngroup+1)+ngroup],
        stderr);
}

printf("\n          * * * STUDENT-T MULTIPLE COMPARISONS * * * \n");
printf("GROUPS | DIFF | Std. Error | Student-t | P-Value\n");

for(i=0; i<ngroup-1; i++){
    for(j=i+1; j<ngroup; j++){
        delta = xymean[i*(ngroup+1)+ngroup] -
            xymean[j*(ngroup+1)+ngroup];
        stderr = sqrt(covm[i*ngroup+i]+covm[j*ngroup+j]-
            2.0*covm[i*ngroup+j]);
    }
}

```

```

t      = delta/stderror;
df     = xyemean[i*(ngroup+1)]+xyemean[j*(ngroup+1)]-2;
pvalue = 1.0 - imsls_f_t_cdf(t, df);
printf(" %d vs %d | %7.4f | %7.4f | %7.3f | "
       "%7.5f\n", i+1, j+1, delta, stderror, t, pvalue);
    }
}
}

```

Output

* * * TEST FOR PARALLELISM * * *					
SOURCE	DF	Sum of Squares	Mean Square	F TEST	Prob of Larger F
Extra due to Nonparallelism	6	0.05	0.01	0.35534	0.901
Extra Assuming Nonparallelism	28	0.62	0.02		
Error Assuming Parallelism	34	0.67			

* * * ANALYSIS OF VARIANCE * * *					
Source	DF	Sum of Squares	Mean Square	Overall F	Prob of Larger F
Model	5	0.352517	0.070503	3.576395	0.0105
Error	34	0.670261	0.019714		
Total	39	1.022778			

* * * ADJUSTED ANALYSIS OF VARIANCE * * *					
Source	DF	Sum of Squares	F TEST	Prob of Larger F	
Groups after Covariates	3	0.17	2.90	0.04931	
Covariates after Groups	2	0.17	4.44	0.01939	

* * * GROUP MEANS * * *			
GROUP	Unadjusted	Adjusted	Std. Error
1	1.4640	1.4614	0.0448
2	1.3190	1.3068	0.0446
3	1.4450	1.4429	0.0447
4	1.3250	1.3418	0.0449

* * * STUDENT-T MULTIPLE COMPARISONS * * *				
GROUPS	DIFF	Std. Error	Student-t	P-Value
1 vs 2	0.1546	0.0630	2.455	0.01225
1 vs 3	0.0185	0.0637	0.290	0.38750
1 vs 4	0.1196	0.0638	1.875	0.03854
2 vs 3	-0.1362	0.0632	-2.153	0.97743
2 vs 4	-0.0350	0.0638	-0.549	0.70528
3 vs 4	0.1011	0.0631	1.602	0.06330

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 (Input)

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:

	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.

	Analysis of Variance Statistics
4	Sum of squares for error.
5	Total (corrected) sum of squares.
6	Model mean square.
7	Error mean square.
8	Overall <i>F</i> -statistic.
9	<i>p</i> -value.
10	<i>R</i> ² (in percent).
11	Adjusted <i>R</i> ² (in percent).
12	Estimate of the standard deviation.
13	Overall mean of <i>y</i> .
14	Coefficient of variation (in percent).

Note that the *p*-value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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,

$$NEF = \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{\min(n, |model_order|)}$$

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:

	Description
0	Degrees of freedom.
1	Sum of squares.
2	<i>F</i> -statistic.
3	<i>p</i> -value.

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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 $(n_levels[0] + 1) \times (n_levels[1] + 1) \times \dots \times (n_levels[n - 1] + 1)$ 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

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk} \quad i = 1,2; \quad j = 1,2,3; \quad k = 1,2, \dots, 10$$

where

$$\sum_{i=1}^2 \alpha_i = 0; \sum_{j=1}^3 \beta_j = 0; \sum_{i=1}^2 \gamma_{ij} = 0 \text{ for } j = 1,2,3; \text{ and } \sum_{j=1}^3 \gamma_{ij} = 0$$

for $i = 1, 2$. The first responses in each cell in the two-way layout are given in the following table:

Protein 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>
#include <stdio.h>

int 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\n",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>
#include <stdio.h>

int 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:

	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>
#include <stdio.h>

int 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 * * *\n", 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 Squares	Mean Square	Prob. Of 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.

0	Unequal numbers in the subgroups.
1	Equal numbers in the subgroups.

int `n_levels[]` (Input)

Array with the number of levels for each factor.

If `equal_option = 1`, `n_levels` is of length `n_factors` and contains the number of levels for each of the factors.

Example: Suppose there are 3 factors, A, B, and C. A has two levels (A1, A2), B has 3 levels at each level of A, and C has 2 levels at each level of B. `n_levels = {2,3,2}` and the number of observations is `nobs = 2 × 3 × 2 = 12`.

	A levels	B levels	C levels	y indices	
	1	1	1	0	
			2	1	
		2	1	1	2
				2	3
		3	1	1	4
				2	5
	2	1	1	6	
			2	7	
		2	1	1	8
				2	9
		3	1	1	10
2				11	
<code>n_levels</code>	2	3	2		
<code>nobs</code>	2×	3×	2	=12	
<code>equal_option = 1</code> example					

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.

Example: Suppose there are 3 factors, A, B, and C, with C nested in B and B nested in A. A has two levels (A1, A2), B has up to 3 levels, and C has up to 2 levels. In the `equal_option = 0` case, the function needs to know explicitly how the number of levels varies throughout. As specified in the table, A has two levels (`n_levels[0] = 2`), B has 3 levels in level 1 of A (`n_levels[1] = 3`) and 2 levels in level 2 of A (`n_levels[2] = 2`). Similarly, factor C has 2 levels in the A1-B1 and A1-B2 combinations (`n_levels[3] = 2`, `n_levels[4] = 2`), but only 1 level in the A1-B3 combination (`n_levels[5] = 1`). `n_levels = {2,3,2,2,1,1,2}` and the number of observations is the sum of the number of levels in the last factor, C, `nobs = 2 + 2 + 1 + 1 + 2 = 8`:

	A levels	B levels	C levels	y indices	
n_levels nobs	1	1	1	0	
				2	1
		2	1	2	2
			2	3	
		3	1	4	
	2	1	1	5	
		2	1	6	
			2	7	
	2	3, 2	2, 2, 1, 1, 2		
			2 + 2 + 1 + 1 + 2	= 8	
equal_option = 0 example					

float **y**[] (Input)
 Array of length **nobs** containing the responses.

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[],
```

```

IMSL_S_Y_MEANS, float **y_means,

IMSL_S_Y_MEANS_USER, float y_means[ ],

0)

```

Optional Arguments

`IMSL_S_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:

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).

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`IMSL_S_ANOVA_TABLE_USER`, *float anova_table[]* (Output)

Storage for array `anova_table` is provided by the user. See `IMSL_S_ANOVA_TABLE`

`IMSL_S_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:

	Description
0	Degrees of freedom.
1	Sum of squares.
2	Mean squares.
3	<i>F</i> -statistic.
4	<i>p</i> -value for <i>F</i> test.
5	Variance component estimate.
6	Percent of variance of variance explained by variance component.
7	Lower endpoint for a confidence interval on the variance component.
8	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**-1) *9+3] and **variance_components** [(**n_factors**-1) *9+4] are set to NaN (not a number). Note that the *p*-value for the *F* test is returned as 0.0 when the value is so small that all significant digits have been lost.

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 **n_factors** * (**n_factors** +1) / 2 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.

	<code>y_means</code>
0	1 + sum of values in <code>n_levels</code> for the first (<code>n_factors-1</code>) factors
1	1 + <code>n_levels[0]</code> + <code>n_levels[0] * n_levels[1]</code> + ... + <code>n_levels[0] * n_levels[1] * ... * n_levels[n_factors - 2]</code> .

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[]* (Output)

Storage for array `y_means` is provided by the user. See `IMSLS_Y_MEANS`

Description

Function `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.

Examples

Example 1

An analysis of a three-factor nested random model with equal numbers in the subgroups is performed using data discussed by [Snedecor and Cochran](#) (1967, Table 10.16.1, pages 285–288). The responses are calcium concentrations (in percent, dry basis) as measured in the leaves of turnip greens. Four plants are taken at random, then three leaves are randomly selected from each plant.

Finally, from each selected leaf two samples are taken to determine calcium concentration. The model is

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

where y_{ijk} is the calcium concentration for the k -th sample of the j -th leaf of the i -th plant, the α_i 's are the plant effects and are taken to be independently distributed

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

the β_{ij} 's are leaf effects each independently distributed

$$N(0, \sigma_\beta^2)$$

and the ϵ_{ijk} 's are errors each independently distributed $N(0, \sigma^2)$. The effects are all assumed to be independently distributed. The data are given in the following table:

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>

int main()
{
    float pvalue, *aov, *varc, *ymean, *ems;
    float 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, "%11.4g",
    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, "%11.4g",
    0);
}

```

Output

```
pvalue = 0.000000
```

```

    * * * Analysis of Variance * * *
degrees of freedom for model          11
degrees of freedom for error          12
total (corrected) degrees of freedom  23
sum of squares for model              10.19
sum of squares for error              0.07985
total (corrected) sum of squares      10.27
model mean square                    0.9264
error mean square                    0.006655
F-statistic                          139.2
p-value                              6.769e-011
R-squared (in percent)               99.22
adjusted R-squared (in percent)      98.51
est. standard deviation of within error 0.08158
overall mean of y                    3.012
coefficient of variation (in percent) 2.708

```

```

    * * * 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
sum of squares for A                 7.56
mean square of A                     2.52

```

F-statistic for A	7.665
p-value for A	0.009725
Estimate of A	0.3652
Percent Variation Explained by A	68.53
95% Confidence Interval Lower Limit for A	0.03955
95% Confidence Interval Upper Limit for A	5.787
degrees of freedom for B	8
sum of squares for B	2.63
mean square of B	0.3288
F-statistic for B	49.41
p-value for B	5.092e-008
Estimate of B	0.1611
Percent Variation Explained by B	30.22
95% Confidence Interval Lower Limit for B	0.06967
95% Confidence Interval Upper Limit for B	0.6004
degrees of freedom for Error	12
sum of squares for Error	0.07985
mean square of Error	0.006655
F-statistic for Error
p-value for Error
Estimate of Error	0.006655
Percent Explained by Error	1.249
95% Confidence Interval Lower Limit for Error	0.003422
95% Confidence Interval Upper Limit for Error	0.01813

Example 2

An analysis of a three-factor nested random model with unequal numbers in the subgroups is performed. The data are given in the following table:

1	1	23.0	19.0	
	2	31.0	37.0	
2	1	33.0	29.0	
	2	29.0		
3	1	36.0	29.0	33.0
4	1	11.0	21.0	
	2	23.0	18.0	
	3	33.0		
	4	23.0		
	5	26.0		
	6	39.0		
	7	20.0		
	8	24.0		
	9	36.0		

5	1	25.0	33.0
6	1	28.0	31.0
	2	25.0	42.0
	3	32.0	36.0
	4	41.0	
	5	35.0	
	6	16.0	
	7	30.0	
	8	40.0	
	9	32.0	
	10	44.0	

```
#include <imsls.h>

int main()
{
    float *aov, *ems, *vc, *ymeans;
    float y[36] = {23.0, 19.0, 31.0, 37.0,
                  33.0, 29.0, 29.0,
                  36.0, 29.0, 33.0,
                  11.0, 21.0,
                  23.0, 18.0,
                  33.0, 23.0, 26.0, 39.0, 20.0, 24.0, 36.0,
                  25.0, 33.0,
                  28.0, 31.0,
                  25.0, 42.0,
                  32.0, 36.0,
                  41.0, 35.0, 16.0, 30.0, 40.0, 32.0, 44.0
    };
    int nl[32] = {
        6, /* Factor A */
        2, 2, 1, 9, 1, 10, /* Factor B */
        2, 2, /* Factor C */
        2, 1,
        3,
        2, 2, 1, 1, 1, 1, 1, 1, 1,
        2,
        2, 2, 2, 1, 1, 1, 1, 1, 1, 1
    };
    int i, ymeans_length;

    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", " A means 5",
    " A means 6", "AB means 1 1", "AB means 1 2",
    "AB means 2 1", "AB means 2 2", "AB means 3 1",
    "AB means 4 1", "AB means 4 2", "AB means 4 3",
    "AB means 4 4", "AB means 4 5", "AB means 4 6",
    "AB means 4 7", "AB means 4 8", "AB means 4 9",
    "AB means 5 1", "AB means 6 1", "AB means 6 2",
    "AB means 6 3", "AB means 6 4", "AB means 6 5",
    "AB means 6 6", "AB means 6 7", "AB means 6 8",
    "AB means 6 9", "AB means 6 10"
};
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"};

imsls_f_anova_nested (3, 0, nl, y,
    IMSLS_ANOVA_TABLE, &aov,
    IMSLS_EMS, &ems,
    IMSLS_VARIANCE_COMPONENTS, &vc,
    IMSLS_Y_MEANS, &ymeans,
    0);

imsls_f_write_matrix("***AnalysisofVariance ***", 15, 1, aov,
    IMSLS_ROW_LABELS, aov_labels,
    IMSLS_WRITE_FORMAT, "%10.5f",
    0);
imsls_f_write_matrix("***ExpectedMeanSquare Coefficients ***",
    6, 1, ems,
    IMSLS_ROW_LABELS, ems_labels,
    IMSLS_WRITE_FORMAT, "%6.2f",
    0);

/* sum level count for factors 1 and 2 */
ymean_length = 1;
for (i=0; i<=6;i++) ymean_length += nl[i];

```

```

imsls_f_write_matrix(" * * * Means ***", ymeans_length, 1, ymeans,
  IMSLS_ROW_LABELS, means_labels,
  IMSLS_WRITE_FORMAT, "%6.2f",
  0);
imsls_f_write_matrix(
  "*** Analysis of Variance / Variance Components **", 27, 1, vc,
  IMSLS_ROW_LABELS, components_labels,
  IMSLS_WRITE_FORMAT, "%10.5f",
  0);
}

```

Output

```

***AnalysisofVariance ***
degrees of freedom for model          24.00000
degrees of freedom for error          11.00000
total (corrected) degrees of freedom  35.00000
sum of squares for model              1810.80591
sum of squares for error              310.16650
total (corrected) sum of squares      2120.97241
model mean square                     75.45025
error mean square                     28.19695
F-statistic                           2.67583
p-value                               0.04587
R-squared (in percent)                85.37621
adjusted R-squared (in percent)       53.46977
est. standard deviation of within error 5.31008
overall mean of y                     29.52778
coefficient of variation (in percent) 17.98334

***ExpectedMeanSquare Coefficients ***
Effect A and Error                    1.00
Effect A and Effect B                 1.97
Effect A and Effect A                 5.38
Effect B and Error                    1.00
Effect B and Effect B                 1.29
Error and Error                       1.00

* * * Means ***
Grand mean                            29.53
A means 1                             27.50
A means 2                             30.33
A means 3                             32.67
A means 4                             24.91
A means 5                             29.00
A means 6                             33.23
AB means 1 1                          21.00
AB means 1 2                          34.00
AB means 2 1                          31.00
AB means 2 2                          29.00
AB means 3 1                          32.67
AB means 4 1                          16.00
AB means 4 2                          20.50
AB means 4 3                          33.00
AB means 4 4                          23.00
AB means 4 5                          26.00
AB means 4 6                          39.00
AB means 4 7                          20.00

```

AB means 4 8	24.00
AB means 4 9	36.00
AB means 5 1	29.00
AB means 6 1	29.50
AB means 6 2	33.50
AB means 6 3	34.00
AB means 6 4	41.00
AB means 6 5	35.00
AB means 6 6	16.00
AB means 6 7	30.00
AB means 6 8	40.00
AB means 6 9	32.00
AB means 6 10	44.00
* * Analysis of Variance / Variance Components * *	
degrees of freedom for A	5.00000
sum of squares for A	461.42230
mean square of A	92.28446
F-statistic for A	0.98770
p-value for A	0.46007
Estimate of A	-0.21371
Percent Variation Explained by A
95% Confidence Interval Lower Limit for A
95% Confidence Interval Upper Limit for A
degrees of freedom for B	19.00000
sum of squares for B	1349.38354
mean square of B	71.02019
F-statistic for B	2.51872
p-value for B	0.05965
Estimate of B	33.19880
Percent Variation Explained by B	54.07344
95% Confidence Interval Lower Limit for B	0.00000
95% Confidence Interval Upper Limit for B	100.58640
degrees of freedom for Error	11.00000
sum of squares for Error	310.16650
mean square of Error	28.19695
F-statistic for Error
p-value for Error
Estimate of Error	28.19695
Percent Explained by Error	45.92656
95% Confidence Interval Lower Limit for Error	14.14990
95% Confidence Interval Upper Limit for Error	81.28591

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_effect[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,
```

```
    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:

	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 \bar{y} .
14	Coefficient of variation (in percent).

Note that the p -value is returned as 0.0 when the value is so small that all significant digits have been lost.

`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	
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.

0	Degrees of freedom.
1	Sum of squares.
2	Mean squares.
3	F-statistic.
4	p-value for F test.
5	Variance component estimate.
6	Percent of variance of variance explained by variance component.
7	Lower endpoint for a confidence interval on the variance component.
8	Upper endpoint for a confidence interval on the variance component.

Elements 5 through 8 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 7 and 8 contain NaN. Note that the p-value for the F test is returned as 0.0 when the value is so small that all significant digits have been lost.

`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
 $(n_model_effects + 1) \times (n_model_effects + 2) / 2$ 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[3]
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
 $(n_levels[0] + 1) \times (n_levels[1] + 1) \times \dots \times (n_levels[n_factors-1] + 1)$ 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*:

$$y_{ijk} = \mu + \alpha_i + b_j + c_{ij} + e_{ijk} \quad i = 1, 2, \dots, a; j = 1, 2, \dots, b; k = 1, 2, \dots, n$$

The fixed effects α_i 's are subject to the restriction

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

the b_j 's are random effects identically and independently distributed

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

c_{ij} are interaction effects each distributed

$$N\left(0, \frac{a-1}{a} \sigma_{AB}^2\right)$$

and are subject to the restrictions

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

and the e_{ijk} 's are errors identically and independently distributed $N(0, \sigma^2)$. In general, interactions of fixed and random factors have sums over subscripts corresponding to fixed factors equal to zero. Also in general, the variance of a random interaction effect is the associated variance component times a product of ratios for each fixed factor in the random interaction term. Each ratio depends on the number of levels in the fixed factor. In the earlier example, the random interaction AB has the ratio $(a-1)/a$ as a multiplier of

$$\sigma_{AB}^2$$

and

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

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 $\text{mode1} = 0$. 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,

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

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` column 6) 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

$$\% \text{variation}(AB | \text{Model} = 1) = \frac{\frac{a-1}{a}\sigma_{AB}^2}{\sigma_B^2 + \frac{a-1}{a}\sigma_{AB}^2 + \sigma^2}$$

while for the parameterization `model = 0`, the percentage is computed using the formula

$$\% \text{variation}(\text{AB} | \text{Model} = 0) = \frac{\tilde{\sigma}_{AB}^2}{\tilde{\sigma}_B^2 + \tilde{\sigma}_{AB}^2 + \sigma^2}$$

In each case, the variance components are replaced by their estimates (stored in `variance_components` column 5).

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

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

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

where y_{ijk} is the response for the k -th experimental unit in block j with treatment i ; the α_i 's are the treatment effects and are subject to the restriction

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

the b_j 's are block effects identically and independently distributed

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

c_{ij} are interaction effects each distributed

$$N\left(0, \frac{3}{4}\sigma_{AB}^2\right)$$

and are subject to the restrictions

$$\sum_{i=1}^4 c_{ij} = 0 \quad \text{for } j = 1, 2, 3, 4$$

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:

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>

int 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, *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
degrees of freedom for error                16
total (corrected) degrees of freedom        31
  sum of squares for model                  216.5
  sum of squares for error                  19
total (corrected) sum of squares            235.5
model mean square                           14.433
error mean square                           1.1875
F-statistic                                 12.154
p-value                                     4.9182e-006
R-squared (in percent)                      91.932
adjusted R-squared (in percent)             84.368
est. standard deviation of within error     1.0897
overall mean of y                           5.375
coefficient of variation (in percent)       20.27345

* * * 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. Function `crd_factorial` does permit a factorial treatment structure. However, unlike `anova_factorial`, function `crd_factorial` allows for missing data 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 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:

$$n_anova = a + \sum_{i=1}^m \binom{n_factors}{i}$$

where

$$a = \begin{cases} 2 & \text{if } n_locations = 1 \\ 3 & \text{if } n_locations > 1 \text{ and treatments are not replicated} \\ 4 & \text{if } n_locations = 1 \text{ and treatments are not replicated at each location} \end{cases}$$

and $m = model_order$.

Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, `anova_tablei,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:

<code>j</code>	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

Note that the *p*-value for the *F*-statistic is returned as 0.0 when the value is so small that all significant digits have been lost.

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_tableij` 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 ‡
.	.
.	.
.	.
<code>-n_factors</code>	<code>(n_factors)</code> -way Interactions ‡
<code>-n_factors-1</code>	Effects Error Term
<code>-n_factors-2</code>	Residual †
<code>-n_factors-3</code>	Corrected Total
-1	Main Effects †

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

$$\binom{n_factors}{n_way}$$

The order of these terms is in ascending order by treatment subscript. The interactions for factor 1 appear first, followed by factor 2, factor 3, and so on.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_crd_factorial (int n_obs, int n_locations, int n_factors,
                             int n_levels[], int model[], float y[],
```

```

IMSLS_RETURN_USER, float anova_table[ ],
IMSLS_N_MISSING, int *n_missing,
IMSLS_CV, float *cv,
IMSLS_GRAND_MEAN, float *grand_mean,
IMSLS_FACTOR_MEANS, float **factor_means,
IMSLS_FACTOR_MEANS_USER, float factor_means[ ],
IMSLS_FACTOR_STD_ERRORS, float **factor_std_err,
IMSLS_FACTOR_STD_ERRORS_USER, float factor_std_err[ ],
IMSLS_TWO_WAY_MEANS, float **two_way_means,
IMSLS_TWO_WAY_MEANS_USER, float two_way_means[ ],
IMSLS_TWO_WAY_STD_ERRORS, float **two_way_std_err,
IMSLS_TWO_WAY_STD_ERRORS_USER, float two_way_std_err[ ],
IMSLS_TREATMENT_MEANS, float **treatment_means,
IMSLS_TREATMENT_MEANS_USER, float treatment_means[ ],
IMSLS_TREATMENT_STD_ERROR, float **treatment_std_err,
IMSLS_TREATMENT_STD_ERROR_USER, float treatment_std_err[ ],
IMSLS_ANOVA_ROW_LABELS, char ***anova_row_labels,
IMSLS_ANOVA_ROW_LABELS_USER, char *anova_row_labels[ ],
0)

```

Optional Arguments

`IMSLS_RETURN_USER, float anova_table[]` (Output)

User defined `n_anova` by 6 array for the `anova_table`.

`IMSLS_N_MISSING, int *n_missing` (Output)

Number of missing values, if any, found in `y`. Missing values are denoted with a NaN (Not a Number) value.

`IMSLS_CV, float *cv` (Output)

Coefficient of Variation computed by:

$$CV = \frac{100 \cdot \sqrt{MS_{residual}}}{grand_mean}$$

IMSLI_GRAND_MEAN, *float* *grand_mean (Output)

Mean of all the data across every location.

IMSLI_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.

IMSLI_FACTOR_MEANS_USER, *float* factor_means [] (Output)

Storage for the array `factor_means`, provided by the user.

IMSLI_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.

IMSLI_FACTOR_STD_ERRORS_USER, *float* factor_std_err [] (Output)

Storage for the array `factor_std_err`, provided by the user.

IMSLI_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:

$$\sum_{i=0}^f \sum_{j=i+1}^{f+1} n_levels[i] \times n_levels[j], \text{ where } f = n_factors - 2$$

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: `A1B1`, `A1B2`, `A2B1`, `A2B2`, `A1C1`, `A1C2`, `A2C1`, `A2C2`, `B1C1`, `B1C2`, `B2C1`, and `B2C2`.

IMSLI_TWO_WAY_MEANS_USER, *float* two_way_means [] (Output)

Storage for the array `two_way_means`, provided by the user.

IMSLI_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

$$two_way = \binom{n_factor}{2}$$

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

$$n_levels[0] \times n_levels[1] \times \dots \times n_levels[n_factors - 1]$$

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: $A_1B_1C_1$, $A_1B_1C_2$, $A_1B_2C_1$, $A_1B_2C_2$, $A_2B_1C_1$, $A_2B_1C_2$, $A_2B_2C_1$, 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 `imsls_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. Missing observations for each treatment 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

$$F = \frac{MS_{effect}}{MS_{residual}}$$

when `n_locations = 1`.

If `n_locations > 1` then the error mean squares for all factor F-tests is the pooled location interaction. For example, if `n_factors = 2` then the error sum of squares, degrees of freedom and mean squares are calculated by:

$$\begin{aligned} SS_{error} &= SS_{A \times Locations} + SS_{B \times Locations} + SS_{A \times B \times Locations} \\ df_{error} &= df_{A \times Locations} + df_{B \times Locations} + df_{A \times B \times Locations} \\ MS_{error} &= \frac{SS_{error}}{df_{error}} \end{aligned}$$

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

int main(){
    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 ***

```

	ID	DF	SSQ	squares	F-Test	p-Value
						Mean
[1]	-1	2	13.061	6.530	7.844	0.245
[2]	-1	1	0.107	0.107	0.129	0.781
[3]	-1	1	1.302	1.302	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
Error	-4	1	1.665	1.665
Total	-5	10	25.715

```

Number of Missing Values Estimated: 1
Grand Mean: 3.962
Coefficient of Variation: 32.574

```

```

Factor Means
Factor 1: 2.580637 4.201973 5.101940
          std. err.(df): 0.912459(1)
Factor 2: 3.866924 4.056109
          std. err.(df): 0.745020(1)
Factor 3: 4.290849 3.632185
          std. err.(df): 0.745020(1)

```

```

Two-Way Means
Factor 1 by Factor 2:
3.277605 1.883670
3.744472 4.659474

```

```
4.578696 5.625184
std. err.(df): = 1.290412(1)
```

```
Factor 1 by Factor 3:
3.489899 1.671376
3.605455 4.798491
5.777192 4.426688
std. err.(df): = 1.290412(1)
```

```
Factor 2 by Factor 3:
3.980268 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: 5.037668
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) 1.824918(1)
```

rcbd_factorial

Analyzes data from balanced and unbalanced randomized complete-block experiments. Unlike [imsls_f_anova_factorial](#), function `rcbd_factorial` allows for missing data 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., $n_levels[i] \geq 2$ for $i=0, 1, \dots, n_factors$.

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:

$$n_anova = a + \sum_{i=1}^m \binom{n_factors}{i}$$

$$a = \begin{cases} 3 & \text{if } n_locations = 1 \\ 5 & \text{if } n_locations > 1 \end{cases}$$

and $m = n_factors$.

Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, `anova_tablei,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:

	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

The values for the mean squares, *F*-statistic and *p*-value are set to NaN for the residual and corrected total effects.

Note that the *p*-value for the *F*-statistic is returned as 0.0 when the value is so small that all significant digits have been lost.

The Source Identifiers in the first column of `anova_tableij` 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

Note: 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

$$\binom{n_factors}{n_way}$$

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:

$$CV = \frac{100 \cdot \sqrt{MS_{residual}}}{grand_mean}$$

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

$$\sum_{i=0}^f \sum_{j=i+1}^{f+1} n_levels[i] \times n_levels[j]$$

where

$$f = n_factors - 2$$

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:

`A1B1, A1B2, A2B1, A2B2, A1C1, A1C2, A2C1, A2C2, B1C1, B1C2, B2C1, and B2C2.`

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

$$n_two_way = \binom{n_factors}{2}$$

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

$$n_levels[0] \times n_levels[1] \times \dots \times n_levels[n_factors - 1]$$

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: $A_1B_1C_1$, $A_1B_1C_2$, $A_1B_2C_1$, $A_1B_2C_2$, $A_2B_1C_1$, $A_2B_1C_2$, $A_2B_2C_1$, 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

$$\begin{aligned} SS_{residual} &= SS_{A \times Blocks} + SS_{B \times Blocks} + SS_{A \times B \times Blocks} \\ df_{residual} &= df_{A \times Blocks} + df_{B \times Blocks} + df_{A \times B \times Blocks} \\ MS_{residual} &= \frac{SS_{residual}}{df_{residual}} \end{aligned}$$

When `n_locations = 1`, then $MS_{residual}$ is also used to calculate the standard errors between means. For example, in a two factor experiment:

$$\begin{aligned} \text{StdErr}(A) &= \sqrt{\frac{2 \cdot MS_{residual}}{N_A}} \\ \text{StdErr}(B) &= \sqrt{\frac{2 \cdot MS_{residual}}{N_B}} \\ \text{StdErr}(A \times B) &= \sqrt{\frac{2 \cdot MS_{residual}}{N_{A \times B}}} \end{aligned}$$

where

$$N_A, N_B \text{ and } N_{A \times B}$$

are the number of observations for each level of the effects A, B and their interaction, respectively.

If `n_locations > 1`, then the error mean square is used as the denominator of the F-test for effects:

$$F_{effect} = \frac{MS_{effect}}{MS_{error}}$$

The error mean square in this calculation is obtained by pooling all interactions between each factor and locations. For example `n_locations > 1` and `n_factors=2` then:

$$\begin{aligned} SS_{error} &= SS_{A \times Locations} + SS_{B \times Locations} + SS_{A \times B \times Locations} \\ df_{error} &= df_{A \times Locations} + df_{B \times Locations} + df_{A \times B \times Locations} \\ MS_{error} &= \frac{SS_{error}}{df_{error}} \end{aligned}$$

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 Agrosiences. 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 <imsls.h>

int 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 ***",
    n_anova_table, 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:

$$n = n_locations * n_treatments^2$$

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, `anova_tablei,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	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

Note that the *p*-value for the *F*-statistic is returned as 0.0 when the value is so small that all significant digits have been lost.

The Source Identifiers in the first column of `anova_tablei,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

Source Identifier	ANOVA Source
-3	COLUMNS
-4	TREATMENTS
-5	LOCATIONS × TREATMENTS †
-6	ERROR WITHIN LOCATIONS
-7	CORRECTED TOTAL

Note: † If `n_locations=1` rows involving location are set to missing (NaN).

Synopsis with Optional Arguments

```
#include <ims1.h>

float *ims1s_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 function. 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)*42 + (j-1)*6 + (k-1)]` contains the value in the k -th column and j -th row of the anova-table for the i -th location.

IMSLS_LOCATION_ANOVA_TABLE_USER, *float anova_table[]* (Output) Storage for the array `location_anova_table`, provided by the user.

`IMSLS_ANOVA_ROW_LABELS`, *char ***anova_row_labels* (Output)

Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the *i*-th row of the ANOVA table can be printed with `printf("%s", anova_row_labels[i])`.

The memory associated with `anova_row_labels` can be freed with a single call to `imsls_free(anova_row_labels)`.

`IMSLS_ANOVA_ROW_LABELS_USER`, *char *anova_row_labels[]* (Output)

Storage for the array `anova_row_labels`, provided by the user. The amount of space required will vary depending upon the number of factors and `n_anova`. An upperbound on the required memory is `char *anova_row_labels[600]`.

Description

The function `imsls_f_latin_square` analyzes latin-square experiments, possibly replicated at multiple locations. Latin-square experiments block treatments using two factors: rows and columns. The number of levels associated with rows and columns must equal the number of treatments. Treatments are blocked by rows and columns in a balanced arrangement to ensure that every row contain one replicate of every treatment. The same balance is required for every column, see [Table 16](#). Notice that the four treatments, T1, T2, T3, and T4, appear exactly once in every column and every row.

Table 16 – Latin-Square Experiment with Four Treatments

		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

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 17](#).

Table 17 – The ANOVA Table for a Latin-Square Experiment at one Location

Source	DF	Sum of Squares	Mean Squares
ROWS	$t - 1$	$SSR = t \sum_{i=1}^t (\bar{y}_{i.} - \bar{y}_{..})^2$	MSR
COLUMNS	$t - 1$	$SSC = t \sum_{j=1}^t (\bar{y}_{.j} - \bar{y}_{..})^2$	MSC
TREATMENTS	$t - 1$	$SST = t \sum_{k=1}^t (\bar{y}_{.k} - \bar{y}_{..})^2$	MST
ERROR	$(t - 1)(t - 2)$	$SSE = SST_{tot} - SSR - SSC - SST$	MSE
TOTAL	$t^2 - 1$	$SST_{tot} = \sum_{i=1}^t \sum_{j=1}^t (y_{ij} - \bar{y}_{..})^2$	

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 18](#). 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

$$\tau_k(i,j)$$

is the effect associated with the k th treatment, and

$$\alpha\tau_{lk}(i,j)$$

is the interaction effect between location l and treatment k .

Table 18 – The ANOVA Table for a Latin-Square Experiment at Multiple Locations

SOURCE	DF	Sum of Squares	Mean Squares
LOCATIONS	$r - 1$	$SSL = t^2 \sum_{l=1}^r (\bar{y}_{l..} - \bar{y}_{...})^2$	MSL
ROWS	$r(t - 1)$	$SSR = t \sum_{l=1}^r \sum_{i=1}^t (\bar{y}_{li.} - \bar{y}_{l..})^2$	MSR
COLUMNS	$r(t - 1)$	$SSC = t \sum_{l=1}^r \sum_{j=1}^t (\bar{y}_{l.j} - \bar{y}_{l..})^2$	MSC
TREATMENTS	$t - 1$	$SST = r \cdot t \sum_{k=1}^t (\bar{y}_{.k} - \bar{y}_{...})^2$	MST
LOCATIONS X TREATMENTS	$(r - 1)(t - 1)$	SSLT by difference	MSLT
ERROR	$(t - 1)[r((t - 1) - 1 - 1)]$	$SSE = \sum_{l=1}^r SSE_l$	MSE
TOTAL	$r \cdot t^2 - 1$	$SSTot = \sum_{l=1}^r \sum_{i=1}^t \sum_{j=1}^t (y_{lij} - \bar{y}_{...})^2$	

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

void l_print_LSD(int n1, int* equalMeans, float *means);

int main()
{
    char **anova_row_labels;
    char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ", "\nMean", "\nsquares", "\nF-Test", "\np-Value"};
}
```

```

int i, l, page_width=100;
int n      = 16; /* Total number of observations */
int n_treatments = 4;
int n_locations = 1;
int df, *equal_means;
float grand_mean, cv;
float *aov, *treatment_means, *std_err;
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 alpha = 0.05;
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
};

printf("\n\n*** Experimental Design ***");
printf("\n\n=====");
printf("\n| COL | 1 | 2 | 3 | 4 |");
printf("\n\n=====");
printf("\n|ROW 1 | 2 | 4 | 3 | 1 |");
printf("\n\n=====");
printf("\n|ROW 2 | 3 | 1 | 2 | 4 |");
printf("\n\n=====");
printf("\n|ROW 3 | 1 | 3 | 4 | 2 |");
printf("\n\n=====");
printf("\n|ROW 4 | 4 | 2 | 1 | 3 |");
printf("\n\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);

/* Print ANOVA table. */
imsls_page(IMSLs_SET_PAGE_WIDTH, &page_width);
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\n");
printf("Standard Error for Comparing Two Treatment Means: %f \n",
    std_err[0]);
printf("(df=%d)\n", 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 ***

	ID	DF	SSQ	Mean squares	F-Test	p-Value
Locations	-1
Rows	-2	3	0.185	0.062	2.064	0.207
Columns	-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: 0.122200 (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_lattice` verifies that:

$$n = n_locations \times t \times r \text{ where}$$

$$t = n_treatments \text{ and } r = n_reps$$

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

j	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
4	F -statistic
5	p -value for this F -statistic

Note that the p -value for the F -statistic is returned as 0.0 when the value is so small that all significant digits have been lost.

The Source Identifiers in the first column of `anova_tableij` 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

Note: † If `n_locations=1`, all entries in this row are set to missing (NaN).

Synopsis with Optional Arguments

```
#include <ims1.h>

float *ims1s_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_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[ ],
```

```

IMSLD_STD_ERRORS, float **std_err,
IMSLD_STD_ERRORS_USER, float std_err[ ],
IMSLD_LOCATION_ANOVA_TABLE, float **location_anova_table,
IMSLD_LOCATION_ANOVA_TABLE_USER, float location_anova_table[ ],
IMSLD_ANOVA_ROW_LABELS, char ***anova_row_labels,
IMSLD_ANOVA_ROW_LABELS_USER, char *anova_row_labels[ ],
IMSLD_RETURN_USER, float anova_table[ ],
0)

```

Optional Arguments

IMSLD_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.

IMSLD_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.

IMSLD_CV, *float* ***cv** (Output)

The coefficient of variation computed by using the location standard deviation.

IMSLD_GRAND_MEAN, *float* ***grand_mean** (Output)

The overall adjusted mean averaged over every location.

IMSLD_TREATMENT_MEANS, *float* ****treatment_means** (Output)

Address of a pointer to an internally allocated array of size **n_treatments** containing the adjusted treatment means.

IMSLD_TREATMENT_MEANS_USER, *float* **treatment_means**[] (Output)

Storage for the array **treatment_means**, provided by the user.

IMSLD_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 k -th column and j -th row of the anova-table for the i -th location.

`IMSLs_LOCATION_ANOVA_TABLE_USER`, *float* `anova_table[]` (Output)

Storage for the array `location_anova_table`, provided by the user.

`IMSLs_ANOVA_ROW_LABELS`, *char **** `anova_row_labels` (Output)

Address of a pointer to a pointer to an internally allocated array containing the labels for each of the `n_anova` rows of the returned ANOVA table. The label for the i -th row of the ANOVA table can be printed with `printf("%s", anova_row_labels[i]);` The memory associated with `anova_row_labels` can be freed with a single call to `imsls_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];`

`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`.

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

$$n_blocks = n_locations \cdot r \cdot (k - 1) + 1$$

For a balanced-lattice,

$$n_blocks = b = r \cdot k = (k + 1) \cdot k = (\sqrt{t} + 1) \cdot \sqrt{t} = 4 \cdot 3 = 12$$

Table 19 – A 3x3 Balanced-Lattice for 9 Treatments in Four Replicates

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)

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 [Table 20](#) and [Table 21](#).

Table 20 – The ANOVA Table for a Lattice Experiment at one Location

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 21 – The ANOVA Table for a Lattice Experiment at Multiple Locations

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	

Examples

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 $r k=5(4)=20$ blocks.

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

```

#include <stdlib.h>
#include <math.h>
void l_print_LSD(int n1, int* equalMeans, float *means);

int 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,
        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,
        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
    };
    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 ");
printf("Means: %f \n(df=%d)\n", std_err[2], df);
equal_means = imsls_f_multiple_comparisons(n_treatments,
    treatment_means, df, std_err[2]/(float)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 ***						
	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	4.11e-005
Treatments (adjusted)	-4	15	21271.29	1418.09	4.21	8.99e-005
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 <math.h>
#include <imsls.h>
#include <stdio.h>
#include <stdlib.h>

void l_print_LSD(int n1, int* equalMeans, float *means);

int 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,
        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, 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%9.3g",
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\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%9.3g",
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 ");
printf("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);
}
/*
* 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 ***						
	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	0.00138
Treatments (unadjusted) ...	-3	24	795.46	33.14	0.02	1
Treatments (adjusted)	-4	24	951.20	39.63	2.89	0.00591
Blocks (adjusted)	-5	16	770.50	48.16	3.51	0.000256
Intra-Block Error	-6	55	753.81	13.71
Corrected Total	-7	98	2535.95

LOCATION 1						
*** ANALYSIS OF VARIANCE TABLE ***						
	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 ***						
	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	0.0178
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:

$$n = \sum_{i=1}^{n_locations} (n_whole * n_split * n_blocks[i-1])$$

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_tablei,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:

<code>j</code>	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

The Source Identifiers in the first column of `anova_tableij` are the only negative values in `anova_table[]`. Note that the p -value for the F -statistic is returned as 0.0 when the value is so small that all significant digits have been lost. 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 on table:

† 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_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_WHOLESPLIT_PLOT_SS, *float* **wholesplit_plot_ss,
IMSLS_WHOLESPLIT_PLOT_SS_USER, *float* wholesplit_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[ ],
IMSLS_RETURN_USER, float anova_table[ ],
0)

```

Optional Arguments

`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` (Input)

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`, (Input)

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`, (Input)

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`, (Input)

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.

IMSL_SPLIT_PLOT_MEANS_USER, *float* split_plot_means[] (Output)
 Storage for the array `split_plot_means`, provided by the user.

IMSL_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_meansij = 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.

IMSL_TREATMENT_MEANS_USER, *float* treatment_means[] (Output)
 Storage for the array `treatment_means`, provided by the user.

IMSL_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.

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]

IMSL_STD_ERRORS_USER, *float* std_err[] (Output)
 Storage for the array `std_err`, provided by the user.

IMSL_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.

IMSL_N_BLOCKS_USER, *int* n_blocks[] (Output)
 Storage for the array `n_blocks`, provided by the user.

IMSL_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.

`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 error 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 ***`wholxsplit_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* `wholxsplit_plot_ss[]` (Output)

Storage for the array `wholxsplit_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 error 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 `imsls_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]`.

`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`.

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 function 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 [Table 22](#). 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 [Table 23](#).

Table 22 – Split-Plot Experiments – Split-Plot B Nested within Whole-Plot A

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

Table 23 – Completely Randomized Experiments – Both Factors Randomized

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

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 function can analyze $3 \times 2 \times 2 \times 2 = 24$ 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_RCB`.
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 two whole-plots and four split-plots.

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

int 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 ***

```

	ID	DF	SSQ	squares	Mean	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 Split-Plot	-9
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:

$$n = \sum_{i=1}^{n_locations} (n_whole \times n_split \times n_sub \times n_blocks[i - 1])$$

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 `IMSL_LOCATIONS`.

int `n_whole` (Input)

Number of levels associated with the whole-plot factor. `n_whole` must be greater than one.

int `n_split` (Input)

Number of levels associated with the split-plot factor. `n_split` must be greater than one.

int `n_sub` (Input)

Number of levels associated with the sub-plot factor. `n_sub` must be greater than one.

int `rep`[] (Input)

An array of length `n` containing the block, or replicate, identifiers for each observation in `y`. Different locations can have different numbers of blocks or replicates. Each block or replicate at a single location must be assigned a different identifier, but different locations can have the same assignments.

int `whole`[] (Input)

An array of length `n` containing the whole-plot identifiers for each observation in `y`. Each level of the whole-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique whole-plot identifiers is equal to `n_whole`.

int `split`[] (Input)

An array of length `n` containing the split-plot identifiers for each observation in `y`. Each level of the split-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique split-plot identifiers is equal to `n_split`.

int `sub`[] (Input)

An array of length `n` containing the sub-plot identifiers for each observation in `y`. Each level of the sub-plot factor must be assigned a different integer. `imsls_f_split_split_plot` verifies that the number of unique sub-plot identifiers is equal to `n_sub`.

float `y`[] (Input)

An array of length `n` containing the experimental observations and any missing values. Missing values cannot be omitted. They are included by placing a NaN (not a number) in `y`. The NaN value can be set using either the function `imsls_f_machine(6)` or `imsls_d_machine(6)`, depending upon whether single or double precision is being used, respectively. At a single location, only one missing value per whole-plot is allowed. The location, whole-plot, split-plot and sub-plot for each observation in `y` are identified by the corresponding values in the arguments `locations`, `whole`, `split` and `sub`.

Return Value

Address of a pointer to the memory location of a two dimensional, 20 by 6 array containing the ANOVA table. Each row in this array contains values for one of the effects in the ANOVA table. The first value in each row, `anova_tablei,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:

<code>j</code>	<code>anova_table_{i,j} = anova_table[<i>i</i>*6+<i>j</i>]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares

<code>j</code>	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
3	Mean squares
4	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

The Source Identifiers in the first column of `anova_tableij` are the only negative values in `anova_table[]`. Note that the *p*-value for the *F*-statistic is returned as 0.0 when the value is so small that all significant digits have been lost. Assignments of identifiers to ANOVA sources use the following coding:

-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` (Input)

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 $k > 0$,
 $treatment_means_{i,j,k} = treatment_means[(i-1)*n_split*n_sub + (j-1)*n_sub + k-1]$ 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]**.

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 `imsls_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.

Table 24 – Split-Plot Experiment – Split-Plot B Nested within Whole-Plot A

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

Table 25 – Split-Split Plot Experiment – Sub-Plot Factor C Nested Within Split-Plot Factor B, Nested Within Whole-Plot Factor A

A2B3C2	A1B2C1	A4B1C2	A3B3C2
A2B3C1	A1B2C2	A4B1C1	A3B3C1
A2B1C1	A1B1C1	A4B3C2	A3B2C2
A2B1C2	A1B1C2	A4B3C1	A3B2C1
A2B2C2	A1B3C1	A4B2C1	A3B1C2
A2B2C1	A1B3C2	A4B2C2	A3B1C1

Contrast the split-split plot experiment to the same experiment run using a strip-split plot design, see [Table 26](#). 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.

Table 26 – Strip-Split Plot Experiment - Split-Plots Nested Within Strip-Plot Factors A and B

Factor B Strip Plots	B3	A2B3C2	A1B3C1	A4B3C2	A3B3C2
		A2B3C1	A1B3C2	A4B3C1	A3B3C1
	B1	A2B1C1	A1B1C1	A4B1C2	A3B1C2
		A2B1C2	A1B1C2	A4B1C1	A3B1C1
	B2	A2B2C2	A1B2C1	A4B2C1	A3B2C2
		A2B2C1	A1B2C2	A4B2C2	A3B2C1

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 $3 \times 2 = 6$ 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 two whole-plots, two-split-plots and two sub-plots.

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

```

int main()
{
    char **anova_row_labels;
    char *col_labels[] = {" ", "\nID", "\nDF", "\nSSQ", "Mean\nsquares",
        "\nF", "\np-value"};
    char dashes[] =
        "*****";
    int i, j, k, l, page_width = 132;
    int n = 24;
    int n_locations = 1;
    int n_whole = 2;
    int n_split = 2;
    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, *cv, *aov, *treatment_means;
    float *whole_plot_means, *split_plot_means;
    float *sub_plot_means, *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%s\n", dashes);
printf("Treatment 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 ",
    std_err[3]);
printf("\n(df=%f)\n", std_err[7]);

equal_means = imsls_f_multiple_comparisons(n_whole * n_split * n_sub,
    treatment_means, (int)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%s", dashes);

imsls_f_write_matrix("Whole-plot Means", n_whole, 1,
    whole_plot_means,
    0);

printf("\nStandard Error for Comparing Two Whole-Plot Means: %f ",
    std_err[0]);

```

```
printf("\n(df=%f)\n", std_err[4]);

equal_means = imsls_f_multiple_comparisons(n_whole,
    whole_plot_means, (int)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%s", dashes);

imsls_f_write_matrix("Split-plot Means", n_split, 1,
    split_plot_means,
    0);

printf("\nStandard Error for Comparing Two Split-Plot Means: %f ",
    std_err[1]);
printf("\n(df=%f)\n", std_err[5]);

equal_means = imsls_f_multiple_comparisons(n_split,
    split_plot_means, (int)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%s", dashes);

imsls_f_write_matrix("Sub-plot Means", n_sub, 1, sub_plot_means,
    0);

printf("\nStandard Error for Comparing Two Sub-Plot Means: %f ",
    std_err[2]);
printf("\n(df=%f)\n", std_err[6]);

equal_means = imsls_f_multiple_comparisons(n_sub, sub_plot_means,
    (int)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 ***

```

	ID	DF	SSQ	squares	Mean	F	p-value
Location	-1
Blocks 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 Split-Plot	-9
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 x Sub-Plot	-17	1	0.51	0.51	0.16	0.703	
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:

$$n = \sum_{i=1}^{n_locations} (n_strip_a \cdot n_strip_b \cdot n_blocks[i - 1])$$

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 `IMSL_S_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 function 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 function 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_tablei,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:

<code>j</code>	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares
4	<i>F</i> -statistic
5	<i>p</i> -value for this <i>F</i> -statistic

The Source Identifiers in the first column of `anova_tablei,j` are the only negative values in `anova_table`. Note that the p -value for the F -statistic is returned as 0.0 when the value is so small that all significant digits have been lost. Assignments of identifiers to ANOVA sources use the following coding:

-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,
```

```

IMSL_S_STRIP_PLOT_A_MEANS_USER, float strip_plot_a_means[ ],
IMSL_S_STRIP_PLOT_B_MEANS, float **strip_plot_b_means,
IMSL_S_STRIP_PLOT_B_MEANS_USER, float strip_plot_b_means[ ],
IMSL_S_TREATMENT_MEANS, float **treatment_means,
IMSL_S_TREATMENT_MEANS_USER, float treatment_means[ ],
IMSL_S_STD_ERRORS, float **std_err,
IMSL_S_STD_ERRORS_USER, float std_err[ ],
IMSL_S_N_BLOCKS, int **n_blocks,
IMSL_S_N_BLOCKS_USER, int n_blocks[ ],
IMSL_S_LOCATION_ANOVA_TABLE, float **location_anova_table,
IMSL_S_LOCATION_ANOVA_TABLE_USER, float location_anova_table[ ],
IMSL_S_ANOVA_ROW_LABELS, char ***anova_row_labels,
IMSL_S_ANOVA_ROW_LABELS_USER, char *anova_row_labels[ ],
0)

```

Optional Arguments

`IMSL_S_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`.

`IMSL_S_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`.

`IMSL_S_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.

`IMSL_S_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.

IMSL5_CV_USER, *float cv*[] (Output)

Storage for the array *cv*, provided by the user.

IMSL5_GRAND_MEAN, *float *grand_mean* (Output)

Mean of all the data across every location.

IMSL5_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.

IMSL5_STRIP_PLOT_A_MEANS_USER, *float strip_plot_a_means*[] (Output)

Storage for the array *strip_plot_a_means*, provided by the user.

IMSL5_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.

IMSL5_STRIP_PLOT_B_MEANS_USER, *float strip_plot_b_means*[] (Output)

Storage for the array *strip_plot_b_means*, provided by the user.

IMSL5_TREATMENT_MEANS, *float **treatment_means* (Output)

Address of a pointer to an internally allocated array of length (*n_strip_a* × *n_strip_b*) containing the treatment means. For *i* > 0 and *j* > 0,

treatment_means_{ij} = *treatment_means* [(*i*-1) × *n_strip_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.

IMSL5_TREATMENT_MEANS_USER, *float treatment_means*[] (Output)

Storage for the array *treatment_means*, provided by the user.

IMSL5_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].

<i>std_err</i> [0]	Factor A Strip-Plot Means	<i>std_err</i> [5]
<i>std_err</i> [1]	Factor B Strip-Plot Means	<i>std_err</i> [6]
<i>std_err</i> [2]	Factor A Strip-Plot Means at the same level of Factor B	<i>std_err</i> [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 `imsls_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 28](#) below. In strip-plot experiments, Factors A and B are completely crossed, see [Table 27](#) 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.

Table 27 – Strip-Plot Experiments – Strip-Plots Completely Crossed

Strip	B3	A2B3	A1B3	A4B3	A3B3
Plot	B1	A2B1	A1B1	A4B1	A3B1
Factor B	B2	A2B2	A1B2	A4B2	A3B2

Table 28 – Split-Plot Experiments – Split-Plot B Nested within Strip-Plot A

Whole Factor Plot			
A2	A1	A4	A3
A2B1	A1B3	A4B1	A3B2
A2B3	A1B1	A4B3	A3B1
A2B2	A1B2	A4B2	A3B2

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

int 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;
    int n_locations = 1;
    int n_strip_a = 2;
    int n_strip_b = 4;

    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, *aov, *treatment_means;
    float *strip_plot_a_means, *strip_plot_b_means;
```

```

float *std_err;
int n_missing, *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("\n\nStandard Error for Comparing Two Factor A Means: \n");
printf("%f (df=%f)\n", std_err[0], std_err[5]);
equal_means = imsls_f_multiple_comparisons(n_strip_a,
    strip_plot_a_means, (int) 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("\n\nStandard Error for Comparing Two Factor B Means: \n");
printf("%f (df=%f)\n", std_err[1], std_err[6]);

```

```

equal_means = imsls_f_multiple_comparisons(n_strip_b,
strip_plot_b_means, (int)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 ***						
	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 x Strip-Plot B	-10
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:

$$n = \sum_{i=1}^{n_locations} (n_strip_a \times n_strip_b \times n_split \times n_blocks[i - 1])$$

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 \mathbf{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 \mathbf{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 \mathbf{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 \mathbf{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 \mathbf{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 \mathbf{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_tablei,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:

<code>j</code>	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
0	Source Identifier (values described below)
1	Degrees of freedom
2	Sum of squares
3	Mean squares

j	<code>anova_table_{i,j} = anova_table[i*6+j]</code>
4	F-statistic
5	p-value for this F-statistic

The Source Identifiers in the first column of `anova_tableij` are the only negative values in `anova_table[]`. Note that the *p*-value for the *F*-statistic is returned as 0.0 when the value is so small that all significant digits have been lost. Assignments of identifiers to ANOVA sources use the following coding:

-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 Arguments

```
#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[ ],
```

```

IMSLD_STD_ERRORS, float **std_err,
IMSLD_STD_ERRORS_USER, float std_err[ ],
IMSLD_N_BLOCKS, int **n_blocks,
IMSLD_N_BLOCKS_USER, int n_blocks[ ],
IMSLD_LOCATION_ANOVA_TABLE, float **location_anova_table,
IMSLD_LOCATION_ANOVA_TABLE_USER, float location_anova_table[ ],
IMSLD_ANOVA_ROW_LABELS, char ***anova_row_labels,
IMSLD_ANOVA_ROW_LABELS_USER, char *anova_row_labels[ ],
0)

```

Optional Arguments

`IMSLD_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`.

`IMSLD_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`.

`IMSLD_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.

`IMSLD_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.

`IMSLD_CV_USER, float cv[]` (Output)

Storage for the array `cv`, provided by the user.

`IMSLD_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 strip-plot B 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_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 **strip_plot_b_split_plot_means* (Output)
Address of a pointer to an internally allocated 2-dimensional array of size `n_strip_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 $i > 0, j > 0,$ and $k > 0,$
 $treatment_means_{i,j,k} = treatment_means((i-1) * n_strip_b * n_split + (j-1) * n_split + (k-1))$ contains the mean of the observations, averaged over all locations, blocks and replicates, for the i -th level of the factor A strip-plot, the j -th level of the factor B strip-plot, and the k -th level of the split-plot.

IMSLS_TREATMENT_MEANS_USER, *float treatment_means[]* (Output)

Storage for the array *treatment_means*, provided by the user.

IMSLS_STD_ERRORS, *float **std_err* (Output)

Address of a pointer to an internally allocated array of length 20 containing ten standard errors and their associated degrees of freedom. The standard errors are in the first 10 elements and their associated degrees of freedom are reported in *std_err[10]* through *std_err[19]*.

<i>std_err[0]</i>	Strip-Plot A Means	<i>std_err[10]</i>
<i>std_err[1]</i>	Strip-Plot B Means	<i>std_err[11]</i>
<i>std_err[2]</i>	Split-Plot Means	<i>std_err[12]</i>
<i>std_err[3]</i>	Strip-Plot A Means at the same level of split-plots	<i>std_err[13]</i>
<i>std_err[4]</i>	Strip-Plot A Means at the same level of strip-plot B	<i>std_err[14]</i>
<i>std_err[5]</i>	Strip-Plot B Means at the same level of split-plots	<i>std_err[15]</i>
<i>std_err[6]</i>	Strip-Plot B Means at the same level of strip-plot A	<i>std_err[16]</i>
<i>std_err[7]</i>	Split-Plot Means at the same level of strip-plot A	<i>std_err[17]</i>
<i>std_err[8]</i>	Split-Plot Means at the same level of strip-plot B	<i>std_err[18]</i>
<i>std_err[9]</i>	Treatment Means (same strip-plot A, strip-plot B and split-plot)	<i>std_err[19]</i>

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 `imsls_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.

Table 29 – Strip-Plot Experiment - Strip-Plots Completely Crossed

Factor B Strip Plots	B3	A2B3	A1B3	A4B3	A3B3
	B1	A2B1	A1B1	A4B1	A3B1
	B2	A2B2	A1B2	A4B2	A3B2

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 30](#). 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.

Table 30 – Split-Plot Experiment – Factor B Split-Plots Nested within Factor A Whole-Plots

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

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 31. The third factor, referred to as the split-plot factor, is randomly assigned to experimental units within each level of strip-plot factor A. `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.

Table 31 – Strip-Split Plot Experiment - Split-Plots Nested Within Strip-Plot Factors A

Factor B Strip Plots	B3	A2B3C2	A1B3C1	A4B3C2	A3B3C2
		A2B3C1	A1B3C2	A4B3C1	A3B3C1
	B1	A2B1C1	A1B1C1	A4B1C2	A3B1C2
		A2B1C2	A1B1C2	A4B1C1	A3B1C1
	B2	A2B2C2	A1B2C1	A4B2C1	A3B2C2
		A2B2C1	A1B2C2	A4B2C2	A3B2C1

Strip-split plot experiments are closely related to split-split plot experiments, see Table 32. 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.

Table 32 – Split-Split Plot Experiment – Sub-Plot Factor C Nested Within Split-Plot Factor B

A2B3C2	A1B2C1	A4B1C2	A3B3C2
A2B3C1	A1B2C2	A4B1C1	A3B3C1
A2B1C1	A1B1C1	A4B3C2	A3B2C2
A2B1C2	A1B1C2	A4B3C1	A3B2C1
A2B2C2	A1B3C1	A4B2C1	A3B1C2
A2B2C1	A1B3C2	A4B2C2	A3B1C1

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. 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.

Table 33 – Strip-Split Plot Experiment – Fertilizer Strip-Plots, Application Rate Strip-Plots

		F2	F1	F4	F3
Application Rate Strip Plot	R3	F2R3S1	F1R3S3	F4R3S3	F3R3S2
		F2R3S2	F1R3S2	F4R3S2	F3R3S1
		F2R3S3	F1R3S1	F4R3S1	F3R3S3
	R2	F2R1S3	F1R1S2	F4R1S3	F3R1S1
		F2R1S1	F1R1S3	F4R1S1	F3R1S2
		F2R1S2	F1R1S1	F4R1S2	F3R1S3
	R1	F2R2S1	F1R2S1	F4R2S2	F3R2S3
		F2R2S2	F1R2S3	F4R2S3	F3R2S1
		F2R2S3	F1R2S2	F4R2S1	F3R2S2

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.

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 $n = 2 \times 2 \times 2 \times 3 = 24$ 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);

int 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("*****");
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: ");
printf("%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, (int) 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*****");
printf("*****");
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: ");
printf("%f \n(df=%f)\n",
    std_err[0], std_err[10]);
equal_means = imsls_f_multiple_comparisons(n_strip_a,
    strip_plot_a_means, (int) 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*****");
printf("*****");
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: ");
printf("%f \n(df=%f)\n", std_err[1], std_err[11]);
equal_means = imsls_f_multiple_comparisons(n_strip_b,
        strip_plot_b_means,
        (int) 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*****");
printf("*****");
imsls_f_write_matrix("Split-plot Means", n_split, 1,
        split_plot_means, 0);
printf("\nStandard Error for Comparing Two Split-Plot Means: ");
printf("%f \n(df=%f)\n", std_err[2], std_err[12]);
equal_means = imsls_f_multiple_comparisons(n_split,
        split_plot_means,
        (int) 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*****");
printf("*****");
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,
        (int) 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*****");
printf("*****");
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, (int) 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*****");
printf("*****");
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,
    (int) 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 ***						
	ID	DF	SSQ	Mean 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 <ims1s.h>
```

```
float *ims1s_f_homogeneity(int n, int n_treatment, int treatment[], float y[], ..., 0)
```

The type *double* is `ims1s_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. `ims1s_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 `ims1s_f_machine(6)` or `ims1s_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 Bartlett's and Levene's tests.

Synopsis with Optional Arguments

```
#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 (Input)

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:

$$H_o : \sigma_1 = \sigma_2 = \dots = \sigma_t$$

versus

$$H_a : \sigma_i \neq \sigma_j$$

for at least one pair ($i \neq j$), where $t = n_treatments$.

Bartlett's test, Bartlett (1937), uses the test statistic:

$$\chi^2 = \frac{M}{C}$$

where

$$M = N \cdot \ln(S_p^2) - \sum n_i \ln(S_i^2), \quad N = \sum_{i=1}^t n_i, \quad S_p^2 = \frac{\sum_{i=1}^t (n_i - 1) S_i^2}{\sum_{i=1}^t (n_i - 1)}$$

$$C = 1 + \frac{1}{3(t-1)} \left[\sum \frac{1}{n_i} - \frac{1}{N} \right]$$

and S_i^2 is the variance of the n_i non-missing observations in the i -th treatment. S_p^2 is referred to as the pooled variance, and it is also known as the error mean squares from a 1-way analysis of variance.

If the usual assumptions associated with the analysis of variance are valid, then Bartlett's test statistic is a chi-squared random variable with degrees of freedom equal to $t-1$.

The original Levene's test, Levene (1960) and Snedecor & Cochran (1967), uses a different test statistic, F_0 , equal to:

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

where

$$z_{ij} = |x_{ij} - \bar{x}_i|$$

x_{ij} is the j -th observation from the i -th treatment and \bar{x}_i is the mean for the i -th treatment. Conover, Johnson, and Johnson (1981) compared over 50 similar tests for homogeneity and concluded that one of the best tests was Levene's test when the treatment mean, \bar{x}_i , is replaced with the treatment median, \tilde{x}_i . This version of Levene's test can be requested by setting `IMSL5_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 j -th observation within the i -th treatment, e_{ij} , returned from `IMSL5_RESIDUALS` is unstandardized, i.e. $e_{ij} = x_{ij} - \bar{x}_i$. For investigating problems of homogeneity of variance, the studentized residuals returned by `IMSL5_STUDENTIZED_RESIDUALS` are recommended since they are standardized 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 `IMSL5_CV`, is also calculated using the pooled variance and the grand mean:

$$\bar{x}_{..} = \sum_i \sum_j x_{ij}$$

$$CV = \frac{100 \cdot \sqrt{S_p^2}}{\bar{x}_{..}}$$

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>
#include <stdio.h>

int main()
{
    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;
    float *residuals;
    float *studentized_residuals;
    float *std_devs;
    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, or Tukey's procedures.

Synopsis

```
#include <imspls.h>
```

```
int *imspls_f_multiple_comparisons (int n_groups, float means[ ], int df, float std_error,
..., 0)
```

The type *double* function is `imspls_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

$$\text{std_error} = \sqrt{\frac{s^2}{n}}$$

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

$$\text{std_error} = \frac{\text{sedif}}{\sqrt{2}}$$

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

Argument	Method
IMSLS_SNK	Student-Newman-Keuls (default)
IMSLS_LSD	Least significant difference
IMSLS_TUKEY	Tukey's <i>w</i> -procedure, also called the honestly significant difference procedure.
IMSLS_BONFERRONI	Bonferroni <i>t</i> 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:

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.

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>

int 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.

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>

int 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
```

false_discovery_rates

Calculates the False Discovery Rate (FDR) q -values corresponding to a set of p -values resulting from multiple simultaneous hypothesis tests.

Synopsis

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

```
float *imsls_f_false_discovery_rates (int n_tests, float pvalues[ ], ..., 0)
```

The type *double* function is `imsls_d_false_discovery_rates`.

Required Arguments

int `n_tests` (Input)

The number of hypothesis tests.

float `pvalues[]` (Input)

An array of length `n_tests` containing the p -values associated with the tests.

Return Value

Pointer to an array of length `n_tests` containing the calculated q -values.

Synopsis with Optional Arguments

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

```
float *imsls_f_false_discovery_rates (int n_tests, float pvalues[ ],
```

```
    IMSLS_LAMBDA, int n_lambdas, float lambdas[ ]
```

```
    IMSLS_GAMMA_PARAM, float gamma,
```

```
    IMSLS_METHOD, int method,
```

```
    IMSLS_SMOOTHING_PAR, float smoothing_par,
```

```
    IMSLS_N_SAMPLE, int n_samples,
```

```

IMSLS_RANDOM_SEED, int random_seed,
IMSLS_CONFIDENCE, float confid,
IMSLS_NULL_PROB, float *pi0,
IMSLS_UPPER_LIMITS, float **upper_limits,
IMSLS_UPPER_LIMITS_USER, float upper_limits[ ],
IMSLS_RETURN_USER, float qvalues[ ],
0)

```

Optional Arguments

`IMSLS_LAMBDA_S`, *int* `n_lambdas`, *float* `lambdas[]` (Input)

An array of length `n_lambdas` containing the grid values on $[0,1)$ used in the estimate of the null probability.

Default: `n_lambdas` = 19, `lambdas` = {0.0,0.05,...,0.90}.

`IMSLS_GAMMA_PARAM`, *float* `gamma` (Input)

Size of the rejection region ($0 \leq \text{gamma} \leq 1.0$) used in the calculation of the *FDR* and *pFDR* measures.

Default: `gamma` = 0.05

`IMSLS_METHOD`, *int* `method` (Input)

Specifies the method used to remove the dependence of the null probability estimate on the `lambda` variable. `method` = 1 or `method` = 0: when `method` = 0, a bootstrap with `n_samples` is used; when `method` = 1, a cubic spline smoother is used.

Default: `method` = 0

`IMSLS_SMOOTHING_PAR`, *float* `smoothing_par` (Input)

Smoothing parameter ($0 \leq \text{smoothing_par} \leq 1.0$) argument for the cubic spline smoother. Only used if `method` = 1.

Default: If `method` = 1 and this optional argument is not provided, the smoothing parameter is selected by cross-validation.

`IMSLS_N_SAMPLE`, *int* `n_samples` (Input)

Number of bootstrap samples to make for `method` = 0 or when estimating upper confidence limits when `IMSLS_CONFIDENCE` is present.

Default: `n_samples` = 100.

`IMSLS_RANDOM_SEED`, *int* `random_seed` (Input)

The seed of the random number generator used in generating the bootstrap samples. If `random_seed` is 0, a value is computed using the system clock; hence, the results may be different between different calls with the same inputs.

Default: `random_seed` = 0.

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

Confidence level ($0.0 < \text{confid} < 100.0$). See `IMSLS_UPPER_LIMITS`.

Default: `confid` = 95.0

`IMSLS_NULL_PROB`, *float* *`pi0` (Output)

The null probability estimate.

`IMSLS_UPPER_LIMITS`, *float* **`upper_limits` (Output)

Address of a pointer to an array of length 2 containing the (`confid`)% upper bounds for *pFDR* and *FDR*.

`IMSLS_UPPER_LIMITS_USER`, *float* `upper_limits[]` (Output)

Storage for array `upper_limits` is supplied by the user. See `IMSLS_UPPER_LIMITS`.

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

Storage for the return value is supplied by the user.

Description

Let $\{p_1, p_2, \dots, p_m\}$ be the *p*-values associated with *m* independent tests of a statistical hypothesis. The following table summarizes the possible outcomes of the *m* tests. Note that the only known quantities in the table are *W*, *R*, and, *m*.

Hypothesis	Accept	Reject	Total
Null is true	<i>U</i>	<i>V</i>	m_0
Alternative is true	<i>T</i>	<i>S</i>	m_1
Total	<i>W</i>	<i>R</i>	<i>m</i>

In the above table, *V* is the number of false discoveries (and the number of type I errors). Whereas the type I error rate is the probability of rejecting at least one true null hypothesis, the *false discovery rate (FDR)* (Benjamini and Hochberg (1995)), is the expected proportion of falsely rejected true nulls. In other words, the *FDR* is the expected proportion of “false positives” among the tests that are deemed significant. Using the notations from the table,

$$FDR = E \left[\frac{V}{R \vee 1} \right] = E \left[\frac{V}{R} \mid R > 0 \right] \Pr [R > 0]$$

The denominator $R \vee 1 = \max(R, 1)$ avoids division by 0 in case there are no significant results ($R = 0$). The *positive false discovery rate*, or *pFDR*, defined in Storey (2001), is conditional on there being at least one significant test ($R > 0$):

$$pFDR = E \left[\frac{V}{R} \mid R > 0 \right]$$

From Theorem 1 in Storey (2001),

$$pFDR = \frac{\pi_0 \Pr [P \leq \gamma \mid H = 0]}{\Pr [P \leq \gamma]}$$

where π_0 is the probability that an individual hypothesis is null, and $\Pr [P \leq \gamma \mid H = 0] = \gamma$ is the probability an individual hypothesis is rejected given that it is null. The parameter γ is the predefined size of the rejection region. The denominator is the probability that a test is rejected, given γ . This relationship arises from Baye's Theorem and the assumption that the p -values are independent.

An estimator for m_0 , the number of true null hypotheses is

$$\hat{m}_0(\lambda) = \frac{m - R(\lambda)}{1 - \lambda} = \frac{W(\lambda)}{1 - \lambda}$$

where λ is a significance level on the interval $[0, 1)$ and $W(\lambda)$ is the number of tests that are accepted at level λ . That is, $W(\lambda) = \#\{ p_i > \lambda \}$. An estimator for the probability of a null hypothesis π_0 is then:

$$\hat{\pi}_0(\lambda) = \frac{\hat{m}_0(\lambda)}{m} = \frac{W(\lambda)}{m(1 - \lambda)}$$

The parameter λ is a tuning parameter used to estimate the true null distribution. The rationale is that the p -values of the null hypotheses are uniformly distributed and most of the larger p -values ($> \lambda$) will be from the null distribution. See Storey and Tibshirani (2003) for further details.

Now using

$$\hat{\Pr}(P \leq \gamma) = \frac{R(\gamma)}{m}$$

Storey (2002) gives the following estimators for *pFDR* and *FDR*:

$$pFDR_{\lambda}(\gamma) = \frac{\hat{\pi}_0(\lambda)\gamma}{\hat{P}(P \leq \gamma)} = \frac{W(\lambda)\gamma}{(1 - \lambda)\{R(\gamma) \vee 1\} \{1 - (1 - \gamma)^m\}}$$

and

$$FDR_{\lambda}(\gamma) = \hat{\pi}_0(\lambda)\gamma = \frac{W(\lambda)\gamma}{(1 - \lambda)\{R(\gamma) \vee 1\}}$$

Note that $1 - (1 - \gamma)^m$ is a lower bound for $P(R > 0)$, the probability of at least one significant test, and

$$\frac{\gamma}{1 - (1 - \gamma)^m}$$

is a conservative estimate for the type I error, given that $R > 0$.

In `imsls_f_false_discovery_rates`, the estimates above are calculated on a grid of λ values on $[0, 1]$ and the minimizer is noted. The calculations are then repeated on B bootstrap samples of the p -values. The dependence on λ is removed by one of two methods. In Algorithm 3 of Storey (2002), $\hat{\lambda}$ minimizes the mean squared error between the bootstrap estimates and the original minimum of the estimates over the grid of λ values. Then, $\hat{\pi}_0 = \hat{\pi}_0(\hat{\lambda})$. A second method, suggested by Storey and Tibshirani (2003), uses a cubic spline smoother $\hat{f}(\lambda)$ on the set of values, $(\lambda_i, \hat{\pi}_0(\lambda_i))$. Then the smoothed value $\hat{\pi}_0 = \hat{f}(1)$. Upper confidence bounds are determined by taking the $(1 - \alpha)$ quantile of the bootstrapped $pFDR$ and FDR values, using the smoothed value $\hat{\pi}_0$ obtained by either method above.

The q -value, introduced in Storey (2001), is the $pFDR$ analogue of the p -value. For independent tests, the q -value of the observed p -value is

$$q(p) = \inf_{\gamma \geq p} \{pFDR(\gamma)\}$$

Whereas a p -value is the minimum type I error rate that can occur while rejecting a test with a given value of the test statistic, the q -value is the minimum $pFDR$ that can occur while still rejecting the test. For more details see Storey (2001) and Storey (2002). To find the q -values, `imsls_f_false_discovery_rates` implements the algorithm given in Storey and Tibshirani (2003).

Examples

Example 1

The p -values are 20 independent realizations of a uniform (0,1) random variable. The null-probability estimate, q -values, and upper confidence limits are returned.

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

int main()
{
    int i,n_tests=20,iseed=123457;
    float p1[20]={
        0.7897864, 0.5600287, 0.04625103, 0.4892959, 0.598915,
        0.2149330, 0.9683629, 0.1449932, 0.4999971, 0.2820091,
        0.3489318, 0.479333, 0.9786092, 0.02232179, 0.2329003,
        0.3600357, 0.1341173, 0.5148499, 0.5693829, 0.9914673,
    };
    float *qvals=NULL,sort_p1[20],pi0,*upper_limits=NULL;

    qvals = imsls_f_false_discovery_rates(n_tests,p1,
        IMSLS_NULL_PROB, &pi0,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_UPPER_LIMITS,&upper_limits,
        0);

    for(i=0;i<n_tests;i++){
        sort_p1[i]=p1[i];
    }
    imsls_f_sort_data(n_tests,1,sort_p1,1,0);
    printf("\nNull Probability Estimate: %4.3f\n",pi0);
    imsls_f_write_matrix("Upper Limits for pFDR and FDR:",
        2, 1, upper_limits, 0);
    printf("\n\tP-Value\t Q-Value\n");
    for(i=0;i<n_tests;i++){
        printf("\t %4.3f \t %4.3f\n", sort_p1[i], qvals[i]);
    }
}
```

Output

```
Null Probability Estimate: 0.500
```

```
Upper Limits for pFDR and FDR:
```

```
  1      1.000
  2      0.869
```

```
 P-Value  Q-Value
 0.022    0.223
 0.046    0.231
 0.134    0.362
 0.145    0.362
 0.215    0.374
 0.233    0.374
```

0.282	0.374
0.349	0.374
0.360	0.374
0.479	0.374
0.489	0.374
0.500	0.374
0.515	0.374
0.560	0.374
0.569	0.374
0.599	0.374
0.790	0.465
0.968	0.496
0.979	0.496
0.991	0.496

Example 2

This example applies the cubic spline smoothing method to get an estimate of the null probability. Note that the p -values are the same as in [Example 1](#), but the null probability estimate is larger in this case.

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

int main()
{
    int i,n_tests=20,iseed=123457;
    float p1[20]={
        0.7897864, 0.5600287, 0.04625103, 0.4892959, 0.598915,
        0.2149330, 0.9683629, 0.1449932, 0.4999971, 0.2820091,
        0.3489318, 0.479333, 0.9786092, 0.02232179, 0.2329003,
        0.3600357, 0.1341173, 0.5148499, 0.5693829, 0.9914673,
    };
    float *qvals=NULL,sorted_p1[20],pi0,upper_limits[2];

    qvals = imsls_f_false_discovery_rates(n_tests,p1,
        IMSLS_NULL_PROB, &pi0,
        IMSLS_METHOD,1,
        IMSLS_RANDOM_SEED,iseed,
        IMSLS_UPPER_LIMITS_USER,upper_limits,
        0);

    for(i=0;i<n_tests;i++){
        sorted_p1[i]=p1[i];
    }
    imsls_f_sort_data(n_tests,1,sorted_p1,1,0);
    printf("\nNull Probability Estimate: %4.3f\n",pi0);
    imsls_f_write_matrix("Upper Limits for pFDR and FDR:",
        2, 1, upper_limits, 0);
    printf("\n\tP-Value\t Q-Value\n");
    for(i=0;i<n_tests;i++){
        printf("\t %4.3f \t %4.3f\n",sorted_p1[i],qvals[i]);
    }
}
```

Output

```
Null Probability Estimate: 0.870
```

Upper Limits for pFDR and FDR:	
1	1
2	1
P-Value	Q-Value
0.022	0.388
0.046	0.402
0.134	0.631
0.145	0.631
0.215	0.651
0.233	0.651
0.282	0.651
0.349	0.651
0.360	0.651
0.479	0.651
0.489	0.651
0.500	0.651
0.515	0.651
0.560	0.651
0.569	0.651
0.599	0.651
0.790	0.808
0.968	0.863
0.979	0.863
0.991	0.863

Warning Errors

IMSLS_NULL_PROBABILITY_0

The null probability estimate is ≤ 0 . Check that the p -values are correct or try lowering the maximum "lambda" value, which is currently = #.

yates

Estimates missing observations in designed experiments using Yate's method.

Synopsis

```
#include <ims1s.h>
```

```
int ims1s_f_yates(int n, int n_independent, float x[ ], ..., 0)
```

The type *double function* is `ims1s_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 `ims1s_f_machine(6)` or `ims1s_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 function 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_GET_SS_W_DATA, float fcn (int n, int n_independent, int n_levels[ ],
                                                  float dataMatrix[ ], void *data), void *data,
                  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 function 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:

0	<i>CRD</i> – Completely Randomized Design. The input matrix, <i>x</i> , 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, <i>n_independent</i> =1. Default value when <i>n_independent</i> =1.
1	<i>RCBD</i> – 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, <i>n_independent</i> =2. This is the default value when <i>n_independent</i> =2.
2	Another design. In this case, the function <i>get_ss</i> is a required input. The design matrix is passed to that function. Initial values for missing observations are set to the grand mean of the data, unless initial values are specified using <i>IMSLS_INITIAL_ESTIMATES</i> .

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 function 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)

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_GET_SS** 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.

Arguments

int n (Input)

Number of observations.

int n_independent (Input)

Number of independent variables.

int n_levels[] (Input)

An array **n_levels** and 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.

float dataMatrix[] (Input)

dataMatrix should be equal to the input matrix **x** with missing values replaced by estimates. **dataMatrix** should not contain any missing observations.

Return Value

Returns the error sum of squares.

IMSLS_GET_SS_W_DATA, *float fcn* (*int n*, *int n_independent*, *int n_levels[]*,
float dataMatrix[], *void *data*), *void *data*, (Input/Output)

float fcn (*int n*, *int n_independent*, *int n_levels[]*, *float dataMatrix[]*, *void *data*)
(Input)

A user-supplied function that returns the error sum of squares calculated using the **n** by **(n_independent+1)** matrix **dataMatrix**. **IMSLS_GET_SS** is required input when **design=2**. See the [Introduction, Passing Data to User-Supplied Functions](#) at the beginning of this manual for more details.

Arguments

int n (Input)

Number of observations.

int n_independent (Input)

Number of independent variables.

int n_levels[] (Input)

An array **n_levels** and 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.

float dataMatrix[] (Input)

dataMatrix should be equal to the input matrix **x** with missing values replaced by estimates. **dataMatrix** should not contain any missing observations.

*void *data* (Input/Output)

A pointer to the data to be passed to the user-supplied function.

Return Value

Returns the error sum of squares.

*void *data* (Input)

A pointer to the data to be passed to the user-supplied function.

IMSLI_S_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: $\text{grad_tol} = \epsilon^{2/3}$, where ϵ is the machine precision.

IMSLI_S_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: $\text{step_tol} = \epsilon^{2/3}$, where ϵ is the machine precision.

IMSLI_S_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`.

IMSLI_S_MISSING_INDEX_USER, *int missing_index[]* (Output)

Storage for the array `missing_index`, provided by the user.

IMSLI_S_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 i -th observation for the i -th treatment then the error sum of squares for a CRD is calculated using the following formula:

$$SSE = \sum_{i=1}^t \sum_{j=1}^r (y_{ij} - \bar{y}_i)^2 \text{ where } \bar{y}_i \text{ is the } i\text{th treatment mean.}$$

If an observation y_{ij} is missing then SSE is minimized by replacing that missing observation with the estimate

$$\hat{x}_{ij} = \bar{y}_i.$$

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:

$$SSE = \sum_{i=1}^t \sum_{j=1}^r (y_{ij} - \bar{y}_i - \bar{y}_{.j} + \bar{\bar{y}}_{..})^2$$

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:

$$\hat{x}_{ij} = \bar{y}_i + \bar{y}_{.j} - \bar{\bar{y}}_{..}$$

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 $r = \mathbf{n_blocks}$, $t = \mathbf{n_treatments}$, $y_{i.}$ = total of all non-missing observations from the i -th 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, `ims1s_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, $\hat{x}_{ij} = \bar{y}_i$.

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 `IMSL_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 [imsls_f_anova_factorial](#).

```
#include <imsls.h>
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) imsls_free(test_effects);
if (anova_table != NULL) imsls_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 an $n = 24$ by 4 matrix with two missing observations.

$$X = \begin{bmatrix} 1 & 1 & 1 & | & \text{NaN} \\ 1 & 2 & 1 & | & 53.8 \\ 1 & 3 & 1 & | & 49.5 \\ 1 & 1 & 2 & | & 41.6 \\ 1 & 2 & 2 & | & \text{NaN} \\ 1 & 3 & 2 & | & 53.8 \\ 2 & 1 & 1 & | & 53.3 \\ 2 & 2 & 1 & | & 57.6 \\ 2 & 3 & 1 & | & 59.8 \\ 2 & 1 & 2 & | & 69.6 \\ 2 & 2 & 2 & | & 69.6 \\ 2 & 3 & 2 & | & 65.8 \\ 3 & 1 & 1 & | & 62.3 \\ 3 & 2 & 1 & | & 63.4 \\ 3 & 3 & 1 & | & 64.5 \\ 3 & 1 & 2 & | & 58.5 \\ 3 & 2 & 2 & | & 50.4 \\ 3 & 3 & 2 & | & 46.1 \\ 4 & 1 & 1 & | & 75.4 \\ 4 & 2 & 1 & | & 70.3 \\ 4 & 3 & 1 & | & 68.8 \\ 4 & 1 & 2 & | & 65.6 \\ 4 & 2 & 2 & | & 67.3 \\ 4 & 3 & 2 & | & 65.3 \end{bmatrix}$$

The program below 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

int main()
{
    char *col_labels[] = {" ", "Whole", "Split", "Block", " "};
    int i;
```

```

int n = N;
int n_independent = N_INDEPENDENT;
int whole[N]={1,1,1,1,1,1,1,
              2,2,2,2,2,2,2,
              3,3,3,3,3,3,3,
              4,4,4,4,4,4,4};
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) imsls_free(test_effects);
    if (anova_table != NULL) imsls_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

Categorical and Discrete Data Analysis

Functions

Statistics in the Two-Way Contingency Table

Two-way contingency table analysis	contingency_table	519
Exact probabilities in an $r \times c$ table; total enumeration	exact_enumeration	533
Exact probabilities in an $r \times c$ table	exact_network	536

Categorical Models

Generalized linear models	categorical_glm	543
Logistic regression model	logistic_regression	560
Logistic regression prediction	logistic_reg_predict	576

Usage Notes

Function `imsls_f_contingency_table` computes many statistics of interest in a two-way table. Statistics computed by this function 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 function 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 function `imsls_f_categorical_glm` in the second section is concerned with generalized linear models (see McCullagh and Nelder 1983) in discrete data. This function 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 functions but will be fit by a quasi-Newton algorithm rather than through iterative proportional fitting.

Two additional functions, `imsls_f_logistic_regression` and `imsls_f_logistic_reg_predict`, are designed specifically for logistic regression. `imsls_f_logistic_regression` estimates a logistic regression model for binomial and multinomial response variables and one or more independent variables. Given an estimated model, `imsls_f_logistic_reg_predict` calculates predicted responses at new values of the regression variables.

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 $(n_rows + 1) \times (n_columns + 1)$ 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 $(n_rows + 1) \times (n_columns + 1)$ 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.

`IMSL5_CONTRIBUTIONS_USER`, *float* `chi_squared_contributions[]` (Output)
 Storage for array `chi_squared_contributions` is provided by the user. See `IMSL5_CONTRIBUTIONS`.

`IMSL5_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 `IMSL5_CHI_SQUARED`).

The chi-squared statistics are given as follows:

	Chi-squared Statistics
0	exact mean
1	exact standard deviation
2	Phi
3	contingency coefficient
4	Cramer's <i>V</i>

`IMSL5_CHI_SQUARED_STATS_USER`, *float* `chi_squared_stats[]` (Output)
 Storage for array `chi_squared_stat` is provided by the user. See `IMSL5_CHI_SQUARED_STATS`.

`IMSL5_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.

	Statistic
0	Gamma
1	Kendall's τ_b
2	Stuart's τ_c
3	Somers' <i>D</i> for rows (given columns)
4	Somers' <i>D</i> 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 <i>U</i> (symmetric)
10	uncertainty $U_{r c}$ (rows)

	Statistic
11	uncertainty $U_{c r}$ (columns)
12	optimal prediction λ (symmetric)
13	optimal prediction $\lambda_{r c}$ (rows)
14	optimal prediction $\lambda_{c r}$ (columns)
15	optimal prediction $\lambda_{r c}$ (rows)
16	optimal prediction $\lambda_{c r}$ (columns)
17	test for linear trend in row probabilities if <code>n_rows = 2</code> If <code>n_rows</code> is not 2, a test for linear trend in column probabilities if <code>n_columns = 2</code> .
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:

	Value
0	estimated statistic
1	standard error for any parameter value
2	standard error under the null hypothesis
3	<i>t</i> value for testing the null hypothesis
4	<i>p</i> -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` \times `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 $p_{ij} = p_{i\bullet}p_{\bullet j}$ denote the predicted cell probabilities under the null hypothesis of independence, where $p_{i\bullet}$ and $p_{\bullet j}$ 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 $u, v = 1, \dots, n$. 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., $H_0: p_{ij} = p_{i\bullet}p_{\bullet j}$. 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:

$$G^2 = -2 \sum_{i,j} x_{ij} \ln(x_{ij} / np_{ij})$$

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:

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

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

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

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 $n^2 m / (m - 1)$, 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, Somers' D for rows is the probability of agreement minus the probability of disagreement, given that the column variable, b_{uv} , is not 0 . Somers' D for columns is defined in a similar manner.

A discussion of all of the measures of association in this section can be found in Kendall and Stuart (1979, p. 592).

Measures of Prediction and Uncertainty

Optimal Prediction Coefficients: The measures in this section do not require any ordering of the row or column variables. They are based entirely upon probabilities. Most are discussed in Bishop et al. (1975, p. 385).

Consider predicting (or classifying) the column for a given row in the table. Under the null hypothesis of independence, choose the column with the highest column marginal probability for all rows. In this case, the probability of misclassification for any row is 1 minus this marginal probability. If independence is not assumed within each row, choose the column with the highest row conditional probability. The probability of misclassification for the row becomes 1 minus this conditional probability.

Define the optimal prediction coefficient $\lambda_{c|r}$ for predicting columns from rows as the proportion of the probability of misclassification that is eliminated because the random variables are not independent. It is estimated by

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

where m is the index of the maximum estimated probability in the row (p_{im}) or row margin ($p_{\bullet m}$). A similar coefficient is defined for predicting the rows from the columns. The symmetric version of the optimal prediction λ is obtained by summing the numerators and denominators of $\lambda_{r|c}$ and $\lambda_{c|r}$, then dividing. Standard errors for these coefficients are given in Bishop et al. (1975, p. 388).

A problem with the optimal prediction coefficients λ is that they vary with the marginal probabilities. One way to correct this is to use row conditional probabilities. The optimal prediction 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

$$\lambda_{c|r}^* = \frac{\sum_i \max_j p_{j|i} - \max_j (\sum_i p_{j|i})}{R - \max_j (\sum_i p_{j|i} p_{j|i})}$$

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:

$$n/2 - \left(\sum_i x_{i\bullet}^2 \right) / (2n)$$

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:

$$q_j = x_{\cdot j} / 2 - \left(\sum_i x_{ij}^2 \right) / (2x_{i\cdot})$$

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:

$$U_{r|c} = \frac{\sum_{i,j} x_{ij} \log(x_{i\cdot} x_{\cdot j} / nx_{ij})}{\sum_i x_{i\cdot} \log(x_{i\cdot} / n)}$$

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

$$\hat{\beta} = \frac{\sum_j x_{\cdot j} (x_{1j} / x_{\cdot j} - x_{1\cdot} / n) (j - \bar{j})}{\sum_j x_{\cdot j} (j - \bar{j})^2}$$

where

$$\bar{j} = \sum_j x_{\cdot j} j / n$$

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:

$$\sum_{i < j} \frac{(x_{ij} - x_{ji})^2}{(x_{ij} + x_{ji})}$$

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:

$$\frac{\left(\sum_{i < j} (x_{ij} - x_{ji}) \right)^2}{\sum_{i < j} (x_{ij} + x_{ji})}$$

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>
#include <stdio.h>

int 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>
#include <stdio.h>

int 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.",
    "1-star-rows", "1-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

`IMSL_S_PROB_TABLE`, *float* *`prt` (Output)

Probability of the observed table occurring, given that the null hypothesis of independent rows and columns is true.

`IMSL_S_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`.

`IMSL_S_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 $r = \mathbf{n_rows}$ and $c = \mathbf{n_columns}$. 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 probabilistic 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 $f_{..}$ 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>

int 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`, *int* *`n_attempts` (Input/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 inadequate, 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 $f_{\bullet\bullet} = \sum \sum f_{ij}$ equal to the sum of all cell frequencies in the observed table, $nt = f_{\bullet\bullet} + 1$, $mx = \max(\mathbf{n_rows}, \mathbf{n_columns})$,

$mn = \min(\mathbf{n_rows}, \mathbf{n_columns})$, $t1 = \max(800 + 7mx, (5 + 2mx)(\mathbf{n_rows} + \mathbf{n_columns} + 1))$, and $t2 = \max(400 + mx, + 1, \mathbf{n_rows} + \mathbf{n_columns} + 1)$.

The following amount of integer workspace is allocated: $3mx + 2mn + t1$.

The following amount of *float* (or *double*, if using `imsls_d_exact_network`) work-space 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 $6n(\mathbf{factor1} + \mathbf{factor2})$. The amount of real workspace allocated is $n(6\mathbf{factor1} + 2\mathbf{factor2})$. Variable n is the index for the attempt, $1 < n \leq \mathbf{max_attempts}$.

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 $r = \mathbf{n_rows}$ and $c = \mathbf{n_columns}$. 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

$$P_f = \frac{\prod_{i=1}^r f_{i\cdot}! \prod_{j=1}^c f_{\cdot j}!}{f_{\cdot\cdot}! \prod_{i=1}^r \prod_{j=1}^c f_{ij}!}$$

where $f_{\cdot\cdot}$ 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 probabilistic 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>

int 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 Algorithm 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 Algorithm 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>

int 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:

<code>model</code>	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

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

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_TOLERANCE, float tolerance,
```

```
    IMSLS_MAX_ITERATIONS, int max_iterations,
```

```
    IMSLS_INTERCEPT, or
```

```
    IMSLS_NO_INTERCEPT,
```

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

```

IMSLS_INITIAL_EST_INTERNAL, or
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 *ifrq* of **x** containing the frequency of response for each observation.

IMSLS_X_COL_FIXED_PARAMETER, *int ifix* (Input)

Column number *ifix* of **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* of **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		Meaning of x [<i>i</i>] [<i>ipar</i>]
0	E	ln (E) is a fixed intercept to be included in the linear predictor (i.e., the <i>offset</i>).
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_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 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** that 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** = $\sqrt{10} \epsilon$, where ϵ is the machine precision.

`IMSLS_TOLERANCE`, *float tolerance* (Input)

Tolerance used in determining linear dependence. When linear dependence is detected, terminal error `IMSLS_RANK_DEFICIENT_FATAL` is issued and no results are computed.

Computations for a rank deficient model can be forced to continue by specifying a negative tolerance. If `tolerance` is negative, the absolute value of `tolerance` will be used to determine linear dependence, but computations will proceed with warning `IMSLS_RANK_DEFICIENT_WARN`. In this case the results should be carefully inspected and used with caution.

Default: `tolerance` = 10ϵ , where ϵ is the machine precision.

`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` (input)

or

`IMSLS_NO_INTERCEPT` (Input)

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 \mathbf{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` (Input)

or

`IMSLS_INITIAL_EST_INPUT`, *int n_coef_input, float estimates[]* (Input)

By default, or if `IMSLS_INIT_EST_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. This requires that the user knows 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`. The returned value has to be increased by one if optional argument `IMSLS_INTERCEPT` is used in the `categorical_glm` call.

`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` 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

$$\sum_{i=0}^{n_class-1} n_class_values [i]$$

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`.

	Parameter
0	Coefficient Estimate.
1	Estimated standard deviation of the estimated coefficient.
2	Asymptotic normal score for testing that the coefficient is zero.
3	The <i>p</i> -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)

Optimization criterion. The maximized log-likelihood, i.e., the value of the log-likelihood at the final parameter estimates.

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[]* (Output)

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.

0	Predicted mean for the observation if <code>model = 0</code> . 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.

<code>obs_status [i]</code>	Status of observation
0	Observation <i>i</i> is in the likelihood
1	Observation <i>i</i> cannot be in the likelihood because it contains at least one missing value in <i>x</i> .
2	Observation <i>i</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 $(\text{max_iterations} + 1) \times 5$ containing in its first `n` rows information about the start and each iteration of the analysis.

0	Method of iteration. Equal to 0 if a Q-N step was taken. Equal to 1 if a N-R step was taken.
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_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 function `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 re-weighted 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		
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	$f(y) = -(1-\theta)^y / (y \cdot \ln \theta)$	$\theta = \frac{\exp(\omega + \eta)}{1 + \exp(\omega + \eta)}$
3	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$	$\theta = \frac{\exp(\omega + \eta)}{1 + \exp(\omega + \eta)}$

model		
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, \mathbf{y}_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 $n = \text{parameter}[i]$ 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

$$\hat{\theta} = \mathbf{y}[i] / \text{parameter}[i]$$

and, when `model` = 3, the linear relationship is given by

$$\ln\left(\frac{\hat{\theta}}{1 - \hat{\theta}}\right) \approx X\beta$$

while if `model = 4`, $\Phi^{-1}(\theta) = X\beta$. 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](#)). Also, at this step of the computations, the regression function is used to detect linear dependence in the model, by the method described for `imsls_f_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

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

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 `max_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

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

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 subtracted 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, the `ifrq` column of `x` 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 (distribution parameter) 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 N 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:

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 $\theta = 0.5$.

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

int 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. Surveillance 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			
		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>

int 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.
IMSLS_RANK_DEFICIENT_WARN	The model is rank deficient (<code>rank = #</code>). Computations will proceed per setting of <code>IMSLS_TOLERANCE</code> . Check results for accuracy.

Fatal Errors

IMSLS_TOO_FEW_COEF	<code>IMSLS_INITIAL_EST_INPUT</code> is specified and <code>"n_coef_input" = #</code> . The model specified requires <code>#</code> coefficients.
IMSLS_MAX_CLASS_TOO_SMALL	The number of distinct values of the classification variables exceeds <code>"max_class" = #</code> .
IMSLS_INVALID_DATA_8	<code>"n_class_values[#]" = #</code> . 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 <code>"lp_max" = #</code> . Rerun with a different model or increase the workspace.
IMSLS_RANK_DEFICIENT_TERM	The model is rank deficient (<code>rank = #</code>). No solution will be computed. Refer to the documentation of optional argument <code>IMSLS_TOLERANCE</code> for other options.

logistic_regression



[more...](#)

Fit a binomial or multinomial logistic regression model using iteratively re-weighted least squares.

Synopsis

```
#include <imsls.h>

float *imsls_f_logistic_regression(int n_observations, int n_independent,
    int n_classes, float x[ ], float y[ ], ..., 0)
```

The type *double* function is `imsls_d_logistic_regression`.

Required Arguments

int n_observations (Input)

The number of observations.

int n_independent (Input)

The number of independent variables.

int n_classes (Input)

The number of discrete outcomes, or classes.

float x[] (Input)

An array of length `n_observations × n_independent` containing the values of the independent variables corresponding to the responses in `y`.

float y[] (Input)

An array of length `n_observations × n_classes` containing the binomial (`n_classes = 2`) or multinomial (`n_classes > 2`) counts per class. In an alternate format, `y` is an array of length `n_observations × (n_classes - 1)` containing the counts for all but one class. The missing class is treated as the reference class. The optional argument `GROUP_COUNTS` specifies this format for `y`. In another alternative format, `y` is an array of length `n_observations` containing the class id's. See optional argument `IMSLS_GROUPS`.

Return Value

Pointer to an array of length $n_coefficients \times n_classes$ containing the estimated coefficients. The function fits a full model, where $n_coefficients = 1 + n_independent$. The optional arguments `IMSLS_NO_INTERCEPT`, `IMSLS_X_INDICES`, and `IMSLS_X_INTERACTIONS` may be used to specify different models. Note that the last column (column $n_classes$) represents the reference class and is set to all zeros.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_logistic_regression (int n_observations, int n_independent,
    int n_classes, float x[], float y[],

    IMSLS_GROUP_COUNTS, or
    IMSLS_GROUPS,

    IMSLS_COLUMN_WISE,

    IMSLS_FREQUENCIES, int frequencies[],

    IMSLS_REFERENCE_CLASS, int ref_class,

    IMSLS_NO_INTERCEPT,

    IMSLS_X_INDICES, int n_xin, int xin[],

    IMSLS_X_INTERACTIONS, int n_xinteract, int xinteract[],

    IMSLS_TOLERANCE, float tolerance,

    IMSLS_MAX_ITER, int max_iter,

    IMSLS_INIT_INPUT, int init,

    IMSLS_PREV_RESULTS, imsls_f_model *prev_model,

    IMSLS_NEXT_RESULTS, imsls_f_model **next_model,

    IMSLS_COEFFICIENTS, float coefficients[],

    IMSLS_LRSTAT, float *lrstat,

    0)
```

Optional Arguments

`IMSLS_GROUP_COUNTS` (Input)

or

`IMSLS_GROUPS` (Input)

These optional arguments specify the format of the input array \mathbf{y} . If `IMSLS_GROUP_COUNTS` is present, \mathbf{y} is of length `n_observations` \times (`n_classes` - 1), and contains counts for all but one of the classes for each observation. The missing class is treated as the reference class.

If `IMSLS_GROUPS` is present, the input array \mathbf{y} is of length `n_observations`, and $\mathbf{y}[i]$ contains the group or class number to which the observation belongs. In this case, `frequencies[i]` is set to 1 for all observations.

Default: \mathbf{y} is `n_observations` \times (`n_classes`), and contains counts for all the classes.

`IMSLS_COLUMN_WISE` (Input)

If present, the input arrays are column-oriented. That is, contiguous elements in \mathbf{x} are values of the same independent variable, or column, except at multiples of `n_observations`.

Default: Input arrays are row-oriented.

`IMSLS_FREQUENCIES`, `intfrequencies[]` (Input)

An array of length `n_observations` containing the number of replications or trials for each of the observations. This argument is required if `IMSLS_GROUP_COUNTS` is specified and any element of $\mathbf{y} > 1$.

Default: `frequencies[i] = 1`.

`IMSLS_REFERENCE_CLASS`, `intref_class` (Input)

Number specifying which class or outcome category to use as the reference class. See the [Description](#) section for details.

Note that the last column of `coefficients` always represents the reference class. So when `ref_class < n_classes`, columns `ref_class` and `n_classes` are swapped for the output `coefficients`, i.e. coefficients for class `n_classes` will be returned in column `ref_class` of `coefficients`. For example, if `ref_class = 1` and `n_classes = 3`, the first column of `coefficients` contains the coefficients for class 3 (`n_classes`), the second column contains the coefficients for class 2, and the third column contains all zeros for the reference class.

Default: `ref_classes=n_classes`

`IMSLS_NO_INTERCEPT` (Input)

If present, the model will not include an intercept term.

Default: The intercept term is included.

IMSLS_X_INDICES, (Input)

An array of length `n_xin` providing the column indices of `x` that correspond to the independent variables the user wishes to be included in the logistic regression model. For example, suppose there are five independent variables `x0`, `x1`, ..., `x4`. To fit a model that includes only `x2` and `x3`, set `n_xin = 2`, `xin[0] = 2`, and `xin[1] = 3`.

Default: All `n_independent` variables are included.

IMSLS_X_INTERACTIONS, (Input)

An array of length `n_xinteract × 2` providing pairs of column indices of `x` that define the interaction terms in the model. Adjacent indices should be unique. For example, suppose there are two independent variables `x0` and `x1`. To fit a model that includes their interaction term, `x0x1`, set `n_xinteract = 1`, `xinteract[0] = 0`, and `xinteract[1] = 1`.

Default: No interaction terms are included.

IMSLS_TOLERANCE, *float tolerance* (Input)

Convergence error criteria. Iteration completes when the normed difference between successive estimates is less than `tolerance` or `max_iter` iterations are reached.

Default: `tolerance = 100.00 × imsls_f_machine(4)`

IMSLS_MAX_ITER, *intmax_iter* (Input)

The maximum number of iterations.

Default: `max_iter = 20`

IMSLS_INIT_INPUT, *intinit* (Input)

`init` must be 0 or 1. If `init = 1`, initial values for the coefficient estimates are provided in the user array `coefficients`. If `init = 0`, a default is set within the function. The default setting is the zero vector.

Default: `init = 0`

IMSLS_PREV_RESULTS, *Imsls_f_model *prev_model* (Input)

Pointer to a structure of type `Imsls_f_model` containing information about a previous logistic regression fit. The model is combined with the fit to new data or to **IMSLS_NEXT_RESULTS**, if provided.

IMSLS_NEXT_RESULTS, *Imsls_f_model **next_model* (Input/Output)

Address of a pointer to a structure of type `Imsls_f_model`. If present and `NULL`, the structure is internally allocated and on output contains the model information. If present and not `NULL`, its contents are combined with the fit to new data or to **IMSLS_PREV_RESULTS**, if provided. The combined results are returned in `next_model`.

IMSLS_COEFFICIENTS, *floatcoefficients[]* (Input/Output)

Storage for the coefficient array of length `n_coefficients × n_classes` is provided by the user. When `init = 1`, `coefficients` should contain the desired initial values of the estimates.

IMSLs_LRSTAT, *float*lrstat* (Output)

The value of the likelihood ratio test statistic.

Description

Function `imsls_f_logistic_regression` fits a logistic regression model for discrete dependent variables with two or more mutually exclusive outcomes or classes. For a binary response y , the objective is to model the conditional probability of success, $\pi_1(x) = \Pr[y = 1 | x]$, where $x = (x_1, x_2, \dots, x_p)'$ is a realization of p independent variables. Logistic regression models the conditional probability, $\pi_1(x)$, using the cdf of the logistic distribution. In particular,

$$\pi_1(x) = \frac{\exp(\eta_1)}{1 + \exp(\eta_1)} \equiv \frac{1}{1 + \exp(-\eta_1)}$$

where

$$\eta_1 = \beta_{01} + x^T \beta_1$$

and

$$\beta_{01}, \beta_1 = (\beta_{11}, \beta_{12}, \dots, \beta_{1p})'$$

are unknown coefficients that are to be estimated.

Solving for the linear component η_1 results in the log-odds or *logit* transformation of $\pi_1(x)$:

$$\text{logit}(\pi_1(x)) = \log \frac{\pi_1(x)}{1 - \pi_1(x)} = \eta_1$$

Given a set of N observations (y_i, x_i) , where y_i follows a binomial (n, π) distribution with parameters $n = 1$ and $\pi = \pi_1(x_i)$, the likelihood and log-likelihood are, respectively,

$$L = \prod_{i=1}^N \pi(x_i)^{y_i} (1 - \pi(x_i))^{1-y_i}$$

$$l = \sum_{i=1}^N y_i \log\left(\frac{\pi(x_i)}{1 - \pi(x_i)}\right) + \log(1 - \pi(x_i))$$

The log-likelihood in terms of the parameters, $\{\beta_{01}, \beta_1\}$, is therefore

$$l(\beta_0, \beta_1) = \sum_{i=1}^N y_i \eta_{i1} - \log(1 + \exp(\eta_{i1}))$$

where

$$\eta_1 = \beta_{01} + x^T \beta_1$$

With a binary outcome, only one probability needs to be modeled. The second probability can be obtained from the constraint, $\pi_1(x) + \pi_2(x) = 1$. If each y_i is the number of successes in n_i independent trials, the log-likelihood becomes

$$l = \sum_{i=1}^N y_i \log\left(\frac{\pi(x_i)}{1 - \pi(x_i)}\right) + n_i \log(1 - \pi(x_i))$$

or

$$l(\beta_0, \beta_1) = \sum_{i=1}^N y_i \eta_{i1} - n_i \log(1 + \exp(\eta_{i1}))$$

See optional argument `IMSL5_FREQENCIES` to set frequencies $n_i > 1$.

To test the significance of the model, the log-likelihood of the fitted model is compared to that of an intercept-only model. In particular, $G = -2 (l(\beta_{01}) - l(\beta_{01}, \beta_1))$ is a likelihood-ratio test statistic and under the null hypothesis, $H_0: \beta_{11} = \beta_{12} = \dots = \beta_{1p} = 0$, G is distributed as chi-squared with $p-1$ degrees of freedom. A significant result suggests that at least one parameter in the model is non-zero. See Hosmer and Lemeshow (2000) for further discussion.

In the multinomial case, the response vector is $y_i = (y_{i1}, y_{i2}, \dots, y_{iK})'$, where $y_{ik} = 1$ when the i -th observation belongs to class k and $y_{ik} = 0$, otherwise. Furthermore, because the outcomes are mutually exclusive,

$$\sum_{k=1}^K y_{ik} = 1$$

and $\pi_1(x) + \pi_2(x) + \dots + \pi_K(x) = 1$. The last class K serves as the baseline or reference class in the sense that it is not modeled directly but found from

$$\pi_K(x) = 1 - \sum_{k=1}^{K-1} \pi_k(x)$$

If there are multiple trials, $n_i > 1$, then the constraint on the responses is

$$\sum_{k=1}^K y_{ik} = n_i$$

The log-likelihood in the multinomial case becomes

$$l(\beta_{0l}, \beta_l)_{l=1}^K = \sum_{i=1}^N \left[\sum_{l=1}^K y_{il} \eta_{il} - \log \left(\sum_{l=1}^K \exp(\eta_{il}) \right) \right]$$

or

$$l(\beta_{0l}, \beta_l)_{l=1}^K = \sum_{i=1}^N \left[\sum_{l=1}^K y_{il} (\beta_{0l} + x_i^T \beta_l) - \log \left(\sum_{l=1}^K \exp(\beta_{0l} + x_i^T \beta_l) \right) \right]$$

The constraint

$$\sum_{k=1}^K \pi_{ik} = 1$$

is handled by setting $\eta_K = 0$ for the K -th class, and then the log-likelihood is

$$l(\beta_{0l}, \beta_l)_{l=1}^{K-1} = \sum_{i=1}^N \left[\sum_{l=1}^{K-1} y_{il} (\beta_{0l} + x_i^T \beta_l) - \log \left(\sum_{l=1}^{K-1} \exp(\beta_{0l} + x_i^T \beta_l) \right) \right]$$

Note that for the multinomial case, the log-odds (or logit) is

$$\log \frac{\pi_{il}}{\pi_{iK}} = \beta_{0l} + x_i^T \beta_l, \quad l = 1, \dots, K-1$$

Note that each of the logits involve the odds ratio of being in class l versus class K , the reference class. Maximum likelihood estimates can be obtained by solving the score equation for each parameter:

$$\begin{aligned}
 \frac{\partial l(\beta_{0l}, \beta_l)}{\partial l(\beta_{jl})} &= \sum_{i=1}^N \left[x_{ij} y_{il} - \frac{x_{ij} \exp(\eta_{il})}{\left(1 + \sum_{m=1}^{K-1} \exp(\eta_{im})\right)} \right] \\
 &= \sum_{i=1}^N \left[x_{ij} (y_{il} - \pi_{il}(x_i)) \right] \\
 &= \mathbf{0}
 \end{aligned}$$

To solve the score equations, the function employs a method known as *iteratively re-weighted least squares* or IRLS. In this case the IRLS is equivalent to the Newton-Raphson algorithm (Hastie, et. al., 2009, Thisted, 1988).

Consider the full vector of parameters

$$\beta = (\beta_{01}, \beta_{11}, \dots, \beta_{p1}, \beta_{02}, \beta_{12}, \dots, \beta_{p2}, \dots, \beta_{0K-1}, \beta_{1K-1}, \dots, \beta_{pK-1})'$$

the Newton-Raphson iteration is

$$\beta^{n+1} = \beta^n - \mathbf{H}^{-1}(\beta^n) \mathbf{G}(\beta^n)$$

where H denotes the Hessian matrix, i.e., the matrix of second partial derivatives defined by

$$\begin{aligned}
 \frac{\partial^2 l(\beta_{0l}, \beta_l)}{\partial \beta_{kl} \partial \beta_{jl}} &= - \sum_{i=1}^N \left[\frac{x_{ij} x_{ik} \exp(\eta_{il}) \sum_{m=1}^K \exp(\eta_{im}) - x_{ij} x_{ik} \exp(\eta_{il}) \exp(\eta_{il})}{\left(\sum_{m=1}^K \exp(\eta_{im})\right)^2} \right] \\
 &= - \sum_{i=1}^N \left[x_{ij} x_{ik} p_{il} (1 - p_{il}) \right]
 \end{aligned}$$

and

$$\begin{aligned}
 \frac{\partial^2 l(\beta_{0l}, \beta_l)}{\partial \beta_{kv} \partial \beta_{jl}} &= - \frac{1}{N} \sum_{i=1}^N \left[\frac{-x_{ij} x_{ik} \exp(\eta_{il}) \exp(\eta_{iv})}{\left(\sum_{l=1}^K \exp(\eta_{il})\right)^2} \right] = \frac{1}{N} \sum_{i=1}^N \left[\frac{x_{ij} x_{ik} \exp(\eta_{il}) \exp(\eta_{iv})}{\left(\sum_{l=1}^K \exp(\eta_{il})\right)^2} \right] = \\
 &= \frac{1}{N} \sum_{i=1}^N \left[x_{ij} x_{ik} p_{il} p_{iv} \right]
 \end{aligned}$$

and G denotes the gradient vector, the vector of first partial derivatives,

$$\frac{\partial l(\beta_{0l}, \beta_l)}{\partial l(\beta_{jl})}$$

Both the gradient and the Hessian are evaluated at the most recent estimate of the parameters, β^n . The iteration continues until convergence or until maximum iterations are reached. Following the theory of maximum likelihood estimation (Kendall and Stuart, 1979), standard errors are obtained from Fisher’s information matrix $(-H)^{-1}$ evaluated at the final estimates.

When the `IMSLS_NEXT_RESULTS` option is specified, the function combines estimates of the same model from separate fits using the method presented in Xi, Lin, and Chen (2008). To illustrate, let β_1 and β_2 be the MLE’s from separate fits to two different sets of data, and let H_1 and H_2 be the associated Hessian matrices. Then the combined estimate,

$$\beta = (\mathbf{H}_1 + \mathbf{H}_2)^{-1}(\beta_1^T \mathbf{H}_1 + \beta_2^T \mathbf{H}_2)$$

approximates the MLE of the combined data set. The model structure, `imsls_f_model**next_model` contains the combined estimates as well as other elements. See Table 1: *imsls_f_model Data Structure* below.

Table 34 – The `imsls_f_model` Data Structure

		Description
<code>n_obs</code>	<i>int</i>	Total number of observations. If the model structure has been updated three times, first with 100 observations, next with 50, and third with 50, then <code>n_obs</code> = 200.
<code>n_updates</code>	<i>int</i>	Total number of times the model structure has been updated. In the above scenario, <code>n_updates</code> = 3.
<code>n_coefs</code>	<i>int</i>	Number of coefficients in the model. This parameter must be the same for each model update.
<code>coefs</code>	<i>float[]</i>	An array of length <code>n_coefs</code> × <code>n_classes</code> containing the coefficients.
<code>meany</code>	<i>float[]</i>	An array of length <code>n_classes</code> containing the overall means for each class variable.
<code>stderrs</code>	<i>float[]</i>	An array of length <code>n_coefs</code> ×(<code>n_classes</code> – 1) containing the estimated standard errors for the estimated coefficients.
<code>grad</code>	<i>float[]</i>	An array of length <code>n_coefs</code> ×(<code>n_classes</code> – 1) containing the estimated gradient at the coefficient estimates.
<code>hess</code>	<i>float[]</i>	An array of length <code>n_coefs</code> ×(<code>n_classes</code> – 1)× <code>n_coefs</code> ×(<code>n_classes</code> – 1) containing the estimated Hessian matrix at the coefficient estimates.

Remarks

Iteration stops when the estimates converge within tolerance, when maximum iterations are reached, or when the gradient becomes within tolerance of 0, whichever event occurs first. When the gradient converges before the coefficient estimate converges, a condition in the data known as complete or quasi-complete separation may be present. Separation in the data means that one or more independent variable perfectly predicts the response. When detected, the function stops the iteration, issues a warning, and returns the current values of the model estimates. Some of the coefficient estimates and standard errors may not be reliable. Furthermore, overflow issues may occur before the gradient converges. In such cases the program issues a fatal error.

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):

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 is the binomial response (`n_classes = 2`) and the log-dosage is the single independent variable. Note that this example illustrates the `GROUP_COUNTS` format for \mathbf{y} and the optional argument `IMSL_FREQUENCIES`.

```
#include <imsls.h>

int main(){
    float y1[8]={6,13,18,28,52,53,61,60};
    float x1[8]={1.69,1.724,1.755,1.784,1.811,1.836,1.861,1.883};
    float freqs[8]={59,60,62,56,63,59,62,60};
    float *coefs;
    int n_classes=2,n_observations=8,n_independent=1,n_coefs=2;
```

```

    coefs=imsls_f_logistic_regression(n_observations,
        n_independent,n_classes,x1,y1,
        IMSLS_GROUP_COUNTS,
        IMSLS_FREQUENCIES,freqs,
        0);

    imsls_f_write_matrix("Coefficient Estimates",
        (n_coefs)*(n_classes-1),1,coefs,0);
}

```

Output

```

Coefficient Estimates
1      -60.76
2       34.30

```

Example 2

In this example the response is a multinomial random variable with 4 outcome classes. The 3 independent variables represent 2 categorical variables and 1 continuous variable. A subset of 2 independent variables along with the intercept defines the logistic regression model. A test of significance is performed.

```

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

int main(){
    float x[50*3]={
        3, 2, 2, 1, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 2, 3, 2, 1, 3, 2,
        2, 1, 2, 1, 3, 2, 1, 2, 1, 2, 3, 2, 1, 2, 1, 1, 2, 3, 1, 2,
        1, 1, 1, 3, 1, 3, 2, 3, 3, 1,
        25.92869, 51.63245, 25.78432, 39.37948, 24.65058, 45.20084,
        52.6796, 44.28342, 40.63523, 51.76094, 26.30368, 20.70230,
        38.74273, 19.47333, 26.42211, 37.05986, 51.67043, 42.40156,
        33.90027, 35.43282, 44.30369, 46.72387, 46.99262, 36.05923,
        36.83197, 61.66257, 25.67714, 39.08567, 48.84341, 39.34391,
        24.73522, 50.55251, 31.34263, 27.15795, 31.72685, 25.00408,
        26.35457, 38.12343, 49.9403, 42.45779, 38.80948, 43.22799,
        41.87624, 48.0782, 43.23673, 39.41294, 23.93346,
        42.8413, 30.40669, 37.77389,
        1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1,
        1, 1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 2,
        2, 2, 1, 1, 2, 1, 1, 2, 1, 1
    };
    float y[50]={
        1, 2, 3, 4, 3, 3, 4, 4, 4, 4, 2, 1, 4, 1, 1, 1, 4, 4, 3, 1,
        2, 3, 3, 4, 2, 3, 4, 1, 2, 4, 3, 4, 4, 1, 3, 4, 4, 2, 3, 4,
        2, 2, 4, 3, 1, 4, 3, 4, 2, 3
    };
    float *coefs,*preds,model_pval,lrstat;
    int xindices[2],dof,n_classes=4,n_observations=50,
        n_independent=3,n_coefs=3;

    Imsls_f_model *model=NULL;

    xindices[0]=0;

```

```

xindices[1]=1;

coefs=imsls_f_logistic_regression(n_observations,
    n_independent,n_classes,x,y,
    IMSLS_GROUPS,
    IMSLS_COLUMN_WISE,
    IMSLS_X_INDICES,2,xindices,
    IMSLS_LRSTAT,&lrstat,
    IMSLS_NEXT_RESULTS,&model,0);

dof = n_coefs*(n_classes-1) - (n_classes-1);
model_pval = 1.0 - imsls_f_chi_squared_cdf(lrstat,dof);

imsls_f_write_matrix("Coefficients",(n_coefs)*(n_classes-1),
    1,coefs,0);
imsls_f_write_matrix("Std Errs",n_coefs*(n_classes-1),1,
    model->stderrs,0);
printf("\nLog-likelihood: %5.2f\n",model->loglike);
printf("LR test statistic: %5.2f\n%d deg. freedom, "
    "p-value: %5.4f\n",lrstat,dof,model_pval,0);
}

```

Output

```

Coefficients
1      2.292
2      0.408
3     -0.111
4     -1.162
5      0.245
6     -0.002
7     -0.067
8      0.178
9     -0.017

```

```

Std Errs
1      2.259
2      0.548
3      0.051
4      2.122
5      0.500
6      0.044
7      1.862
8      0.442
9      0.039

```

```

Log-likelihood: -62.92
LR test statistic: 7.68
6 deg. freedom, p-value: 0.2623

```

Example 3

Example 3 uses the same data as in [Example 2](#) and an additional set of 50 observations using the same data generating process. The model structure includes all 3 independent variables and an intercept, and a single model fit is approximated from two separate model fits. Example 3 also includes a fit on the full data set for comparison purposes.

```

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

int main(){
    float x1[50 * 3]={
        3, 2, 2, 1, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 2, 3, 2, 1, 3, 2, 2,
        1, 2, 1, 3, 2, 1, 2, 1, 2, 3, 2, 1, 2, 1, 1, 2, 3, 1, 2, 1, 1,
        1, 3, 1, 3, 2, 3, 3, 1,
        25.92869, 51.63245, 25.78432, 39.37948, 24.65058, 45.20084,
        52.6796, 44.28342, 40.63523, 51.76094, 26.30368, 20.70230,
        38.74273, 19.47333, 26.42211, 37.05986, 51.67043, 42.40156,
        33.90027, 35.43282, 44.30369, 46.72387, 46.99262, 36.05923,
        36.83197, 61.66257, 25.67714, 39.08567, 48.84341, 39.34391,
        24.73522, 50.55251, 31.34263, 27.15795, 31.72685, 25.00408,
        26.35457, 38.12343, 49.9403, 42.45779, 38.80948, 43.22799,
        41.87624, 48.0782, 43.23673, 39.41294, 23.93346,
        42.8413, 30.40669, 37.77389,
        1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 1,
        1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2,
        1, 1, 2, 1, 1, 2, 1, 1
    };

    float x2[50 * 3]={
        1, 1, 3, 3, 2, 3, 3, 3, 2, 1, 1, 1, 1, 3, 3, 2, 2, 3, 3, 2, 3,
        2, 1, 3, 3, 2, 2, 3, 3, 2, 1, 2, 1, 2, 3, 3, 1, 1, 2, 2, 3, 1,
        1, 2, 2, 1, 1, 2, 3, 1,
        35.66064, 26.68771, 23.11251, 58.14765, 44.95038, 42.45634,
        34.97379, 53.54269, 32.57257, 46.91201, 30.93306, 51.63743,
        34.67712, 53.84584, 14.97474, 44.4485, 47.10448, 43.96467,
        55.55741, 36.63123, 32.35164, 55.75668, 36.83637, 46.7913,
        44.24153, 49.94011, 41.91916, 24.78584, 50.79019, 39.97886,
        34.42149, 41.93271, 28.59433, 38.47255, 32.11676, 37.19347,
        52.89337, 34.64874, 48.61935, 33.99104, 38.32489, 35.53967,
        29.59645, 21.14665, 51.11257, 34.20155, 44.40374, 49.67626,
        58.35377, 28.03744,
        1, 1, 2, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2,
        1, 2, 1, 2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 1, 2, 2,
        1, 1, 2, 1, 1, 2, 1, 1
    };

    float y1[50]={
        1, 2, 3, 4, 3, 3, 4, 4, 4, 4, 2, 1, 4, 1, 1, 1, 4, 4, 3, 1, 2,
        3, 3, 4, 2, 3, 4, 1, 2, 4, 3, 4, 4, 1, 3, 4, 4, 2, 3, 4, 2, 2,
        4, 3, 1, 4, 3, 4, 2, 3
    };

    float y2[50]={
        1, 4, 1, 4, 1, 1, 3, 1, 2, 4, 3, 1, 3, 2, 4, 4, 4, 2, 3, 2, 1,
        4, 4, 4, 4, 3, 1, 1, 3, 1, 4, 2, 4, 2, 1, 2, 3, 1, 1, 4, 1, 2,
        4, 3, 4, 2, 4, 3, 2, 4
    };

    float x3[100 * 3], y3[100], *coefs;
    int i, j, n_classes=4, n_observations=50;
    int n_independent = 3, n_coefs= 4;
    imsls_f_model *modell1 = NULL, *modell2 = NULL, *modell3 = NULL;

    /* first call with x1, y1 */
    coefs=imsls_f_logistic_regression(n_observations, n_independent,

```

```

    n_classes, x1, y1,
    IMSLS_GROUPS,
    IMSLS_COLUMN_WISE,
    IMSLS_NEXT_RESULTS, &modell1,
    0);

    imsls_f_write_matrix("First Model Coefficients:",
        n_coefs * (n_classes - 1), 1, modell1->coefs,
        0);
    imsls_f_write_matrix("First Model Standard Errors:",
        n_coefs * (n_classes - 1), 1, modell1->stderrs,
        0);

    imsls_free(coefs);

    /* second call with x2,y2 */
    coefs=imsls_f_logistic_regression(n_observations, n_independent,
        n_classes, x2, y2,
        IMSLS_GROUPS,
        IMSLS_COLUMN_WISE,
        IMSLS_PREV_RESULTS, modell1,
        IMSLS_NEXT_RESULTS, &modell12,
        0);

    imsls_f_write_matrix("Combined Model Coefficients:",
        n_coefs * (n_classes - 1), 1, modell12->coefs,
        0);
    imsls_f_write_matrix("Combined Model Standard Errors:",
        n_coefs * (n_classes - 1), 1, modell12->stderrs,
        0);

    /* combine data */
    for(j = 0; j < n_independent; j++){
        for(i=0; i < n_observations; i++){
            y3[i]=y1[i];
            y3[i + n_observations]=y2[i];
            x3[i + j * 2 * n_observations] = x1[i + j * n_observations];
            x3[i + j * 2 * n_observations + n_observations] =
                x2[i + j * n_observations];
        }
    }

    imsls_free(coefs);

    coefs=imsls_f_logistic_regression(2 * n_observations, n_independent,
        n_classes, x3, y3,
        IMSLS_GROUPS,
        IMSLS_COLUMN_WISE,
        IMSLS_NEXT_RESULTS, &modell3,
        0);

    imsls_f_write_matrix("Full Data Model Coefficients:",
        n_coefs * (n_classes - 1), 1, modell3->coefs,
        0);
    imsls_f_write_matrix("Full Data Model Standard Errors:",
        n_coefs * (n_classes - 1), 1, modell3->stderrs,
        0);
}

```

Output

First Model Coefficients:

1	1.691
2	0.350
3	-0.137
4	1.057
5	-1.254
6	0.242
7	-0.004
8	0.115
9	1.032
10	0.278
11	0.016
12	-1.954

First Model Standard Errors:

1	2.389
2	0.565
3	0.061
4	1.025
5	2.197
6	0.509
7	0.047
8	0.885
9	2.007
10	0.461
11	0.043
12	0.958

Combined Model Coefficients:

1	-1.169
2	0.649
3	-0.038
4	0.608
5	-1.935
6	0.435
7	0.002
8	0.215
9	-0.193
10	0.282
11	0.002
12	-0.630

Combined Model Standard Errors:

1	1.489
2	0.359
3	0.029
4	0.588
5	1.523
6	0.358
7	0.030
8	0.584
9	1.461
10	0.344
11	0.030
12	0.596

Full Data Model Coefficients:

1	-1.009
2	0.640
3	-0.051
4	0.764
5	-2.008
6	0.436
7	0.003
8	0.263
9	-0.413
10	0.299
11	0.004
12	-0.593
Full Data Model Standard Errors:	
1	1.466
2	0.350
3	0.029
4	0.579
5	1.520
6	0.357
7	0.029
8	0.581
9	1.389
10	0.336
11	0.028
12	0.577

Warning Errors

IMSLS_NO_CONV_SEP

Convergence did not occur in # iterations. "tolerance" = #, the error between estimates = #, and the gradient has norm = #. Adjust "tolerance" or "max_iter", or there may be a separation problem in the data.

IMSLS_EMPTY_INT_RESULTS

Intermediate results given to the function are empty and may be expected to be non-empty in this scenario.

Fatal Errors

IMSLS_NO_CONV_OVERFLOW

The linear predictor = # is too large and will lead to overflow when exponentiated. The algorithm fails to converge.

logistic_reg_predict



[more...](#)

Predict a binomial or multinomial outcome given an estimated model and new values of the independent variables.

Synopsis

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

```
float *imsls_f_logistic_reg_predict (int n_observations, int n_independent,  
                                     int n_classes, float coefs[ ], float x[ ], ..., 0)
```

The type *double* function is `imsls_d_logistic_reg_predict`.

Required Arguments

int `n_observations` (Input)

The number of observations.

int `n_independent` (Input)

The number of independent variables.

int `n_classes` (Input)

The number of discrete outcomes, or classes.

float `coefs[]` (Input)

Array of length `n_coefficients × n_classes` containing the coefficient estimates of the logistic regression model. `n_coefficients` is the number of coefficients in the model.

float `x[]` (Input)

Array of length `n_observations × n_independent` containing the values of the independent variables.

Return Value

Pointer to an array containing the predicted responses. The predicted value is the predicted number of outcomes in each class for each new observation provided in `x`. If `frequencies[i] = 1` for all observations, then the return value is equivalent to the predicted probabilities. If the option `IMSLS_CONFIDENCE` is specified, the length of the return array is $(n_observations \times n_classes \times 3)$ and the array includes the lower and upper prediction limits. Otherwise, the array is of length $(n_observations \times n_classes)$. Note that if the data is column-oriented (see `IMSLS_COLUMN_WISE`), the return value will also be column-oriented.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_logistic_reg_predict (int n_observations, int n_independent,
    int n_classes, float coefs [], float x [],
    IMSLS_Y, floaty [],
    IMSLS_GROUP_COUNTS, or
    IMSLS_GROUPS,
    IMSLS_COLUMN_WISE,
    IMSLS_FREQUENCIES, int frequencies [],
    IMSLS_REFERENCE_CLASS, int ref_class,
    IMSLS_NO_INTERCEPT,
    IMSLS_X_INDICES, int n_xin, int xin [],
    IMSLS_X_INTERACTIONS, int n_xinteract, int xinteract [],
    IMSLS_CONFIDENCE, float confid,
    IMSLS_MODEL, Imsls_f_model *model,
    IMSLS_PREDERR, float*prederr,
    IMSLS_RETURN_USER, floatyhat [],
    0)
```

Optional Arguments

`IMSLS_Y`, *floaty*[] (Input)

Array containing the actual responses corresponding to the independent variables. If present, the expected length for `y` is `n_observations × n_classes` unless one of `IMSLS_GROUPS` or `IMSLS_GROUP_COUNTS` is also present. `IMSLS_Y` is required when `IMSLS_PREDERR` is requested.

Default: The function expects that `y` is not given.

`IMSLS_GROUP_COUNTS` (Input)

or

`IMSLS_GROUPS`, (Input)

These optional arguments specify alternative formats of the input array `y`. If `IMSLS_GROUP_COUNTS` is present, `y` is of length `n_observations × (n_classes - 1)`, and contains counts for all but one of the classes for each observation. The missing class is treated as the reference class. If `IMSLS_GROUP_COUNTS` is present and if any `y[i] > 1`, `IMSLS_FREQUENCIES` is required. If `IMSLS_GROUPS` is present, the input array `y` is of length `n_observations` and `y[i]` contains the group number to which the observation belongs. In this case, `frequencies[i]` is set to 1 for all observations.

Default: Unless one of the arguments is present, the function expects that `y` is `n_observations × n_classes` and contains counts for all the classes.

`IMSLS_COLUMN_WISE`, (Input)

If present, the input arrays are column-oriented. That is, contiguous elements in `x` are values of the same independent variable, or column, except at multiples of `n_observations`.

Default: Input arrays are row-oriented.

`IMSLS_FREQUENCIES`, *intfrequencies*[] (Input)

Array of length `n_observations` containing the number of replications or trials for each of the observations. This argument is required if `IMSLS_GROUP_COUNTS` is present and if any `y[i] > 1`.

Default: `frequencies[i] = 1`.

`IMSLS_REFERENCE_CLASS`, *intref_class* (Input)

Number specifying which class or outcome category to use as the reference class. The purpose of the reference class is explained in the [Description](#) section.

Default: `ref_class = n_classes`.

`IMSLS_NO_INTERCEPT` (Input)

If present, the model will not include an intercept term.

Default: The intercept term is included.

`IMSLS_X_INDICES`, *int n_xin*, *int xin*[] (Input)

An array of length `n_xin` providing the variable indices of `x` that correspond to the independent variables the user wishes to be included in the logistic regression model.

Default: All `n_independent` variables are included.

`IMSLS_X_INTERACTIONS`, *int n_xinteract*, *int xinteract*[] (Input)

An array of length `n_xinteract` × 2 providing pairs of variable indices of `x` that define the interaction terms in the model. Adjacent indices should be unique.

Default: No interaction terms are included.

`IMSLS_CONFIDENCE`, *float confid* (Input)

This value provides the confidence level to use in the calculation of the prediction intervals. If this argument is present and valid ($0 < \text{confid} < 100$), `confid%` prediction intervals are provided for each predicted value.

Default: Prediction intervals are not provided.

`IMSLS_MODEL`, *lmsls_f_model*model* (Input)

Pointer to a structure of type `lmsls_f_model` containing information about the logistic regression fit. See `lmsls_f_logistic_regression`. Required when `IMSLS_CONFIDENCE` is present.

Default: Not needed if `IMSLS_CONFIDENCE` is not present.

`IMSLS_PREDERR`, *float *prederr* (Output)

The mean squared prediction error when `IMSLS_Y` is present.

`IMSLS_RETURN_USER`, *floatyhat*[] (Output)

Storage for the return value is provided by the user. See the description of the [Return Value](#) above for details.

Description

Function `lmsls_f_logistic_reg_predict` calculates the predicted outcomes for a binomial or multinomial response variable given an estimated logistic regression model and new observations of the independent variables.

For a binary response `y`, the objective is to estimate the conditional probability of success,

$\pi_1(x) = \Pr[y = 1 | x]$, where $x = (x_1, x_2, \dots, x_p)'$ is a realization of p independent variables. In particular, the estimated probability of success

$$\hat{\pi}_1(x) = \frac{\exp(\hat{\eta}_1)}{1 + \exp(\hat{\eta}_1)}$$

where

$$\hat{\eta}_1 = \hat{\beta}_{01} + x^T \hat{\beta}_1$$

and

$\hat{\beta}_{01}, \hat{\beta}_1 = (\hat{\beta}_{11}, \hat{\beta}_{12}, \dots, \hat{\beta}_{1p})'$ are the coefficient estimates. Then $\hat{y} = n_i \pi_1(x_i)$. That is, \hat{y} is the expected value of the response under the estimated model given the values of the independent variables.

Similarly, for a multinomial response, with class K the reference class,

$$\hat{\pi}_k(x) = \frac{\exp(\hat{\eta}_{ik})}{\sum_{l=1}^K \exp(\hat{\eta}_{il})} = \frac{\exp(\hat{\eta}_{ik})}{1 + \sum_{l=1}^{K-1} \exp(\hat{\eta}_{il})}$$

Then

$$\hat{\pi}_K(x) = 1 - \sum_{l=1}^{K-1} \hat{\pi}_l(x)$$

and $\hat{y}_k = n_i \pi_k(x_i)$. If the actual responses are given, the mean squared prediction error is

$$\text{mspe} = \frac{1}{NK} \sum_{k=1}^K \sum_{i=1}^N (\hat{y}_{ik} - y_{ik})^2$$

If requested, $100(1 - \alpha)\%$ prediction intervals are provided for the predicted values by first finding the prediction standard errors of the logits, $\hat{\eta}_{ik} = \hat{\beta}_{0k} + x_i^T \hat{\beta}_k$, and then evaluating

$$\frac{\exp(\hat{\eta}_{ik} \pm z_{\alpha/2} SE(\hat{\eta}_{ik}))}{1 + \sum_{l=1}^{K-1} \exp(\hat{\eta}_{il} \pm z_{\alpha/2} SE(\hat{\eta}_{il}))}$$

to obtain the upper and lower limits for $\hat{\pi}_k(x_i)$, where $z_{\alpha/2}$ is the upper $\alpha/2$ quantile of the standard normal distribution. Note that properties of the prediction intervals are only valid when the new observations are inside the range of the original data used to fit the model. Generally, the model should not be used to extrapolate outside the range of the original data. See Hosmer and Lemeshow (2000) for further details.

Examples

Example 1

The model fit to the beetle mortality data of Prentice (1976) is used to predict the expected mortality at three new doses. For the original data, see [Example 1](#) in `imsls_f_logistic_regression`.

Log Dosage	Number of Beetles Exposed	Number of Deaths
1.66	16	??
1.87	22	??
1.71	11	??

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

int main(){
    float y1[8]={6, 13, 18, 28, 52, 53, 61, 60};
    float x1[8]={1.69, 1.724, 1.755, 1.784, 1.811, 1.836, 1.861, 1.883};
    float x2[3]={1.66, 1.87, 1.71};
    float freqs1[8]={59, 60, 62, 56, 63, 59, 62, 60};
    float freqs2[3]={16, 22, 11};
    float *coefs, *yhat;
    int n_classes=2, n_observations=8, n_independent=1,
        n_coefs=2, i, n_new_observations=3;

    coefs=imsls_f_logistic_regression(n_observations,n_independent,
        n_classes,x1,y1,
        IMSLS_GROUP_COUNTS,
        IMSLS_FREQUENCIES,freqs1,
        0);

    imsls_f_write_matrix("Coefficient Estimates",(n_coefs)*(n_classes-1),
        1,coefs,0);

    yhat=imsls_f_logistic_reg_predict(n_new_observations,n_independent,
        n_classes,coefs,x2,IMSLS_FREQUENCIES,freqs2,0);

    printf( "\nDose\t N\tExpected Deaths\n");
    for(i=0;i<n_new_observations;i++){
        printf( "%5.2f\t%2.1f\t\t%5.2f\n",
            x2[i],freqs2[i],yhat[2*i]);
    }
}
```

Output

```
Coefficient Estimates
1      -60.76
2       34.30
```

Dose	N	Expected Deaths
1.66	16.0	0.34
1.87	22.0	21.28
1.71	11.0	1.19

Example 2

A logistic regression model is fit to artificial (noisy) data with 4 classes and 3 independent variables and used to predict class probabilities at 10 new values of the independent variables. Also shown are the mean squared prediction error and upper and lower limits of the 95% prediction interval for each predicted value.

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

int main(){
    float x[50*3]={
        3, 2, 2, 1, 3, 3, 3, 2, 3, 3, 3, 3, 3, 3, 2, 3, 2, 1, 3, 2,
        2, 1, 2, 1, 3, 2, 1, 2, 1, 2, 3, 2, 1, 2, 1, 1, 2, 3, 1, 2,
        1, 1, 1, 3, 1, 3, 2, 3, 3, 1,
        25.92869, 51.63245, 25.78432, 39.37948, 24.65058, 45.20084,
        52.6796, 44.28342, 40.63523, 51.76094, 26.30368, 20.70230,
        38.74273, 19.47333, 26.42211, 37.05986, 51.67043, 42.40156,
        33.90027, 35.43282, 44.30369, 46.72387, 46.99262, 36.05923,
        36.83197, 61.66257, 25.67714, 39.08567, 48.84341, 39.34391,
        24.73522, 50.55251, 31.34263, 27.15795, 31.72685, 25.00408,
        26.35457, 38.12343, 49.9403, 42.45779, 38.80948, 43.22799,
        41.87624, 48.0782, 43.23673, 39.41294, 23.93346,
        42.8413, 30.40669, 37.77389,
        1, 2, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 2, 1, 1, 2, 2, 1, 2, 1, 1,
        1, 1, 1, 2, 2, 1, 2, 2, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 2, 2, 2,
        1, 1, 2, 1, 1, 2, 1, 1
    };
    float y[50]={
        1, 2, 3, 4, 3, 3, 4, 4, 4, 4, 2, 1, 4, 1, 1, 1, 4, 4, 3, 1, 2,
        3, 3, 4, 2, 3, 4, 1, 2, 4, 3, 4, 4, 1, 3, 4, 4, 2, 3, 4, 2, 2,
        4, 3, 1, 4, 3, 4, 2, 3
    };
    float newx[10*3]={
        2, 2, 1, 3, 3, 2, 3, 3, 3,
        25.92869, 51.63245, 25.78432, 39.37948, 24.65058, 45.20084,
        52.6796, 44.28342, 40.63523, 51.76094,
        1, 2, 1, 1, 1, 1, 2, 2, 2, 1
    };
    float newy[10]={
        3, 2, 1, 1, 4, 3, 2, 2, 1, 2
    };
    float *coefs,*yhat,mspe,model_pval,lrstat;
    int i,j,n_classes,n_observations,n_new_obs,n_independent,n_coefs,dof;
    Imsls_f_model *model_info_ptr=NULL;

    n_classes=4;
    n_observations=50;
    n_new_obs=10;
    n_independent=3;
    n_coefs=4;

    coefs=imsls_f_logistic_regression(n_observations,n_independent,
```

```

    n_classes,x,y,
    IMSLS_GROUPS,
    IMSLS_COLUMN_WISE,
    IMSLS_LRSTAT,&lrstat,
    IMSLS_NEXT_RESULTS,&model_info_ptr,
    0);

yhat=imsls_f_logistic_reg_predict(n_new_obs,n_independent,
    n_classes,coefs,newx,
    IMSLS_Y,newy,
    IMSLS_GROUPS,
    IMSLS_COLUMN_WISE,
    IMSLS_CONFIDENCE,95.0,
    IMSLS_MODEL,model_info_ptr,
    IMSLS_PREDERR,&mspe,
    0);

dof = n_coefs*(n_classes-1) - (n_classes-1);
model_pval = 1.0 -
    imsls_f_chi_squared_cdf(lrstat,dof);
printf("Model Fit Summary:\n");
printf("Log-likelihood: %5.2f \n",model_info_ptr->loglike);
printf("LR test statistic: %5.2f\n",lrstat);
printf("Degrees of freedom: %d\n", dof);
printf("P-value: %5.4f\n", model_pval);
printf("\nPrediction Summary:\n");
printf("\nMean squared prediction error: %4.2f\n", mspe);
printf("\n%Obs Class Estimate Lower Upper\n");
for(j=0;j<n_new_obs;j++){
    for(i=0;i<n_classes;i++){
        printf(" %d\t%d      %4.2f   %4.2f   %4.2f\n",j+1,i+1,
            yhat[i*3*n_new_obs+j],
            yhat[(i*3+1)*n_new_obs+j],
            yhat[(i*3+2)*n_new_obs+j]);
    }
}
}
}

```

Output

```

Model Fit Summary:
Log-likelihood: -58.58
LR test statistic: 16.37
Degrees of freedom: 9
P-value: 0.0595

Prediction Summary:
Mean squared prediction error: 0.21

Obs Class Estimate Lower Upper
1 1 0.26 0.20 0.20
1 2 0.14 0.11 0.11
1 3 0.31 0.24 0.24
1 4 0.29 0.45 0.46
2 1 0.04 0.03 0.03
2 2 0.27 0.17 0.17
2 3 0.12 0.08 0.08
2 4 0.57 0.72 0.72
3 1 0.23 0.17 0.17

```

3	2	0.13	0.10	0.10
3	3	0.28	0.21	0.21
3	4	0.36	0.52	0.53
4	1	0.06	0.04	0.05
4	2	0.16	0.13	0.13
4	3	0.49	0.38	0.38
4	4	0.29	0.45	0.45
5	1	0.34	0.28	0.28
5	2	0.13	0.11	0.11
5	3	0.30	0.25	0.25
5	4	0.22	0.36	0.37
6	1	0.03	0.02	0.02
6	2	0.16	0.12	0.12
6	3	0.53	0.41	0.41
6	4	0.29	0.44	0.45
7	1	0.04	0.02	0.02
7	2	0.27	0.17	0.17
7	3	0.13	0.08	0.08
7	4	0.57	0.72	0.73
8	1	0.14	0.09	0.09
8	2	0.29	0.19	0.20
8	3	0.12	0.08	0.08
8	4	0.46	0.63	0.63
9	1	0.21	0.14	0.15
9	2	0.27	0.19	0.19
9	3	0.10	0.07	0.07
9	4	0.42	0.59	0.60
10	1	0.01	0.01	0.01
10	2	0.15	0.12	0.12
10	3	0.57	0.44	0.45
10	4	0.28	0.43	0.44

Warning Errors

IMSL5_NO_ACTUALS

The average squared prediction error cannot be calculated because no actual “y” values are given.

Fatal Errors

IMSL5_OVERFLOW

The linear predictor = # is too large and will lead to overflow when exponentiated.

Nonparametric Statistics

Functions

One sample tests - Nonparametric Statistics

Sign test	sign_test	587
Wilcoxon rank sum test	wilcoxon_sign_rank	591
Noether's test for cyclical trend	noether_cyclical_trend	596
Cox and Stuarts' sign test for trends in location and dispersion	cox_stuart_trends_test	600
Tie statistics	tie_statistics	606

Two or more samples

Wilcoxon's rank sum test	wilcoxon_rank_sum	609
Kruskal-Wallis test	kruskal_wallis_test	615
Friedman's test	friedmans_test	618
Cochran's Q test	cochran_q_test	624
K-sample trends test	k_trends_test	628

Usage Notes

Much of what is considered nonparametric `imsls_f_k_trends_test` 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](#)).

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 `IMSLF_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

`IMSL_S_PERCENTAGE`, *float* `percentage` (Input)

Value in the range (0, 1). Argument `percentile` is the $100 \times \text{percentage}$ percentile of the population.

Default: `percentage` = 0.5

`IMSL_S_PERCENTILE`, *float* `percentile` (Input)

Hypothesized percentile of the population from which `x` was drawn.

Default: `percentile` = 0.0

`IMSL_S_N_POSITIVE_DEVIATIONS`, *int* *`n_positive_deviations` (Output)

Number of positive differences $x[j - 1] - \text{percentile}$ for $j = 1, 2, \dots, n_{\text{observations}}$.

`IMSL_S_N_ZERO_DEVIATIONS`, *int* *`n_zero_deviations` (Output)

Number of zero differences (ties) $x[j - 1] - \text{percentile}$ for $j = 1, 2, \dots, n_{\text{observations}}$.

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 $100p$ -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:

- $H_0: Pr(\mathbf{x} \leq q) \geq p$ (the p -th quantile is at least q)
 $H_1: Pr(\mathbf{x} \leq q) < p$
 Reject H_0 if *probability* is less than or equal to the significance level
- $H_0: Pr(\mathbf{x} \leq q) \leq p$ (the p -th quantile is at least q)
 $H_1: Pr(\mathbf{x} \leq q) > p$
 Reject H_0 if *probability* is greater than or equal to 1 minus the significance level

- $H_0: Pr(\mathbf{x} = q) = p$ (the p -th quantile is q)
 $H_1: Pr(\mathbf{x} \leq q) < p$ or $Pr(\mathbf{x} \leq q) > p$
 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 $\mathbf{x}_i = \mathbf{y}_i - \mathbf{z}_i$ 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>

int 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>
```

```
#include <stdio.h>

int 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,
```

```

IMSL_STAT_USER, float stat[ ],
IMSL_N_MISSING, float *n_missing,
IMSL_RETURN_USER, float prob[ ],
0)

```

Optional Arguments

`IMSL_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.

`IMSL_STAT`, *float **stat* (Output)

Address of a pointer to an internally allocated array of length 10 containing the following statistics:

	Statistics
0	The positive rank sum, W_+ , using method 1.
1	The absolute value of the negative rank sum, W_- , using method 1.
2	The standardized (to an asymptotic variance of 1.0) minimum of (W_+ , W_-) using method 1.
3	The asymptotic probability of not exceeding <code>stat[2]</code> 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 <code>stat[6]</code> 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 <code>fuzz</code> of zero.

`IMSL_STAT_USER`, *float stat[]* (Output)

Storage for array `stat` is provided by the user.

See `IMSL_STAT`.

`IMSL_N_MISSING`, *float *n_missing*, (Output)

Number of missing values in `y`.

`IMSL_S_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:

- $H_0 : M \leq 0$ $H_1 : M > 0$
Reject if `stat[0]` [or `stat[4]`] is too large.
- $H_0 : M \geq 0$ $H_1 : M < 0$
Reject if `stat[1]` [or `stat[5]`] is too large.
- $H_0 : M = 0$ $H_1 : M \neq 0$
Reject if `stat[2]` [or `stat[6]`] is too small. Alternatively, if an asymptotic test is desired, reject if $2 * \text{stat}[3]$ [or $2 * \text{stat}[7]$] 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) $\{X_1 = 223, 216, 211, 212, 209, 205, 201; \text{ and } X_2 = 208, 205, 202, 207, 206, 204, 203\}$. 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>
int main()
{
    float *stat=NULL, *result=NULL;
    int nob = 7, nmiss;
    float fuzz = .0001;
    float x[] = {-25., -21., -19., -15., -13., -11., -8.};
    result = imsls_f_wilcoxon_sign_rank(nob, x,
                                      IMSLS_N_MISSING, &nmiss,
                                      IMSLS_FUZZ, fuzz,
                                      IMSLS_STAT, &stat,
                                      0);

    printf("Statistic\t\t\tMethod 1\tMethod 2\n");
    printf("W+\t\t\t\t %3.0f\t\t %3.0f\n", stat[0], stat[4]);
    printf("W-\t\t\t\t %3.0f\t\t %3.0f\n", stat[1], stat[5]);
    printf("Standardized Minimum\t\t%6.4f\t\t%6.4f\n", stat[2], stat[6]);
    printf("p-value\t\t\t\t %6.4f\t\t %6.4f\n\n", stat[3], stat[7]);
    printf("Number of zeros\t\t\t%3.0f\n", stat[8]);
    printf("Number of ties\t\t\t%3.0f\n", stat[9]);
    printf("Number of missing\t\t %d\n", nmiss);
}
```

Output

```
*** WARNING ERROR 4 from imsls_f_wilcoxon_sign_rank. NOBS = 7. The number
*** of observations, NOBS, is less than 50, and exact
*** tables should be referenced for probabilities.

Statistic                Method 1    Method 2
W+.....                0          0
W-.....                28         28
Standardized Minimum.... -2.3664    -2.3664
p-value.....            0.0090     0.0090
```

Number of zeros.....	0
Number of ties.....	0
Number of missing.....	0

noether_cyclical_trend

Performs the Noether test for cyclical trend.

Synopsis

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

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

The type *double* function is `imsls_d_noether_cyclical_trend`.

Required Arguments

int `n_observations` (Input)

Number of observations in `x`. `n_observations` must be greater than or equal to 3.

float `x[]` (Input)

Array of length `n_observations` containing the data in chronological order.

Return Value

Array, `p`, of length 3 containing the probabilities of `stat[1]` or more, `stat[2]` or more, or `stat[3]` or more monotonic sequences.

If `stat[0]` is less than 1, `p[0]` is set to NaN (not a number).

Synopsis with Optional Arguments

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

```
float *imsls_f_noether_cyclical_trend(int n_observations, float x[],
```

```
    IMSLS_FUZZ, float fuzz,
```

```
    IMSLS_STAT, int **stat,
```

```
    IMSLS_STAT_USER, int stat[],
```

```
    IMSLS_N_MISSING, int *n_missing,
```

IMSLS_RETURN_USER, *float* p[],

0)

Optional Arguments

IMSLS_FUZZ, *float* fuzz (Input)

Nonnegative constant used to determine ties in computing ranks in the combined samples. A tie is declared when two observations in the combined sample are within fuzz of each other.

Default value for `fuzz` is 0.0.

IMSLS_STAT, *int ***stat (Output)

Address of a pointer to an internally allocated array of length 6 containing the following statistics:

Row	Statistics
stat[0]	The number of consecutive sequences of length three used to detect cyclical trend when tying middle elements are eliminated from the sequence, and the next consecutive observation is used.
stat[1]	The number of monotonic sequences of length three in the set defined by stat[0].
stat[2]	The number of nonmonotonic sequences where tied threesomes are counted as nonmonotonic.
stat[3]	The number of monotonic sequences where tied threesomes are counted as monotonic.
stat[4]	The number of middle observations eliminated because they were tied in forming the stat[0] sequences.
stat[5]	The number of tied sequences found in forming the stat[2] and stat[3] sequences. A sequence is called a tied sequence if the middle element is tied with either of the two other elements.

IMSLS_STAT_USER, *int* stat[] (Output)

Storage for array stat is provided by the user.

See IMSLS_STAT.

IMSLS_N_MISSING, *int *n_missing* (Output)

Number of missing values in X.

IMSLS_RETURN_USER, *float* p[] (Input)

User allocated array of length 3 containing the return values.

Description

Routine `imsls_f_noether_cyclical_trend` performs the Noether test for cyclical trend (Noether 1956) for a sequence of measurements. In this test, the observations are first divided into sets of three consecutive observations. Each set is then inspected, and if the set is monotonically increasing or decreasing, the count variable is incremented.

The count variables, `stat[1]`, `stat[2]`, and `stat[3]`, differ in the manner in which ties are handled. A tie can occur in a set (of size three) only if the middle element is tied with either of the two ending elements. Tied ending elements are not considered. In `stat[1]`, tied middle observations are eliminated, and a new set of size 3 is obtained by using the next observation in the sample. In `stat[2]`, the original set of size three is used, and tied middle observations are counted as nonmonotonic. In `stat[3]`, tied middle observations are counted as monotonic.

The probabilities of occurrence of the counts are obtained from the binomial distribution with $p = 1/3$, where p is the probability that a random sample of size three from a continuous distribution is monotonic. The binomial sample size is, of course, the number of sequences of size three found (adjusted for ties).

Hypothesis test:

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

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>
int 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.**

	<code>pstat[I]</code>
0	Probability of <code>nstat[0] + nstat[2]</code> or more negative signs (ties are considered negative).
1	Probability of obtaining <code>nstat[1]</code> or more positive signs (ties are considered negative).
2	Probability of <code>nstat[0] + nstat[2]</code> or more negative signs (ties are considered positive).
3	Probability of obtaining <code>nstat[1]</code> or more positive signs (ties are considered positive).

The last four elements of `pstat` are computed from three groups of observations.

4	Probability of <code>nstat[0] + nstat[2]</code> or more negative signs (ties are considered negative).
5	Probability of obtaining <code>nstat[1]</code> or more positive signs (ties are considered negative).
6	Probability of <code>nstat[0] + nstat[2]</code> or more negative signs (ties are considered positive).
7	Probability of obtaining <code>nstat[1]</code> 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 supplied, 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:

<code>i</code>	<code>nstat[i]</code>
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 <code>pstat[0]</code> through <code>pstat[3]</code> (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 <code>pstat[4]</code> through <code>pstat[7]</code> (three groups).

`IMSLS_STAT_USER`, `int 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 value 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 `IMSL_DISPERSION` is supplied) proceed exactly as with the tests for location, but using one of two derived dispersion measures. The input value `k` is used to define `n_observations/k` groups of consecutive observations starting with observation 1. The first `k` observations define the first group, the next `k` observations define the second group, etc., with the last observations omitted if `n_observations` is not evenly divisible by `k`. A dispersion score is then computed for each group as either the range (`ids = 0`), or a multiple of the variance (`ids ≠ 0`) of the observations in the group. The dispersion scores form a derived sample. The tests proceed on the derived sample as above.

Ties

Ties are defined as occurring when a group one observation is within `fuzz` of its last group counterpart. Ties imply that the probability distribution of \mathbf{x} is not strictly continuous, which means that $\Pr(\mathbf{x}_1 > \mathbf{x}_2) \neq 0.5$ 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>
int 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_observations, 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.

$$ties[0] = \sum_{j=1}^{\tau} [t_j(t_j - 1)] / 2$$

$$ties[1] = \sum_{j=1}^{\tau} [t_j(t_j - 1)(t_j + 1)] / 12$$

$$ties[2] = \sum_{j=1}^{\tau} t_j(t_j - 1)(2t_j + 5)$$

$$ties[3] = \sum_{j=1}^{\tau} t_j(t_j - 1)(t_j - 2)$$

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_observations, 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 <stdlib.h>
```

```
int 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", 1, 4, ties,
        IMSLS_WRITE_FORMAT, "%5.2f",
        0);
}
```

Output

TIES	1	2	3
4.00	2.50	84.00	6.00

wilcoxon_rank_sum

Performs a Wilcoxon rank sum test for comparing the medians of two populations.

Synopsis

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

```
float imsls_f_wilcoxon_rank_sum (int nx, float x[ ], int ny, float y[ ], ..., 0)
```

The type *double* function is `imsls_d_wilcoxon_rank_sum`.

Required Arguments

int nx (Input)

Number of observations in the first sample.

float x[] (Input)

Array of length *nx* containing the first sample.

int ny (Input)

Number of observations in the second sample.

float y[] (Input)

Array of length *ny* containing the second sample.

Return Value

The two-sided p -value for the Wilcoxon rank sum statistic computed with average ranks used in the case of ties. The p -value is computed using either exact or approximate calculations depending upon the number of observations and the optional argument `IMSLS_EXACT_P_VALUE`.

Synopsis with Optional Arguments

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

```
float imsls_f_wilcoxon_rank_sum (int nx, float x[ ], int ny, float y[ ],  
                                IMSLS_FUZZ, float fuzz,
```

```

IMSLS_N_MISSING_X, int *nmissx,
IMSLS_N_MISSING_Y, int *nmissy,
IMSLS_MANN_WHITNEY, float *mann_whitney,
IMSLS_EXACT_P_VALUES, float **p,
IMSLS_EXACT_P_VALUES_USER, float p[ ],
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: $\text{fuzz} = 100 \times \text{imsls_f_machine}(4) \times \max\{|x_{i1}|, |x_{j2}|\}$

IMSLS_N_MISSING_X, *int *nmissx* (Output)

Pointer to a scalar for the number of missing observations detected in **x**.

IMSLS_N_MISSING_Y, *int *nmissy* (Output)

Pointer to a scalar for the number of missing observations detected in **y**.

IMSLS_MANN_WHITNEY, *float *mann_whitney* (Output)

Pointer to a scalar for the Mann-Whitney test statistic. Although the test statistics for the Mann-Whitney and Wilcoxon rank sum tests are computed differently, the p -values for these tests are equal since the Wilcoxon test statistic is a linear transformation of the Mann-Whitney test statistic.

IMSLS_EXACT_P_VALUES, *float **p* (Output)

Address of a pointer to an internally allocated array of length 3 containing the exact p -values according to the following table:

	p-values
0	The exact left-tailed p -value.
1	The exact right-tailed p -value.
2	The exact two-tailed p -value.

IMSLS_EXACT_P_VALUES_USER, *float* p[] (Output)

Storage for array p is provided by the user. See IMSLS_EXACT_P_VALUES.

IMSLS_STAT, *float **stat* (Output)

Address of a pointer to an internally allocated array of length 10 containing the following statistics:

	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 <i>stat</i> [6] under the null hypothesis of no difference.
8	Standard normal score associated with <i>stat</i> [6].
9	Two-sided p -value associated with <i>stat</i> [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` conducts the Wilcoxon rank sum test for identical population distribution functions. The Wilcoxon test and the Mann-Whitney U test are equivalent. 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 \mathbf{x} 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 uses the average rank of the tied observations for handling tied observations between samples. 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.

The p -value returned in `stat[9]` is the two-sided p -value calculated using the normal approximation with the normal score returned in `stat[8]`. The p -value returned by this routine is either the approximate or exact two-sided p -value depending upon the number of observations and `IMSLS_EXACT_P_VALUES`. The exact two-sided p -value is returned when the optional argument `IMSLS_EXACT_P_VALUES` is used or when both `nx` and `ny` are 25 or less.

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.

	Null and Alternative Hypothesis	Action
1	$H_0:Pr(x < y) = 0.5$ vs $H_1:Pr(x < y) \neq 0.5$ or $H_0:E(x) = E(y)$ vs $H_1:E(x) \neq E(y)$	Reject if <code>p_value</code> is less than the user's significance level of the test.
2	$H_0:Pr(x < y) \leq 0.5$ vs $H_1:Pr(x < y) > 0.5$ or $H_0:E(x) \geq E(y)$ vs $H_1:E(x) < E(y)$	Reject if <code>stat[6]</code> is too small or if <code>p[0]</code> is less than the user's significance level of the test
3	$H_0:Pr(x < y) \geq 0.5$ vs $H_1:Pr(x < y) < 0.5$ or $H_0:E(x) \leq E(y)$ vs $H_1:E(x) > E(y)$	Reject if <code>stat[6]</code> is too large or if <code>p[1]</code> is less than the user's significance level of the test

Assumptions

1. Arguments `x` and `y` 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 -values are calculated using either the large-sample normal approximation or the exact probability calculations. This approximate calculation is usually considered adequate when the size of one or both samples is greater than 50. For smaller samples, the exact probability calculations returned by `IMSLs_EXACT_P_VALUES` are recommended.

Example

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, [Utilities](#)).

The statistics are output in the array `stat`.

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

int main()
{
    int    nx = 5;
    int    ny = 5;
    float x[5] = {7.3, 6.9, 7.2, 7.8, 7.2};
    float y[5] = {7.4, 6.8, 6.9, 6.7, 7.1};
    float *stat, *p;
    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)",
        "Approximate Two-sided p-value of W ....."
    };

    imsls_f_wilcoxon_rank_sum(nx, x, ny, y,
        IMSLS_EXACT_P_VALUES, &p,
        IMSLS_STAT, &stat,
        0);

    imsls_f_write_matrix("statistics", 10, 1, stat,
        IMSLS_ROW_LABELS, labels,
```

```

        IMSLS_WRITE_FORMAT, "%7.3f",
        0);

    printf("Exact Left-Tailed p-value ..... %8.3f\n", p[0]);
    printf("Exact Right-Tailed p-value ..... %8.3f\n", p[1]);
    printf("Exact Two-sided p-value ..... %8.3f\n", p[2]);
}
    
```

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
Approximate Two-sided p-value of W ..... 0.141
Exact Left-Tailed p-value ..... 0.937
Exact Right-Tailed p-value ..... 0.079
Exact Two-sided p-value ..... 0.159
    
```

Warning Errors

IMSLS_AT_LEAST_ONE_TIE At least one tie is detected between the samples.

Fatal Errors

IMSLS_ALL_X_Y_MISSING Each element of x and/or y 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	<code>stat[i]</code>
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 \mathbf{y} . If (after sorting) $|\mathbf{y}[i] - \mathbf{y}[i + 1]|$ 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>

int 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 (n_blocks, int n_treatments, float y[ ],
    IMSLS_FUZZ, fuzz,
    IMSLS_ALPHA, float alpha,
```

```

IMSL5_STAT, float **stat,

IMSL5_STAT_USER, float stat[ ],

IMSL5_SUM_RANK, float **sum_ranks,

IMSL5_SUM_RANK_USER, float sum_rank[ ],

IMSL5_DIFFERENCE, float *difference,

0)

```

Optional Arguments

IMSL5_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.

IMSL5_ALPHA, *float alpha* (Input)

Critical level for multiple comparisons. *alpha* should be between 0 and 1 exclusive.

Default value is 0.05.

IMSL5_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>i</i>	stat[<i>i</i>]
0	Friedman two-sided test statistic.
1	Approximate <i>F</i> value for <i>stat</i> [0].
2	Page test statistic for testing the ordered alternative that the median of treatment <i>i</i> is less than or equal to the median of treatment <i>i + 1</i> , with strict inequality holding for some <i>i</i> .
3	Asymptotic <i>p</i> -value for <i>stat</i> [0]. Chi-squared approximation.
4	Asymptotic <i>p</i> -value for <i>stat</i> [1]. <i>F</i> approximation.
5	Asymptotic <i>p</i> -value for <i>stat</i> [2]. Normal approximation.

IMSL5_STAT_USER, *float stat[]* (Output)

Storage for array *stat* is provided by the user. See **IMSL5_STAT**.

`IMSLS_SUM_RANK`, *float **sum_rank* (Output)

Address of a pointer to an array of length `n_treatments` containing the sum of the ranks of each treatment.

`IMSLS_SUM_RANK_USER`, *float sum_rank[]* (Output)

Storage for array `sum_rank` is provided by the user.

See `IMSLS_SUM_RANK`.

`IMSLS_DIFFERENCE`, *float *difference* (Output)

Minimum absolute difference in two elements of `sum_rank` to infer at the alpha level of significance that the medians of the corresponding treatments are different.

Description

Function `imsls_f_friedmans_test` may be used to test the hypothesis of equality of treatment effects within each block in a randomized block design. No missing values are allowed. Ties are handled by using the average ranks. The test statistic is the nonparametric analogue of an analysis of variance F test statistic.

The test proceeds by first ranking the observations within each block. Let A denote the sum of the squared ranks, i.e., let

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

where $\text{Rank}(Y_{ij})$ is the rank of the i -th observation within the j -th block, $b = \text{n_blocks}$ is the number of blocks, and $k = \text{n_treatments}$ is the number of treatments. Let

$$B = \frac{1}{b} \sum_{i=1}^k R_i^2$$

where

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

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

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

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 $\alpha = \text{alpha}$ 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

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

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>
#include <stdio.h>
int 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 %4.2f %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
p value for Friedman's T = 0.040566

Friedman's T = ..... 8.28
Friedman's F = ..... 4.93
Page Test = ..... 111.00
Prob Friedman's T = ..... 0.04057
Prob Friedman's F = ..... 0.01859
Prob Page Test = ..... 0.98495
```

```
Sum of Ranks = ..... 16.00 17.00 7.00 10.00
difference = ..... 6.656388
```

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.

cochrans_q_test

Performs a Cochran Q test for related observations.

Synopsis

```
#include <imspls.h>
```

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

The type *double* function is `impls_d_cochrans_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` \times `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 <imspls.h>
```

```
float impls_f_cochrans_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 or 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 : $p_{i1} = p_{i2} = \dots = p_{ic}$ 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 Cochran'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 $5 \times n_variables$ 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>
#include <stdio.h>

int 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, 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);
}
```

Output

```
pq = 0.00024
```

Warning Errors

IMSLS_ALL_0_OR_1

"x" consists of either all ones or all zeros. "α" 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	<code>stat[i]</code>
0	Test statistic (ties are randomized).
1	Conservative test statistic with ties counted in favor of the null hypothesis.
2	<i>p</i> -value associated with <code>stat[0]</code> .
3	<i>p</i> -value associated with <code>stat[1]</code> .
4	Continuity corrected <code>stat[2]</code> .
5	Continuity corrected <code>stat[3]</code> .

i	stat[i]
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 <i>t</i> -statistic associated with stat [2].
13	The <i>t</i> -statistic associated with stat[3].
14	The <i>t</i> -statistic associated with stat [4].
15	The <i>t</i> -statistic associated with stat[5].
16	Degrees of freedom for each <i>t</i> -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 $F_1(\mathbf{x}) < F_2(\mathbf{x}) < \dots < F_k(\mathbf{x})$, where F_1, F_2 , etc., are cumulative distribution functions, and the operator $<$ implies that the less than relationship holds for all values of \mathbf{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

$$M^{km} = (m_{ij}^{km}) = \begin{pmatrix} 2 \text{ if } X_{ki} < X_{mj} \\ 0 \text{ otherwise} \end{pmatrix}$$

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 $n_i = \mathbf{ni} [i]$. 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.

- The samples are independent.

Hypothesis tests

$$H_0: F_1(\mathbf{X}) \geq F_2(\mathbf{X}) \geq \dots \geq F_k(\mathbf{X})$$

$$H_1: F_1(\mathbf{X}) < F_2(\mathbf{X}) < \dots < F_k(\mathbf{X})$$

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>

int 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 associated w/stat[3] ...",
        "stat[14]- t-statistic associated w/stat[4] ...",
        "stat[15]- t-statistic associated 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

Tests of Goodness of Fit

Functions

General Goodness-of-fit tests

Chi-squared goodness-of-fit test	chi_squared_test	635
Conducts the Shapiro-Wilk test for normality.	shapiro_wilk_normality_test	648
Performs a Lilliefors test for normality	lilliefors_normality_test	651
Performs a chi-squared test for normality	chi_squared_normality_test	654
One-sample continuous data Kolmogorov-Smirnov	kolmogorov_one	657
Two-sample continuous data Kolmogorov-Smirnov	kolmogorov_two	661
Mardia's test for multivariate normality	multivar_normality_test	665
Anderson-Darling test for normality	ad_normality_test	670
Cramer-Von Mises test for normality	cvm_normality_test	673

Tests for Randomness

Runs test, Paris-serial test, d2 test or triplets tests	randomness_test	676
---	---------------------------------	-----

Usage Notes

The functions 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 is one function (Lilliefors) that can be used to test specifically for exponential distributions and five functions (Shapiro-Wilk, Lilliefors, Mardia, Anderson-Darling, and Cramer-von Mises) that can be used to test specifically for normal distributions.

When the sample size is less than 5,000 observations, the Shapiro-Wilk test provides an accurate estimate for the p -value of this test. Lilliefors test is also popular but it only provides accurate p -value estimates for values between 0.01 and 0.1. Values below 0.01 are always returned as 0.01, and values above 0.1 are returned as 0.5. The general version of the chi-squared test is also available for the normal distribution.

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 functions 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, chi-squared, Anderson-Darling, and Cramer-von Mises goodness-of-fit test functions allow for missing values (NaN, not a number) in the input data. The functions 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_IDO, int ido,  
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_IDO`, *int* `ido` (Input)

Processing option. The argument `ido` must be one of 0, 1, 2, or 3. If `ido` = 0 (the default), all of the observations are input during one invocation. If `ido` = 1, 2, or 3, blocks of rows of the data can be

processed sequentially in separate invocations of `imsls_f_chi_squared_test`; with this option, it is not a requirement that all observations be memory resident, thus enabling one to handle large data sets.

<code>ido</code>	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 <code>n_observations</code> observations of <code>x</code> will be performed.
2	This is an intermediate invocation; updating for the <code>n_observations</code> observations of <code>x</code> will be performed.
3	This is the final invocation of this function. Updating for the data in <code>x</code> and wrap-up computations are performed. Workspace is released. No further invocations of <code>imsls_f_chi_squared_test</code> with <code>ido</code> greater than 1 should be made without first invoking <code>imsls_f_chi_squared_test</code> with <code>ido = 1</code> .

Default: `ido = 0`

`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 [Passing Data to User-Supplied Functions](#) section at the beginning of this manual for more details.

Description

Function `imsls_f_chi_squared_test` performs a chi-squared goodness-of-fit test that a random sample of observations is distributed according to a specified theoretical cumulative distribution. The theoretical distribution, which can be continuous, discrete, or a mixture of discrete and continuous distributions, is specified by the user-defined function `user_proc_cdf`. Because the user is allowed to give a range for the observations, a test that is conditional on the specified range is performed.

Argument `n_categories` gives the number of intervals into which the observations are to be divided. By default, equiprobable intervals are computed by `imsls_f_chi_squared_test`, but intervals that are not equiprobable can be specified through the use of optional argument `IMSLS_CUTPOINTS`.

Regardless of the method used to obtain the cutpoints, the intervals are such that the lower endpoint is not included in the interval, while the upper endpoint is always included. If the cumulative distribution function has discrete elements, then user-provided cutpoints should always be used since `imsls_f_chi_squared_test` cannot determine the discrete elements in discrete distributions.

By default, the lower and upper endpoints of the first and last intervals are $-\infty$ and $+\infty$, respectively. If `IMSLS_BOUNDS` is specified, the endpoints are user-defined by the two arguments `lower_bound` and `upper_bound`.

A tally of counts is maintained for the observations in x as follows:

- If the cutpoints are specified by the user, the tally is made in the interval to which x_i belongs, using the user-specified endpoints.
- If the cutpoints are determined by `imsls_f_chi_squared_test`, then the cumulative probability at x_i , $F(x_i)$, is computed by the function `user_proc_cdf`.

The tally for x_i is made in interval number $\lfloor mF(x_i) + 1 \rfloor$, where $m = \text{n_categories}$ and $\lfloor \cdot \rfloor$ is the function that takes the greatest integer that is no larger than the argument of the function. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred to reduce the total computing time.

If the expected count in any cell is less than 1, then the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.

Programming Notes

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.

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), as the hypothesized distribution function. In this example, the null hypothesis is not rejected.

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

#define SEED                123457
#define N_CATEGORIES        10
#define N_OBSERVATIONS     1000

int 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

int main()
{
    float      *cell_counts, *cutpoints, *cell_chi_squared;
    float      chi_squared_statistics[3], *x;
    char      *stat_row_labels[] = {"chi-squared",
                                    "degrees of freedom", "p-value"};

    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 $\theta = 5.0$ 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);

int main()
{
    int                i, *poisson;
    float                cell_statistics[3][N_CATEGORIES];
    float                chi_squared_statistics[3], x[N_NUMBERS];
    float                cutpoints[] = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5,
                                        7.5, 8.5, 9.5};
    char                *cell_row_labels[] = {"count", "expected count",
                                             "cell chi-squared"};
    char                *cell_col_labels[] = {"Poisson value", "0", "1", "2",
                                             "3", "4", "5", "6", "7",
                                             "8", "9"};
    char                *stat_row_labels[] = {"chi-squared",
                                             "degrees of freedom", "p-value"};

    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

Example 4

Continuing with [Example 1](#) data, the example below invokes the `imsls_f_chi_squared_test` function using values of `ido` greater than 0. Also, optional arguments are used for the data.

```

#include <imsls.h>

#define SEED                123457
#define N_CATEGORIES        10
#define N_OBSERVATIONS      1000
#define N_OBSERVATIONS_BLOCK_1  300
#define N_OBSERVATIONS_BLOCK_2  300
#define N_OBSERVATIONS_BLOCK_3  400

int main()
{
    float      *cell_counts, *cutpoints, *cell_chi_squared;
    float      chi_squared_statistics[3], *x;
    char      *stat_row_labels[] = {"chi-squared",
                                     "degrees of freedom", "p-value"};

    float  lv_x_block_1[N_OBSERVATIONS_BLOCK_1];
    float  lv_x_block_2[N_OBSERVATIONS_BLOCK_2];
    float  lv_x_block_3[N_OBSERVATIONS_BLOCK_3];

    int i;
    imsls_random_seed_set(SEED);
        /* Generate normal deviates */
    x = imsls_f_random_normal (N_OBSERVATIONS, 0);

    for(i=0; i<N_OBSERVATIONS_BLOCK_1; i++)
        lv_x_block_1[i]=x[i];

    for(i=0; i<N_OBSERVATIONS_BLOCK_2; i++)
        lv_x_block_2[i]=x[N_OBSERVATIONS_BLOCK_1+i];

    for(i=0; i<N_OBSERVATIONS_BLOCK_3; i++)
        lv_x_block_3[i]=x[N_OBSERVATIONS_BLOCK_1+N_OBSERVATIONS_BLOCK_2+i];

        /* Perform chi squared test */
    chi_squared_statistics[2] = imsls_f_chi_squared_test
        (imsls_f_normal_cdf,
         N_OBSERVATIONS_BLOCK_1, N_CATEGORIES, lv_x_block_1,
         IMSLS_IDO, 1,
         IMSLS_CUTPOINTS, &cutpoints,
         IMSLS_CHI_SQUARED, &chi_squared_statistics[0],
         IMSLS_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
         IMSLS_CELL_COUNTS, &cell_counts,
         IMSLS_CELL_CHI_SQUARED, &cell_chi_squared,
         0);
    if (cutpoints) imsls_free (cutpoints);
    if (cell_counts) imsls_free (cell_counts);
    if (cell_chi_squared) imsls_free (cell_chi_squared);

    chi_squared_statistics[2] = imsls_f_chi_squared_test
        (imsls_f_normal_cdf,
         N_OBSERVATIONS_BLOCK_2, N_CATEGORIES, lv_x_block_2,
         IMSLS_IDO, 2,
         IMSLS_CUTPOINTS, &cutpoints,
         IMSLS_CHI_SQUARED, &chi_squared_statistics[0],
         IMSLS_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
         IMSLS_CELL_COUNTS, &cell_counts,
         IMSLS_CELL_CHI_SQUARED, &cell_chi_squared,
         0);

```

```

if (cutpoints) imsls_free (cutpoints);
if (cell_counts) imsls_free (cell_counts);
if (cell_chi_squared) imsls_free (cell_chi_squared);

chi_squared_statistics[2] = imsls_f_chi_squared_test
  (imsls_f_normal_cdf,
   N_OBSERVATIONS_BLOCK_3, N_CATEGORIES, lv_x_block_3,
   IMSLS_IDO, 3,
   IMSLS_CUTPOINTS, &cutpoints,
   IMSLS_CHI_SQUARED, &chi_squared_statistics[0],
   IMSLS_DEGREES_OF_FREEDOM, &chi_squared_statistics[1],
   IMSLS_CELL_COUNTS, &cell_counts,
   IMSLS_CELL_CHI_SQUARED, &cell_chi_squared,
   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);

if (cutpoints) imsls_free (cutpoints);
if (cell_counts) imsls_free (cell_counts);
if (cell_chi_squared) imsls_free (cell_chi_squared);

}

```

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		

Warning Errors

IMSL5_EXPECTED_VAL_LESS_THAN_1	An expected value is less than 1.
IMSL5_EXPECTED_VAL_LESS_THAN_5	An expected value is less than 5.
IMSL5_X_VALUE_OUT_OF_RANGE	Row x contains a value which is out of range.
IMSL5_MISSING_DATA_ELEMENT	At least one data element is missing.

Fatal Errors

IMSL5_ALL_OBSERVATIONS_MISSING	All observations contain missing values.
IMSL5_INCORRECT_CDF_1	Function <code>user_proc_cdf</code> 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.
IMSL5_INCORRECT_CDF_2	Function <code>user_proc_cdf</code> is not a cumulative distribution function. The probability of the range of the distribution is not positive.
IMSL5_INCORRECT_CDF_3	Function <code>user_proc_cdf</code> 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.
IMSL5_INCORRECT_CDF_4	Function <code>user_proc_cdf</code> is not a cumulative distribution function. Its evaluation at a cutpoint is inconsistent with either the evaluation at the lower or upper bound.
IMSL5_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.
IMSL5_TOO_MANY_CELL_DELETIONS	There are more observations deleted from the cell than added.
IMSL5_NO_BOUND_AFTER_100_TRYS	After 100 attempts, a bound for the inverse cannot be determined. Try again with a different initial estimate.
IMSL5_NO_UNIQUE_INVERSE_EXISTS	No unique inverse exists.
IMSL5_CONVERGENCE_ASSUMED	Over 100 iterations have occurred without convergence. Convergence is assumed.

IMSL_BAD_IDO_6	"ido" = #. Initial allocations must be performed by invoking the function with "ido" = 1.
IMSL_BAD_IDO_7	"ido" = #. A new analysis may not begin until the previous analysis is terminated by invoking the function with "ido" = 3.
IMSL_BAD_N_CATEGORIES	"n_categories" = #. The number of categories variable, "n_categories", must be the same in separate function calls.
IMSL_STOP_USER_FCN	Request from user supplied function to stop algorithm. User flag = "#".

shapiro_wilk_normality_test

Performs the Shapiro-Wilk test for normality.

Synopsis

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

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

The type *double* function is `imsls_d_shapiro_wilk_normality_test`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `x[]` (Input)
Array of size `n_observations` containing the observations.

Return Value

The *p*-value for the Shapiro-Wilk test for normality is returned.

Synopsis with Optional Arguments

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

```
float imsls_f_shapiro_wilk_normality_test (int n_observations, float x[ ],  
IMSLS_SHAPIRO_WILK_W, float *shapiro_wilk_w,  
0)
```

Optional Arguments

`IMSLS_SHAPIRO_WILK_W, float *shapiro_wilk_w` (Output)
A pointer to a scalar for the Shapiro-Wilk *W* test statistic.

Description

The Shapiro-Wilk test for normality 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; and 1991). The minimum sample size is 3 and sample sizes as large as 5000 have been validated. In the Shapiro and Wilk test, W is given by

$$W = \left(\sum a_i x_{(i)} \right)^2 / \left(\sum (x_i - \bar{x})^2 \right)$$

where $x_{(i)}$ is the i -th largest order statistic and \bar{x} is the sample mean. Royston (1982 and 1991) gives approximations and tabled values that can be used to compute the coefficients a_i , $i = 1, \dots, n$, and obtains the significance level of the W statistic.

Example

This 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>
#include <stdio.h>

int 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_shapiro_wilk_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 =      0.3473
```

Shapiro Wilk W statistic = 0.9744

`lilliefors_normality_test`

Performs a Lilliefors test for normality.

Synopsis

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

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

The type *double* function is `imsls_d_lilliefors_normality_test`.

Required Arguments

`int n_observations` (Input)

Number of observations. Argument `n_observations` must be greater than 4.

`float x[]` (Input)

Array of size `n_observations` containing the observations.

Return Value

The *p*-value for the Lilliefors test for normality. 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.

Synopsis with Optional Arguments

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

```
float imsls_f_lilliefors_normality_test (int n_observations, float x[ ],  
    IMSLS_MAX_DIFFERENCE, float *max_difference,  
    0)
```

Optional Arguments

`IMSL_MAX_DIFFERENCE`, *float* *`max_difference` (Output)

The maximum absolute difference between the empirical and the theoretical distributions is returned in `max_difference`.

Description

This function computes Lilliefors test and its p -value 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 -value is 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, the Lilliefors test by convention calls for rejection and the p -value is set to 0.01. If the computed probability of a greater D is greater than 0.1, by convention the null hypothesis is accepted and the p -value is set to 0.50. Note that because parameters are estimated, p -value in Lilliefors test is not the same as in the Kolmogorov-Smirnov Test.

Observations from a normal distribution 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).

Example

The data are the head circumference measurements for 50 male infants. The Lilliefors test fails to reject the null hypothesis of normality, i.e., `p_value` is greater than 0.1.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int n_observations = 50;
    float x[] = {37.715, 37.89, 37.538, 35.828, 35.039,
                34.005, 35.766, 35.337, 37.529, 35.857,
                37.827, 35.083, 35.235, 36.782, 35.946,
                35.2, 38.995, 36.889, 35.932, 35.835,
                38.323, 35.624, 30.925, 37.69, 33.759,
                36.697, 39.222, 37.191, 34.814, 36.775,
                35.751, 33.163, 35.205, 32.805, 32.517,
                37.516, 33.654, 37.382, 36.83, 33.465,
                33.613, 35.211, 34.932, 30.645, 35.063,
                34.604, 34.666, 33.789, 34.678, 35.123};
    float p_value, max_diff;
    p_value = imsls_f_lilliefors_normality_test (n_observations, x,
        IMSLS_MAX_DIFFERENCE, &max_diff, 0);
    printf ("p-value = %11.4f\n", p_value);
    printf ("max difference = %f \n", max_diff);
}
```

Output

```
p-value =      0.5000
max difference = 0.085558
```

Warning Errors

IMSLS_TWO_OR_MORE_TIED

Two or more elements in "x" are tied.

Fatal Errors

IMSLS_NEED_AT_LEAST_5

All but # elements of "x" are missing. At least five non-missing 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 non-missing observations are tied.

chi_squared_normality_test

Performs a chi-squared test for normality.

Synopsis

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

```
float imsls_f_chi_squared_normality_test (int n_categories, int n_observations,  
float x[ ], ..., 0)
```

The type *double* function is `imsls_d_chi_squared_normality_test`.

Required Arguments

int `n_categories` (Input)

Number of cells into which the observations are to be tallied. `n_categories` must be at least 2.

int `n_observations` (Input)

Number of observations.

float `x[]` (Input)

Array of size `n_observations` containing the observations.

Return Value

The *p*-value for the chi-squared test for normality. An approximate probability is computed.

Synopsis with Optional Arguments

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

```
float imsls_f_chi_squared_normality_test (int n_categories, int n_observations,  
float x[ ],
```

```
      IMSLS_CHI_SQUARED, float *chi_squared,
```

```
      IMSLS_DEGREES_OF_FREEDOM, float *df,
```

```
      0)
```

Optional Arguments

`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`.

Description

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.

Example

This example is taken from Conover (1980, pp. 195, 364). The data consists of 50 two-digit numbers taken from a telephone book. Since `p_value` is greater than 0.1 the chi-squared test fails to reject the null hypothesis of normality.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int    n_observations = 50;
    int    n_categories = 6;
    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, df, chi_squared;

    p_value = imsls_f_chi_squared_normality_test( n_categories,
        n_observations, x,
        IMSLS_DEGREES_OF_FREEDOM, &df,
        IMSLS_CHI_SQUARED, &chi_squared,
        0);

    printf ("p-value = %11.4f\n", p_value);
    printf ("degrees of freedom = %11.4f\n", df);
    printf ("chi squared test = %11.4f\n", chi_squared);
}
```

```
}
```

Output

p-value =	0.4208
degrees of freedom =	5.0000
chi squared test =	4.9600

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 $CDF(\mathbf{x})$, where \mathbf{x} is the value at which `cdf` is to be evaluated (Input) and `cdf` is the value of CDF at \mathbf{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, float **differences,
```

```
    IMSLS_DIFFERENCES_USER, float differences [ ],
```

```
    IMSLS_N_MISSING, int *n_missing,
```

```

IMSL_RETURN_USER, float test_statistic[ ],
IMSL_FCN_W_DATA, float cdf ( ), void *data,
0)

```

Optional Arguments

`IMSL_DIFFERENCES`, *float **differences* (Output)

Address of a pointer to the internally allocated array containing D_n , D_n^+ , D_n^- .

`IMSL_DIFFERENCES_USER`, *float differences[]* (Output)

Storage for the array `differences` is provided by the user.

See `IMSL_DIFFERENCES`.

`IMSL_N_MISSING`, *int *n_missing* (Output)

Number of missing values is returned in `*n_missing`.

`IMSL_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.

`IMSL_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 [Passing Data to User-Supplied Functions](#) section 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:

- $H_0:F(x) = F^*(x)$ $H_1:F(x) \neq F^*(x)$
- $H_0:F(x) \geq F^*(x)$ $H_1:F(x) < F^*(x)$
- $H_0:F(x) \leq F^*(x)$ $H_1:F(x) > F^*(x)$

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 $n = \text{n_observations} - \text{n_missing}$. The test statistics for both one-sided alternatives

$$D_n^+ = \text{differences}[1]$$

and

$$D_n^- = \text{differences}[2]$$

and the two-sided ($D_n = \text{differences}[0]$) 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);
int 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

```

Warning Errors

IMSLS_TIE_DETECTED

ties were detected in the sample.

Fatal Errors

IMSLS_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

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, float **differences,
```

```

    IMSLS_DIFFERENCES_USER, float differences[ ],
    IMSLS_N_MISSING_X, int *xmissing,
    IMSLS_N_MISSING_Y, int *ymissing,
    IMSLS_RETURN_USER, float test_statistic[ ],
    0)

```

Optional Arguments

`IMSLD_DIFFERENCES`, *float **differences* (Output)

Address of a pointer to the internally allocated array containing D_n , D_n^+ , D_n^- .

`IMSLD_DIFFERENCES_USER`, *float differences[]* (Output)

Storage for array `differences` is provided by the user.

See `IMSLD_DIFFERENCES`.

`IMSLD_N_MISSING_X`, *int *xmissing* (Output)

Number of missing values in the `x` sample is returned in `*xmissing`.

`IMSLD_N_MISSING_Y`, *int *ymissing* (Output)

Number of missing values in the `y` sample is returned in `*ymissing`.

`IMSLD_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 `imsld_f_kolmogorov_two` computes Kolmogorov-Smirnov two-sample test statistics for testing that two continuous cumulative distribution functions (CDFs) 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` \times `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 $n = \text{n_observations_x} - \text{n_missing_x}$ and $m = \text{n_observations_y} - \text{n_missing_y}$, and let the corresponding population distribution functions be denoted by $F(x)$ and $G(y)$, respectively. Then, the hypotheses tested by `imsld_f_kolmogorov_two` are as follows:

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

The test statistics are given as follows:

$$D_{mn} = \max(D_{mn}^+, D_{mn}^-) \quad (\text{differences [0]})$$

$$D_{mn}^+ = \max_x(F_n(x) - G_m(x)) \quad (\text{differences [1]})$$

$$D_{mn}^- = \max_x(G_m(x) - F_n(x)) \quad (\text{differences [2]})$$

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 10^4 , according to an algorithm given by Kim and Jennrich (1973;). When $n*m$ is greater than 10^4 , the very good approximations given by Kim and Jennrich are used to obtain the two-sided p -values. The one-sided probability is taken as one half the two-sided probability. This is a very good approximation when the p -value is small (say, less than 0.10) and not very good for large p -values.

Example

This 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>
int main()
{
    float *statistics=NULL, *diffs = NULL, *x=NULL, *y=NULL;
    int nobsx = 100, nobsy = 60, nmissx, nmissy;
    imsls_random_seed_set(123457);
    x = imsls_f_random_uniform(nobsx, 0);
    y = imsls_f_random_uniform(nobsy, 0);
    statistics = imsls_f_kolmogorov_two(nobsx, x, nobsy, y,
                                       IMSLS_N_MISSING_X, &nmissx,
                                       IMSLS_N_MISSING_Y, &nmissy,
                                       IMSLS_DIFFERENCES, &diffs,
                                       0);
    printf("D      = %8.4f\n", diffs[0]);
    printf("D+    = %8.4f\n", diffs[1]);
}
```

```
    printf("D-      = %8.4f\n", diffs[2]);  
    printf("Z       = %8.4f\n", statistics[0]);  
    printf("Prob greater D one sided = %8.4f\n", statistics[1]);  
    printf("Prob greater D two sided = %8.4f\n", statistics[2]);  
    printf("Missing X = %d\n", nmissx);  
    printf("Missing Y = %d\n", nmissy);  
}
```

Output

```
D      = 0.1800  
D+     = 0.1800  
D-     = 0.0100  
Z      = 1.1023  
Prob greater D one sided = 0.0720  
Prob greater D two sided = 0.1440  
Missing X = 0  
Missing Y = 0
```

multivar_normality_test

Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.

Synopsis

```
#include <imsls.h>

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

The type *double* function is `imsls_d_multivar_normality_test`.

Required Arguments

int `n_observations` (Input)

Number of observations (number of rows of data) **x**.

int `n_variables` (Input)

Dimensionality 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

<code>i</code>	<code>stat[i]</code>
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.

i	stat[i]
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, 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_observations` containing the frequencies. Frequencies must be integer valued. Default assumes all frequencies equal one.

IMSLS_WEIGHTS, *float weights[]* (Input)

Array of size **n_observations** 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, respectively, for $p = \mathbf{n_variables}$. These measures are then used in computing tests for multivariate normality. Three test statistics, one based upon $b_{1,p}$ alone, one based upon $b_{2,p}$ alone, and an omnibus test statistic formed by combining normal scores obtained from $b_{1,p}$ and $b_{2,p}$ are computed. On the order of np^3 , operations are required in computing $b_{1,p}$ when the method of Isogai (1983) is used, where $n = \mathbf{n_observations}$. 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, $N(\boldsymbol{\mu}, \boldsymbol{\Sigma}/w_i)$ distribution, where $\boldsymbol{\Sigma}$ 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 $p = \text{n_variables} = 1$. 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:

$$S_W = W_1^2(b_{1,p}) + W_2^2(b_{2,p})$$

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 this 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>
int 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\n",
    ni, swt);
printf(" Number rows missing = %3d\n", nrmiss);
imsls_f_write_matrix("stat", 13, 1, stats,
    IMSLS_ROW_NUMBER_ZERO, 0);
imsls_f_write_matrix("means", 1, nvar, xmean, 0);
imsls_f_write_matrix("R", nvar, nvar, r, 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

```

ad_normality_test

Performs an Anderson–Darling test for normality.

Synopsis

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

```
float imsls_f_ad_normality_test (int nobs, float x[ ], ... , 0)
```

The type *double* function is `imsls_d_ad_normality_test`.

Required Arguments

int **nobs** (Input)

Number of observations. **nobs** must be greater than or equal to 3.

float **x[]** (Input)

Vector of length **nobs** containing the observations.

Return Value

The *p*-value for the Anderson–Darling test of normality.

Synopsis with Optional Arguments

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

```
float imsls_f_ad_normality_test (int nobs, float x[ ],
```

```
    IMSLS_STAT, float *adstat,
```

```
    IMSLS_N_MISSING, int *nmiss,
```

```
    0)
```

Optional Arguments

IMSLSTAT, *float *adstat* (Output)

The Anderson–Darling statistic.

IMSLNMISSING, *int *nmiss* (Output)

The number of missing observations.

Description

Given a data sample $\{X_i, i = 1 \dots n\}$, where $n = \mathbf{nobs}$ and $X_i = \mathbf{x}[i-1]$, function `imsls_f_ad_normality_test` computes the Anderson-Darling (AD) normality statistic $A = \mathbf{adstat}$ and the corresponding Return Value (p -value) $P = \mathbf{p}$ == {probability that a normally distributed n element sample would have an AD statistic $> A$ }. If P is sufficiently small (e.g. $P < .05$), then the AD test indicates that the null hypothesis that the data sample is normally-distributed should be rejected. A is calculated:

$$A = -n - \frac{1}{n} \sum_{i=1}^n [(2i-1)\ln(\phi(Y_i)) + (2n-2i+1)\ln(1-\phi(Y_i))]$$

where $Y_i = (X_i - \bar{X})/s$ and \bar{X} and s are the sample mean and standard deviation respectively. P is calculated by first transforming A to an “ n -adjusted” statistic A^* :

$$A^* = A \left(1.0 + \frac{0.75}{n} + \frac{2.25}{n^2} \right)$$

and then calculating P in terms of A^* using a parabolic approximation taken from Table 4.9 in Stephens (1986).

Example

The following example is taken from Conover (1980, pages 364 and 195). The data consists of 50 two-digit numbers taken from a telephone book. The AD test fails to reject the null hypothesis of normality at the .05 level of significance.

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

int main()
{
    int nobs = 50, nmiss;
    float p_value, adstat;

    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};
```

```
p_value = imsls_f_ad_normality_test (nobs, x,  
                                     IMSLS_STAT, &adstat,  
                                     IMSLS_N_MISSING, &nmiss,  
                                     0);
```

```
printf ("Anderson-Darling statistic = %11.4f \n", adstat);  
printf ("p-value = %11.4f\n", p_value);  
printf ("# missing values = %4d\n", nmiss);  
}
```

Output

```
Anderson-Darling statistic =    0.3339  
p-value =          0.5024  
# missing values =    0
```

Informational Errors

IMSLS_PVAL_UNDERFLOW

The p -value has fallen below the minimum value of # for which its calculation has any accuracy; ZERO is returned.

Fatal Errors

IMSLS_TOO_MANY_MISSING

After removing the missing observations only 2 observations remain. The test cannot proceed.

cvm_normality_test

Performs a Cramer-von Mises test for normality.

Synopsis

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

```
float imsls_f_cvm_normality_test (int nobs, float x[ ], ... , 0)
```

The type *double* function is `imsls_d_cvm_normality_test`.

Required Arguments

int **nobs** (Input)

Number of observations. **nobs** must be greater than or equal to 3.

float **x[]** (Input)

Vector of length **nobs** containing the observations.

Return Value

The *p*-value for the Cramer-von Mises test of normality.

Synopsis with Optional Arguments

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

```
float imsls_f_cvm_normality_test (int nobs, float x[ ],
```

```
    IMSLS_STAT, float *cvmstat,
```

```
    IMSLS_N_MISSING, int *nmiss,
```

```
    0)
```

Optional Arguments

IMSL_STAT, *float *cvmstat* (Output)

The Cramer-von Mises statistic.

IMSL_N_MISSING, *int *nmiss* (Output)

The number of missing observations.

Description

Given a data sample $\{X_i, i=1 \dots n\}$, where $n = \text{nobs}$ and $X_i = \mathbf{x}[i-1]$, function

`imsls_f_cvm_normality_test` computes the Cramer-von Mises (CvM) normality statistic $W = \text{cvmstat}$ and the corresponding Return Value (p -value) $P = \mathbf{P} == \{\text{probability that a normally distributed } n \text{ element sample would have a CvM statistic } > W\}$. If P is sufficiently small (e.g. $P < .05$), then the CvM test indicates that the null hypothesis that the data sample is normally-distributed should be rejected. W is calculated:

$$W = \frac{1}{12n} + \sum_{i=1}^n \left[\phi(Y_i) - \frac{2i-1}{2n} \right]^2$$

Where $\phi(Y_i)$ is the cumulative distribution function of standard normal $N(0,1)$ distribution, $Y_i = (X_i - \bar{X})/s$, and \bar{X} and s are the sample mean and standard deviation respectively. P is calculated by first transforming W to an “ n -adjusted” statistic W^* :

$$W^* = W \left(1.0 + \frac{0.5}{n} \right)$$

and then calculating P in terms of W^* using a parabolic approximation taken from Table 4.9 in Stephens (1986).

Example

This example is taken from Conover (1980, pages 364 and 195). The data consists of 50 two digit numbers taken from a telephone book. The CvM test fails to reject the null hypothesis of normality at the .05 level of significance.

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

int main()
{
    int nobs = 50, nmiss;
    float p_value, cvmstat;

    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};
```

```
p_value = imsls_f_cvm_normality_test (nobs, x,  
                                     IMSLS_STAT, &cvmstat,  
                                     IMSLS_N_MISSING, &nmiss,  
                                     0);
```

```
printf ("Cramer-von Mises statistic = %11.4f \n", cvmstat);  
printf ("p-value = %11.4f\n", p_value);  
printf ("# missing values = %4d\n", nmiss);  
}
```

Output

```
Cramer-von Mises statistic =      0.0520  
p-value =          0.4747  
# missing values =      0
```

Informational Errors

IMSLS_PVAL_UNDERFLOW

The p -value has fallen below the minimum value of # for which its calculation has any accuracy; ZERO is returned.

Fatal Errors

IMSLS_TOO_MANY_MISSING

After removing the missing observations only 2 observations remain. The test cannot proceed.

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, float x[ ], int n_run,  
    IMSLS_IDO, int ido, float intermediate_results[ ],  
    IMSLS_RUNS, float **runs_count, float **covariances, or  
    IMSLS_RUNS_USER, float runs_count[ ], float covariances[ ], or  
    IMSLS_PAIRS, int pairs_lag, float **pairs_count, or
```

IMSLS_PAIRS_USER, *int* pairs_lag, *float* pairs_count[], or
 IMSLS_DSQUARE, *float* **dsquare_count, or
 IMSLS_DSQUARE_USER, *float* dsquare_count[], or
 IMSLS_DCUBE, *float* **dcube_count, or
 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,
 0)

Optional Arguments

IMSLS_IDO, *int* ido, *float* intermediate_results[] (Input/Output)

Process data in blocks.

int ido (Input)

Processing option. The argument `ido` must be 1, 2, or 3. With this option, it is not a requirement that all observations be memory resident, thus enabling one to handle large data sets. Blocks of rows of the data can be processed sequentially in separate invocations of `imsls_f_randomness_test`. Output argument values are returned only when `ido = 3`. (See [Example 5](#).)

ido	Action
1	This is the first invocation with this data; additional calls will be made. The first set of <code>n_observations</code> observations is input in <code>x</code> .
2	This is an intermediate invocation. The next set of <code>n_observations</code> observations is input in <code>x</code> .
3	This is the final invocation of this function. No further invocations of <code>imsls_f_randomness_test</code> with <code>ido</code> greater than 1 should be made without first invoking <code>imsls_f_randomness_test</code> with <code>ido = 1</code> . The last set of <code>n_observations</code> observations is input in <code>x</code> .

Default: `ido` is not used. All the data is input at once.

float intermediate_results[] (Input/Output)

User-supplied array containing results from invocations of the function. The length of *intermediate_results* is:

Test	Length
Runs test (IMSLS_RUNS)	<i>n_run</i>
Pairs test (IMSLS_PAIRS)	<i>n_run</i> by <i>n_run</i>
d^2 test (IMSLS_DSQUARE)	<i>n_run</i>
triplets test (IMSLS_DCUBE)	<i>n_run</i> by <i>n_run</i> by <i>n_run</i>

In processing blocks of data, *x* can have different number of observations, *n_observations*, in separate invocations.

IMSLS_RUNS, *float **runs_count*, *float **covariances*, (Output)

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_count*. *n_run* by *n_run* 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 the *IMSLS_RUNS* argument must be used.

or

IMSLS_RUNS_USER, *float runs_count[]*, *float covariances[]* (Output)

Storage for *runs_count* and *covariances* is provided by the user. See *IMSLS_RUNS*.

or

IMSLS_PAIRS, *int pairs_lag* (Input), *float **pairs_count*, (Output)

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. An *n_run* by *n_run* matrix containing the count of the number of pairs in each cell is returned in *pairs_count*.

or

IMSLS_PAIRS_USER, *int pairs_lag*, *float pairs_count[]* (Output)

Storage for *pairs_lag* and *pairs_count* is provided by the user. See *IMSLS_PAIRS*.

or

IMSLS_DSQUARE, *float **dsquare_count*, (Output)

Indicates the d^2 test is to be performed. *dsquare_count* is an address of a pointer to an internally allocated array of length *n_run* containing the tabulations for the d^2 test.

or

`IMSLS_DSQUARE_USER`, *float dsquare_count* [] (Output)

Storage for `dsquare_count` is provided by the user.

See `IMSLS_DSQUARE`.

or

`IMSLS_DCUBE`, *float **dcube_count*, (Output)

Indicates the triplets test is to be performed. `dcube_count` 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.

or

`IMSLS_DCUBE_USER`, *float dcube_count* [] (Output)

Storage for `dcube_count` is provided by the user. See `IMSLS_DCUBE`.

`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. This option is valid only for the runs test.

`IMSLS_RUNS_EXPECT_USER`, *float runs_expect* [] (Output)

Storage for `runs_expect` is provided by the user. See `IMSLS_RUNS_EXPECT`.

`IMSLS_EXPECT`, *float *expect* (Output)

Expected number of counts for each cell. This argument is valid only if one of `IMSLS_PAIRS`, `IMSLS_DSQUARE`, or `IMSLS_DCUBE` is used. It is not valid for the runs test.

`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.

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.

`IMSL_S_RUNS` first tallies the number of runs up (increasing sequences) of each desired length. For $i = 1, \dots, r - 1$, where $r = \mathbf{n_run}$, `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, `IMSL_S_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

`IMSL_S_PAIRS` computes the pairs test (or the Good's serial test) on a hypothesized sequence of uniform (0,1) pseudo-random numbers. The test proceeds as follows. Subsequent pairs (`x[i]`, `x[i + pairs_lag]`) are tallied into a $k \times k$ matrix, where $k = \mathbf{n_run}$. In this tally, element (j, m) of the matrix is incremented, where

$$\begin{aligned} j &= [kx[i - 1]] + 1 \\ m &= [kx[i + l - 1]] + 1 \end{aligned}$$

where $l = \mathbf{pairs_lag}$, and the notation $[\]$ represents the greatest integer function, $[Y]$ is the greatest integer less than or equal to Y , where Y is a real number. If $l = 1$, then $i = 1, 3, 5, \dots, n - 1$. If $l > 1$, then $i = 1, 2, 3, \dots, n - l$, where n is the total number of pseudo-random numbers input on the current invocation of `IMSL_S_PAIRS` (i.e., $n = \mathbf{n_observations}$).

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 $e = \Sigma o_{ij} / k^2$, and o_{ij} is the observed count in cell (i, j) ($o_{ij} = \mathbf{pairs_count}[i][j]$).

Because pair statistics for the trailing observations are not tallied on any call, the user should call `IMSL5_PAIRS` with `n_observations` as large as possible. For `pairs_lag < 20` and `n_observations = 2000`, little power is lost.

d^2 Test

`IMSL5_DSQUARE` computes the d^2 test for succeeding quadruples of hypothesized pseudo-random uniform (0, 1) deviates. The d^2 test is performed as follows. Let X_1, X_2, X_3 , and X_4 denote four pseudo-random 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

$$\Pr(D^2 \leq d^2) = d^2\pi - \frac{8d^3}{3} + \frac{d^4}{2}$$

when $D^2 \leq 1$, where π denotes the value of pi. If $D^2 > 1$, this probability is given as

$$\Pr(D^2 \leq d^2) = \frac{1}{3} + (\pi - 2)d^2 + 4\sqrt{d^2 - 1} + 8\frac{(d^2 - 1)^{\frac{3}{2}}}{3} - \frac{d^4}{2} - 4d^2 \arctan\left(\frac{\sqrt{1 - \frac{1}{d^2}}}{\frac{1}{d}}\right)$$

See Gruenberger and Mark (1951) for a derivation of this distribution.

For each succeeding set of 4 pseudo-random uniform numbers input in `x`, d^2 and the cumulative probability of d^2 ($\Pr(D^2 \leq d^2)$) 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 pseudo-random 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

IMSL5_DCUBE computes the triplets test on a sequence of hypothesized pseudo-random 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 $m = \mathbf{n_run}$. 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 pseudo-random 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

$$\chi^2 = \sum_{i,j,k=0}^{m-1} \frac{(o_{ijk} - e)^2}{e}$$

where $o_{ijk} = \mathbf{dcube_count [i] [j] [k]}.$

The computed chi-squared has $m^3 - 1$ degrees of freedom, and the null hypothesis of pseudo-random uniform (0, 1) deviates is rejected if χ^2 is too large.

Examples

Example 1

This example illustrates the use of the runs test on 10^4 pseudo-random uniform deviates. 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>
int main()
{
    int nran = 10000, n_run = 6;
    char *fmt = "%8.1f";
    float *x, pvalue, *runs_count, *runs_expect, *covariances, 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_count, &covariances,
                                    0);
    imsls_f_write_matrix("runs_count", 1, n_run, runs_count, 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

This example illustrates the calculations of the `IMSL_S_PAIRS` statistics when a random sample of size 10^4 is used and the `pairs_lag` is 1. The results are not significant. IMSL function `imsls_f_random_uniform` (Chapter 12, [Random Number Generation](#)) is used in obtaining the pseudo-random deviates.

```

#include <imsls.h>
#include <stdio.h>
int main()
{
    int nran = 10000, n_run = 10;
    float *x, pvalue, *pairs_count, 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_count,
                                    0);
    imsls_f_write_matrix("pairs_count", n_run, n_run, pairs_count, 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 this example, 2000 observations generated via IMSL function `imsls_f_random_uniform` (Chapter 12, [Random Number Generation](#)) are input to `IMSL_DSQUARE` in one call. In the example, the null hypothesis of a uniform distribution is not rejected.

```

#include <imsls.h>
#include <stdio.h>
int 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 this example, 2001 deviates generated by IMSL function `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>
int 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

```

Example 5

This example is based on Example 1 to illustrate the use of the `IMSL_S_IDO` optional argument. In this example, `imsls_f_randomness_test` is called 10 times, with 1000 pseudo-random uniform deviates each time. 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>

#define NRAN 1000
#define N_RUN 6

int main()
{
    int ido = 1, i;
    char *fmt = "%8.1f";
    float x[NRAN], intermediate_results[N_RUN], pvalue, *runs_count,
          *runs_expect, *covariances, chisq, df;

    imsls_random_seed_set(123457);

    for (i = 0; i < 10; i++) {
        if (i == 9) ido = 3;

        imsls_f_random_uniform(NRAN, IMSLS_RETURN_USER, x, 0);

        pvalue = imsls_f_randomness_test(NRAN, x, N_RUN,
            IMSLS_IDO, ido, intermediate_results,
            IMSLS_CHI_SQUARED, &chisq,
            IMSLS_DF, &df,
            IMSLS_RUNS_EXPECT, &runs_expect,
            IMSLS_RUNS, &runs_count, &covariances,
            0);
        ido = 2;
    }

    imsls_f_write_matrix("runs_count", 1, N_RUN, runs_count,
        IMSLS_WRITE_FORMAT, fmt,
        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 =	8.765146					
df =	6.000000					
pvalue =	0.187223					

Time Series and Forecasting

Functions

ARIMA Models

Computes least-squares or method of moments estimates of parameters	arma	696
Computes maximum likelihood estimates of parameters	max_arma	709
Computes forecasts and their associated probability limits	arma_forecast	716
Fits a univariate seasonal or non-seasonal ARIMA time series model	arma	724
Fits a univariate, non-seasonal ARIMA time series model	regression_arma	737

Automatic ARIMA Selection and Fitting Utilities

Automatic selection and fitting of a univariate autoregressive time series model	auto_uni_ar	748
Estimates the optimum seasonality parameters for a time series using an autoregressive model	seasonal_fit	755
Detects and determines outliers and simultaneously estimates the model parameters in a time series	ts_outlier_identification	764
Computes forecasts for an outlier contaminated time series	ts_outlier_forecast	774
Automatic ARIMA modeling and forecasting in the presence of possible outliers	auto_arma	783
Estimates structural breaks in non-stationary univariate time series models	auto_parm	801

Bayesian Time Series Estimation

Decomposes a time series into trend, seasonal, and an error component	bayesian_seasonal_adj	813
---	---------------------------------------	-----

Model Construction and Evaluation Utilities

Performs a Box-Cox transformation	box_cox_transform	822
Performs differencing on a time series	difference	827
Sample autocorrelation function	autocorrelation	833
Computes the sample cross correlation function	crosscorrelation	839
Computes the multichannel cross-correlation function	multi_crosscorrelation	846

Sample partial autocorrelation functionpartial_autocorrelation	857
Lack-of-fit test based on the correlation function	lack_of_fit	861
Estimates missing values in a time series	estimate_missing	865
Exponential Smoothing Methods		
Holt-Winters additive or multiplicative method	hw_time_series	873
GARCH Modeling		
Computes estimates of the parameters of a GARCH (p,q) model	garch	883
State-Space Models		
Performs Kalman filtering and evaluates the likelihood function for the state-space model	kalman	889
Vector Auto-Regression and Error Correction		
Estimates a vector auto-regressive time series model with optional moving average components.	vector_autoregression	901

Usage Notes

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

Model Construction and Evaluation Utilities

A major component of the model identification step concerns determining if a given time series is stationary. The sample correlation functions computed by functions `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 function `imsls_f_box_cox_transform` coupled with the ability to difference the transformed data using function `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 function `imsls_f_random_arma` (see Chapter 12, [Random Number Generation](#)) may be used to generate a time series according to a specified autoregressive moving average model.

ARIMA Models

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

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

where $Z = \{\dots, -2, -1, 0, 1, 2, \dots\}$ 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:

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

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

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

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

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 back-casting 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).

Regression in Autoregressive Integrated Moving Average

There may be one or more external time series that relate to the time series of interest, which may be useful in improving forecasts. Function `imsls_f_regression_arma` allows for the inclusion of one or more regression time series in the above ARIMA model. That is, if there are r time series $\{X_{i,t}, i = 1, \dots, r\}$ associated with a time series Y_t , the regression ARIMA model (integrated of order d) is

$$W_t = \nabla^d Z_t$$

where

$$Z_t = \left(Y_t - \sum_{i=1}^r \beta_i X_{i,t} \right)$$

That is, Z_t is the residual (indexed by t) of the regression of Y_t on $\{X_{i,t}, i = 1, \dots, r\}$.

Automatic ARIMA Selection and Fitting Utilities

A popular criterion for comparing autoregressive-moving average (ARMA) models with different lags is a measure known as Akaike's Information Criterion (AIC). The AIC for an ARMA univariate series is calculated by:

$$AIC = -2 \cdot \ln(L) + 2r$$

where L = the value for the maximum likelihood function for the fitted model, and $r = p + q + 1$, the number of parameters in the ARMA model. To use the criterion, several choices for p and q are fit to a time series and the fit with the smallest AIC is considered best. The function, `imsls_f_auto_uni_ar` uses the AIC criterion to select a best fitting AR model. The function `imsls_f_auto_arima` performs a more comprehensive search, considering not only the ARMA parameters, but also the appropriate Box-Cox transformation, degree of differencing and seasonal adjustment, and also filters the data for outliers by calling `imsls_f_ts_outlier_identification`.

The function `imsls_f_auto_parm` uses a second criterion, called "Minimum Description Length" or MDL, to automatically fit piecewise AR models to a time series with structural breaks (i.e., a potentially non-stationary time series having stationary segments).

The MDL is defined as

$$\begin{aligned} MDL(m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}) \\ = \ln m + (m-1) \ln n + \sum_{j=1}^{m+1} \frac{2 + p_j}{2} \ln n_j - \ln L, \end{aligned}$$

where m is the number of structural breaks in the series, $\{\tau_1, \tau_2, \dots, \tau_{m+1}\}$ are the locations of the breaks, n_j is the number of observations in the j -th segment, p_j is the order of the AR model fit to the j -th segment, and L is the combined likelihood over all segments. `imsls_f_auto_parm` also allows the choice to use the AIC criterion,

$$\begin{aligned}
 AIC(m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}) &= 2(\text{number of parameters}) - 2 \ln L \\
 &= 2 \left(1 + m + \sum_{j=1}^{m+1} (2 + p_j) \right) - 2 \ln L
 \end{aligned}$$

Exponential Smoothing Methods

Exponential smoothing approximates the value of a time series at time t with a weighted average of previous values, with weights defined in such a way that they decay exponentially over time.

The weights can be determined by a smoothing parameter α and the relation,

$$\begin{aligned}
 y_t &= \alpha y_{t-1} + \hat{y}_t \\
 &\Rightarrow \\
 y_t &= \sum_{j=0}^{t-1} \alpha^{t-j} (1 - \alpha)^j y_j = \sum_{j=0}^{t-1} w_j y_j
 \end{aligned}$$

The parameter α is on the interval $(0,1)$ and controls the rate of decay. For values close to 1, the effect decays rapidly, whereas for values close to 0, the influence of a past value persists for some time. Exponential smoothing as a forecasting procedure is widely used and largely effective for short term, mean level forecasting. With the addition of a term for linear trend and terms or factors for seasonal patterns, exponential smoothing is an intuitive procedure for many series that arise in business applications. The function `imsls_f_hw_time_series` performs the Holt-Winters method, otherwise known as *triple exponential smoothing*, and allows for either an additive or a multiplicative seasonal effect.

Garch Models

An important assumption in the ARMA model

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

is that the errors A_t are independent random variables with mean 0 and constant variance, σ^2 .

For some time series, the assumptions of independent errors and constant variance will not hold for the underlying series. For example, in the stock market, large errors in one direction are often followed by large errors in the opposite direction. Variability over time can increase with a price level or trading volume. To account for heteroscedastic (non-equal) variances, Engle (1982) introduced the Autoregressive Conditional Heteroscedastic or ARCH, model:

$$A_t = z_t \sigma_t$$

$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^q \alpha_i A_{t-i}^2,$$

where the z_t 's are independent and identically distributed standard normal random variables. In the ARCH model, the variance term depends on previous squared errors, up to a given lag q . A generalized ARCH model, called GARCH, was introduced by Bollerslev (1986) and has the form:

$$A_t = z_t \sigma_t$$

$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^p \beta_i \sigma_{t-i}^2 + \sum_{i=1}^q \alpha_i A_{t-i}^2,$$

In the GARCH model, the variance has an auto-regressive term in addition to the squared error term. The function `imsls_f_garch` estimates ARCH or GARCH models.

State-Space Models

A state-space model is represented by two equations: the *state equation*

$$\dot{b}(t) = f(t, b(t), u(t))$$

and the observation equation,

$$Y(t) = h(t, b(t), u(t))$$

where $b(t)$ is the state variable, $Y(t)$ is the observation variable, $u(t)$ is a vector of inputs, and

$$\dot{b}(t) = \frac{d}{dt} b(t)$$

State-space models originated in the study of dynamical systems. The system state $b(t)$ is not directly observed but is inferred from the observable variable, $Y(t)$, through the relation defined by the function, h . $Y(t)$ is sometimes called the *measurement variable* or *output variable*. While f and h are completely general functions in the definition, they are most often linear, based on the assumption that the underlying system behaves linearly or approximately so. There is often a stochastic or noise term added to the equations and, in the time series context, there are usually no control inputs ($u(t) = 0$). Under these conditions, the state-space model takes the form,

$$\dot{b}(t) = Z(t)b(t) + w(t)$$

and

$$Y(t) = T(t)b(t) + e(t)$$

where Z and T are known matrices and w and e are noise variables. Time may evolve continuously or discretely. For a discrete time variable, it is customary to write the equations as:

$$b(k+1) = Z(k)b(k) + w(k)$$

and

$$Y(k) = T(k)b(k) + e(k)$$

where $k = \dots-3, -2, -1, 0, 1, 2, 3, \dots$

Many time series can be expressed in the state-space form, including ARIMA models (See Section 5.5 in Box, Jenkins, and Reinsel (2008)). For ARIMA models, the state-space form is convenient for the calculation of the likelihood and forecasts, and for handling missing values. Once a time series is formulated as a state-space model, the problem becomes one of solving the equations for the unknown parameters. The Kalman filter ([ims1s_f_kalman](#)) is a recursive method for solving the state-space equations when the functions f and h are linear.

arma



[more...](#)

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 `IMSL_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, int ma_lags[ ],  
                  IMSLS_METHOD_OF_MOMENTS, or  
                  IMSLS_LEAST_SQUARES,  
                  IMSLS_BACKCASTING, int maxbc, float tolerance,  
                  IMSLS_RELATIVE_ERROR, float relative_error,  
                  IMSLS_ABS_FCN_TOL, float afcntol,  
                  IMSLS_GRAD_TOL, float grad_tol,  
                  IMSLS_STEP_TOL, float step_tol,  
                  IMSLS_MAX_ITERATIONS, int max_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_VAR_NOISE, float *avar,  
                  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.

Default: `IMSLS_METHOD_OF_MOMENTS` is used.

`IMSLS_BACKCASTING`, *int* `maxbc`, *float* `tolerance` (Input)

If `IMSLS_BACKCASTING` is specified, `maxbc` 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: `maxbc` = 10; `tolerance` = 0.01 × standard deviation of `z`.

`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_ABS_FCN_TOL, *float* **afcncntol** (Input)

The absolute function tolerance used by the nonlinear least-squares solver that determines the MA parameters in the method of moments. This variable is only needed for non-standard ARMA models.

Default: **afcncntol** = $\max(10^{-10}, \epsilon^{2/3})$, where $\epsilon = \text{imsls_f_machine}(4)$ is the machine precision.

See documentation for function [imsls_f_machine](#) (Chapter 15, [Utilities](#)).

IMSLS_GRAD_TOL, *float* **grad_tol** (Input)

The scaled gradient tolerance used by the nonlinear least-squares solver that determines the MA parameters in the method of moments. This variable is only needed for non-standard ARMA models.

Default: **grad_tol** = $\sqrt{\epsilon}$ in single, $\sqrt[3]{\epsilon}$ in double, where $\epsilon = \text{imsls_f_machine}(4)$ is the machine precision.

See documentation for function [imsls_f_machine](#) (Chapter 15, [Utilities](#)).

IMSLS_STEP_TOL, *float* **step_tol** (Input)

The scaled step tolerance used by the nonlinear least-squares solver that determines the MA parameters in the method of moments. This variable is only needed for non-standard ARMA models.

Default: **step_tol** = $\epsilon^{2/3}$, where $\epsilon = \text{imsls_f_machine}(4)$ is the machine precision.

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 $na = (n_observations - \max(ar_lags[l]) + \max(bc))$ containing the residuals (including backcasts) at the final parameter estimate point in the first $n_observations - \max(ar_lags[l]) + nb$, 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 containing the variance-covariance matrix of the estimated ARMA parameters and (optionally) of the estimated mean of series z . The size of the array is $np \times np$, 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_AUTOCOVAR, *float **autocov* (Output)

Address of a pointer to an array of length $p + q + 2$ 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 at lag k , where $k = 0, 1, \dots, p + q + 1$.

IMSLS_AUTOCOVAR_USER, *float autocov[]* (Output)

Storage for array `autocov` is provided by the user. See **IMSLS_AUTOCOVAR**.

IMSLS_SS_RESIDUAL, *float *ss_residual* (Output)

If specified, `ss_residual` contains the sum of squares of the random shock, $ss_residual = residual[1]^2 + \dots + residual[na]^2$, where `na` is equal to the number of residuals.

IMSLS_VAR_NOISE, *float *avar* (Output)

If specified, `avar` contains the innovation variance of the series.

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. If p or q equals zero, a `NULL` array may be used.

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](#).

Description

Function `imsls_f_arma` computes estimates of parameters for a nonseasonal ARMA model given a sample of observations, $\{W_t\}$, for $t = 1, 2, \dots, n$, where $n = \mathbf{n_observations}$. 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
<code>IMSLS_METHOD_OF_MOMENTS</code>	<code>IMSLS_LEAST_SQUARES</code>	<code>IMSLS_RELATIVE_ERROR</code>
<code>IMSLS_ABS_FCN_TOL</code>	<code>IMSLS_CONSTANT</code> (or <code>IMSLS_NO_CONSTANT</code>)	<code>IMSLS_MAX_ITERATIONS</code>
<code>IMSLS_GRAD_TOL</code>	<code>IMSLS_BACKCASTING</code>	<code>IMSLS_MEAN_ESTIMATE</code>
<code>IMSLS_STEP_TOL</code>	<code>IMSLS_INITIAL_ESTIMATES</code>	<code>IMSLS_AUTOCOV(_USER)</code>
	<code>IMSLS_RESIDUAL(_USER)</code>	<code>IMSLS_RETURN_USER</code>
	<code>IMSLS_PARAM_EST_COV(_USER)</code>	<code>IMSLS_ARMA_INFO</code>
	<code>IMSLS_SS_RESIDUAL</code>	<code>IMSLS_AR_LAGS</code>
		<code>IMSLS_MA_LAGS</code>
		<code>IMSLS_VAR_NOISE</code>

Method of Moments Estimation

The method of moments assumes that the stationary time series $\{Z_t\}$ can be described by a nonseasonal ARMA model of the form

$$\phi(B)Z_t = \theta_0 + \theta(B)A_t, t \in \{0, \pm 1, \pm 2, \dots\}$$

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

$$\begin{aligned}\phi(B) &= 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \dots - \phi_p B^{l_\phi(p)} && \text{for } p \geq 0 \\ \theta(B) &= 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \dots - \theta_q B^{l_\theta(q)} && \text{for } q \geq 0\end{aligned}$$

with p autoregressive and q moving average parameters.

Function `imsls_f_arma` first orders the AR and MA lags in strictly increasing order. Without loss of generality, it therefore can be assumed that

$$1 \leq l_\phi(1) < l_\phi(2) < \dots < l_\phi(p), 1 \leq l_\theta(1) < l_\theta(2) < \dots < l_\theta(q)$$

so that the nonseasonal ARMA model is of order (p', q') , where $p' = l_\phi(p)$ and $q' = l_\theta(q)$.

In order to keep the notation simple, the following explanations assume the standard ARMA (p, q) model with $l_\phi(i) = i, i = 1, \dots, p$, and $l_\theta(i) = i, i = 1, \dots, q$.

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

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

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 $k = 0, 1, \dots, K$, 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:

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

where

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

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 $k = 1, \dots, K$, and p autoregressive parameters ϕ_i for $i = 1, \dots, p$.

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:

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

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

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

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

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

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

and

$$f_j(\tau_0, \dots, \tau_q) = \sum_{i=0}^{q-j} \tau_i \tau_{i+j} - \hat{\sigma}'(j) \text{ for } j = 0, \dots, q.$$

The nonlinear system

$$f(\tau_0, \dots, \tau_q) = 0$$

is solved by a trust-region method. If $\tau^i = (\tau_0^i, \dots, \tau_q^i)$ is the estimate of τ obtained at the i -th iteration and if a full Newton step is possible, then the new value at the $(i + 1)$ -th iteration is determined by

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

where

$$T^i = \left(\frac{\partial f_j^i}{\partial \tau_k} \right)_{0 \leq j, k \leq q}$$

is a square matrix of order $q + 1$ with entries

$$\frac{\partial f_j^i}{\partial \tau_k} = \tau_{k+j}^i + \tau_{k-j}^i, \quad \tau_l^i = 0 \text{ for } l < 0 \text{ and } l > q.$$

The estimation procedure begins with the initial value

$$\tau^0 = (\sqrt{\hat{\sigma}^2(0)}, 0, \dots, 0)^T$$

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

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

The random shock variance is estimated by the following:

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

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,

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

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

$$\begin{aligned} \phi(B) &= 1 - \phi_1 B^{\phi(1)} - \phi_2 B^{\phi(2)} - \dots - \phi_p B^{\phi(p)} & \text{for } p \geq 0 \\ \theta(B) &= 1 - \theta_1 B^{\theta(1)} - \theta_2 B^{\theta(2)} - \dots - \theta_q B^{\theta(q)} & \text{for } q \geq 0 \end{aligned}$$

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

$$1 \leq l_{\phi}(1) \leq l_{\phi}(2) \leq \dots \leq l_{\phi}(p)$$

$$1 \leq l_{\theta}(1) \leq l_{\theta}(2) \leq \dots \leq l_{\theta}(q)$$

so that the nonseasonal ARMA model is of order (p', q') , where $p' = l_{\phi}(p)$ 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

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

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

$$S_{\infty}(\mu, \phi, \theta) / (2\sigma_A^2)$$

dominates

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

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

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

for the ARMA(2, 1) model

$$Z_t = \theta_0 + \phi_1 Z_{t-1} + \phi_2 Z_{t-2} - \theta_1 A_{t-1} + A_t$$

where the errors A_t are independently normally distributed with mean zero and variance σ_A^2 .

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

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

```
#include <imsls.h>  
#include <stdio.h>  
  
int main()  
{  
    int p=2, q=1, i, n_observations=100;  
    float w[176][2], z[100], *parameters;  
  
    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,  
                             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      1.5300 and      -0.8931.  
MA estimate is      -0.1324.
```

Warning Errors

IMSL5_LEAST_SQUARES_FAILED	Least-squares estimation of the parameters has failed to converge. Solution from last iteration is returned. The estimates of the parameters at the last iteration may be used as new starting values.
IMSL5_NEED_POSITIVE_GRADTL	The gradient tolerance must be nonnegative while "grad_tol" = # is given. The algorithm will use "grad_tol" = #.
IMSL5_NEGATIVE_STEP_TOL	The step tolerance must be nonnegative while "step_tol" = # is given. The algorithm will use "step_tol" = #.
IMSL5_NEGATIVE_ABS_FCN_TOL	The absolute function tolerance must be nonnegative while "afcntol" = # is given. The algorithm will use "afcntol" = #.

Fatal Errors

IMSL5_TOO_MANY_CALLS	The number of calls to the function has exceeded "itmax"*("n"+1) = %(i1). The user may try a new initial guess.
IMSL5_INCREASE_ERRREL	The bound for the relative error, "errrel" = %(r1), is too small. No further improvement in the approximate solution is possible. The user should increase "errrel".
IMSL5_NEW_INITIAL_GUESS	The iteration has not made good progress. The user may try a new initial guess.

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, lmsls_f_arma **arma_info,
IMSLS_MEAN_ESTIMATE, float *w_mean,
IMSLS_RESIDUAL, float **residuals,
IMSLS_RESIDUAL_USER, float residuals[],
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 options:

<i>iprint</i>	Action
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 $-2 \times (\ln(\text{likelihood}))$ 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_RESIDUAL**, *float **residuals* (Output)
Address of a pointer to an internally allocated array of length **n_obs** containing the residuals of the requested ARMA fit.
- IMSLS_RESIDUAL_USER**, *float residuals[]* (Output)
Storage array **residuals** is provided by the user. See **IMSLS_RESIDUAL**.
- 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

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

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

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

and

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

Function `imsls_f_max_arma` estimates the AR coefficients $\phi_1, \phi_2, \dots, \phi_p$ and the MA coefficients $\theta_1, \theta_2, \dots, \theta_q$ 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

$$\phi_1, \phi_2, \dots, \phi_p$$

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

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

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. Functions `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,

$$AIC = \log_likeli + 2(p + q)$$

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:

$$\tilde{w}_t = 1.22\tilde{w}_{t-1} - 0.56\tilde{w}_{t-2} + a_t + 0.38a_{t-1}$$

Defining the overall constant ϕ_0 by $\phi_0 := \mu \left(1 - \sum_{i=1}^p \phi_i\right)$, 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

$$w_t = 15.76 + 1.22w_{t-1} - 0.56w_{t-2} + a_t + 0.38a_{t-1}.$$

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

int 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, z,
        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)
    {
        imsls_free(parameters);
        parameters = NULL;
    }
}

```

Output

```

AR estimates are      1.2245 and      -0.5601.
MA estimate is       -0.3831.
Constant estimate is   15.7624.
-2*ln(Maximum Log Likelihood) =   539.5839.
White noise variance =   214.5123.

```

Example 2

This example is the same as [Example 1](#), but now initial values for the AR and MA parameters are explicitly given.

```

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

int 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, z,
                      0);

    for (i=0; i<n_obs; i++)
        w[i] = z[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);
}
```

Output

```
AR estimates are      1.2252 and     -0.5607.
MA estimate is      -0.3828.
Constant estimate is    15.7587.
-2*ln(Maximum Log Likelihood) =   539.5839.
White noise variance =   214.5083.
```

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_ONE_STEP_FORECAST, float **forecast,
```

```
    IMSLS_ONE_STEP_FORECAST_USER, float forecast[ ],
```

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[j]), maxma = max(ma_lags[j]), and n_observations = the number of observations in the series, as input in function imsls_f_arma. n_predict forecasts beginning at origins n_observations - backward_origin + 1 through n_observations are generated.

Default: backward_origin = 0.

IMSLS_ONE_STEP_FORECAST, float **forecast (Output)

Address of a pointer to an internally allocated array of length backward_origin + n_predict containing forecasts. The first backward_origin forecasts are one-step ahead forecasts for the last backward_origin values in the series. The next n_predict values in the returned series are forecasts for the next values beyond the series.

IMSLS_ONE_STEP_FORECAST_USER, float forecast[] (Output)

Storage for array forecast is provided by the user. See IMSLS_ONE_STEP_FORECAST.

IMSLS_RETURN_USER, float forecasts[] (Output)

If specified, a user-specified array of length n_predict × (backward_origin + 3) as defined below.

j	forecasts for lead times $l = 1, \dots, n_predict$ at origins $n_observations - backward_origin + 1 + j$, where $j = 0, \dots, backward_origin$
$backward_origin + 1$	deviations from each forecast that give the confidence percent probability limits
$backward_origin + 2$	psi weights of the infinite order moving average form of the model

Also see [Examples](#) for additional explanation of how to interpret this output.

Description

The Box-Jenkins forecasts and their associated probability limits for a nonseasonal ARMA model are computed given a sample of $n = \mathbf{n_observations}$ $\{Z_t\}$ for $t = 1, 2, \dots, n$, where $\mathbf{n_observations}$ = 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 $t \in \{0, \pm 1, \pm 2, \dots\}$, where B is the backward shift operator, θ_0 is the constant, and

$$\begin{aligned}\phi(B) &= 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \dots - \phi_p B^{l_\phi(p)} \\ \theta(B) &= 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \dots - \theta_q B^{l_\theta(q)}\end{aligned}$$

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

$$\begin{aligned}1 &\leq l_\phi(1) \leq l_\phi(2) \leq \dots \leq l_\phi(p) \\ 1 &\leq l_\theta(1) \leq l_\theta(2) \leq \dots \leq l_\theta(q)\end{aligned}$$

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

$$\begin{aligned}l_\phi(i) &= i, 1 \leq i \leq p \\ l_\theta(j) &= j, 1 \leq j \leq q\end{aligned}$$

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

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

where the following is true:

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

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

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

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

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

$$\sigma_A^2$$

(returned from `imsls_f_arma`) and

$$\{\psi_j\}$$

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

The Box-Jenkins forecasts minimize the mean-square error

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

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).

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. 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_arma` with the method of moments option. With `backward_origin = 3`, columns zero through three of `forecasts` provide forecasts starting with 1867, 1868, 1869, and 1870, respectively. Note that the values in the first row are the one-step ahead forecasts for 1867, 1868, 1869, and 1870; the values in the second row are the two-step ahead forecasts for 1868, 1869, 1870, and 1871; etc. Column four gives the deviations for computing probability limits, and column five 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 ∓ 33.22 .

```
#include <imsls.h>

int 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 1866	Forecast From 1867	Forecast From 1868	Forecast From 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		

Example 2

Using the same data as in example 1, option `IMSLS_ONE_STEP_FORECAST` is used to compute the one-step ahead forecasts with `backward_origin = 0` and `n_predict = 5`. This obtains the one-step ahead forecasts for the last 10 observations in the series, i.e. years 1860-1869, plus the next 5 years. The upper 90% confidence limits are computed for these forecasts using the deviations in column `backward_origin + 1` of `forecasts`.

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

int main()
{
    int p = 2;
    int q = 1;
    int i;
    int n_observations = 100;
    int max_iterations = 0;
    int n_predict = 5;
    int backward_origin = 10;
    int year=1860;
    int devindex;
    float w[176][2];
    float z[100];
    float *parameters;
    float rel_error = 0.0;
    float *forecasts;
    float *one_step_forecast;
    float confidence=90.;
    Imsls_f_arma *arma_info;

    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);

/* get one-step ahead forecasts */
forecasts = imsls_f_arma_forecast(arma_info, n_predict,
    IMSLS_BACKWARD_ORIGIN, backward_origin,
    IMSLS_ONE_STEP_FORECAST, &one_step_forecast,
    IMSLS_CONFIDENCE, confidence,
    0);

devindex = backward_origin+1; /* forecasts index for deviations */
printf ("          ARMA ONE-STEP AHEAD FORECASTS\n");
printf ("Year  Observed  Forecast   Residual  UCL(90%\x25) \n\n");

for (i=0; i<backward_origin; i++)
    printf ("%d   %7.3f   %7.3f   %7.3f   %7.3f\n", year+i,
        z[n_observations-backward_origin+i],
        one_step_forecast[i],
        z[n_observations-backward_origin+i]-one_step_forecast[i],
        one_step_forecast[i]+forecasts[devindex]);

for (i=backward_origin; i<backward_origin+n_predict; i++)
    printf ("%d      -      %7.3f      -      %7.3f\n",
        year+i, one_step_forecast[i],
        one_step_forecast[i]+
        forecasts[devindex+(i-backward_origin)*(backward_origin+3)] );
}

```

Output

ARMA ONE-STEP AHEAD FORECASTS				
Year	Observed	Forecast	Residual	UCL(90%)
1860	95.700	100.737	-5.037	128.615
1861	77.200	81.295	-4.095	109.173
1862	59.100	57.067	2.033	84.944
1863	44.000	44.426	-0.426	72.303
1864	47.000	36.353	10.647	64.230
1865	30.500	47.396	-16.896	75.274
1866	16.300	28.558	-12.258	56.436
1867	7.300	19.804	-12.504	47.682
1868	37.300	16.804	20.496	44.681
1869	73.900	55.213	18.687	83.090
1870	-	83.723	-	111.600
1871	-	77.213	-	124.460
1872	-	63.464	-	120.210
1873	-	50.100	-	109.386
1874	-	41.380	-	100.757

arima



[more...](#)

Fits a univariate seasonal or non-seasonal ARIMA time series model with optional regression variables.

Synopsis

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

```
float *imsls_f_arima (int n_obs, float y[], int model[], ..., 0)
```

The type *double* function is `imsls_d_arima`.

Required Arguments

int `n_obs` (Input)

Number of observations.

float `y[]` (Input)

Array of length `n_obs` containing the observations.

int `model[]` (Input)

Array of length 3 containing the model order parameters p , d , q .

	Description
0	Order of the autoregressive part, p , where $p \geq 0$.
1	Order of the non-seasonal difference operator, d , where $d \geq 0$.
2	Order of the moving average part, q , where $q \geq 0$.

Return Value

Pointer to an array of length $1 + p + q + P + Q$ with the estimated constant, autoregressive (AR), and moving average (MA) parameters.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_arima (int n_obs, float y[], int model[],

    IMSLS_SEASONAL_MODEL, int sper, int seasonal_model[],

    IMSLS_REGRESSION, int n_cols, float x[],

    IMSLS_REGRESSION_INDICES, int n_regressors, int indices[],

    IMSLS_REGRESSION_FORECASTS, float xlead[],

    IMSLS_INITIAL_EST_INPUT, int n_params, float params[],

    IMSLS_NO_TREND,

    IMSLS_MAX_ITERATIONS, int max_iterations,

    IMSLS_PRINT_LEVEL, int iprint,

    IMSLS_DIFFERENCES, float ** differences,

    IMSLS_DIFFERENCES_USER, float differences[],

    IMSLS_FORECASTS, int n_predict,
        float **forecasts, float **forecast_variances,

    IMSLS_FORECASTS_USER, int n_predict,
        float forecasts[], float forecast_variances[],

    IMSLS_REGRESSION_COEF, float **coefficients,

    IMSLS_REGRESSION_COEF_USER, float coefficients[],

    IMSLS_SE_ARMA, float **arma_std_errors,

    IMSLS_SE_ARMA_USER, float arma_std_errors[],

    IMSLS_VAR_NOISE, float *avar,

    IMSLS_SE_COEF, float **regcoef_std_errors,

    IMSLS_SE_COEF_USER, float regcoef_std_errors[],

    IMSLS_COEF_COVARIANCES, float **coef_covar,

    IMSLS_COEF_COVARIANCES_USER, float coef_covar[],

    IMSLS_AIC, float *aic,
```

`IMSLS_LOG_LIKELIHOOD`, *float* *`log_likelihood`,
`IMSLS_RETURN_USER`, *float* *`constant`, *float* `ar` [], *float* `ma` [],
 0)

Optional Arguments

`IMSLS_SEASONAL_MODEL`, *int* `sper`, *int* `seasonal_model` [] (Input)

Argument `sper` specifies the seasonal period, with `sper` ≥ 0 . Argument `seasonal_model` is an array of length 3 containing the seasonal model order parameters P , D , Q .

	Description
0	Order of the autoregressive part, P , where $P \geq 0$.
1	Order of the non-seasonal difference operator, D , where $D \geq 0$.
2	Order of the moving average part, Q , where $Q \geq 0$.

Default: $P = D = Q = 0$.

`IMSLS_REGRESSION`, *int* `n_cols`, *float* `x` [] (Input)

Array of length `n_obs` \times `n_cols` containing the regression variables. Specific columns of `x` may be selected using the optional argument `IMSLS_REGRESSION_INDICES`.

If optional arguments `IMSLS_FORECASTS` and `IMSLS_REGRESSION` are present, then `IMSLS_REGRESSION_FORECASTS` is required.

Default: `n_cols` = 0 (No regression variables are included).

`IMSLS_REGRESSION_INDICES`, *int* `n_regressors`, *int* `indices` [] (Input)

Argument `n_regressors` specifies the length of array `indices` and the number of regression variables to be included in the ARIMA fit. Argument `indices` contains the indices of the regression variables in matrices `x` and `xlead`.

Default: All regression variables in `x` and `xlead` will be used.

`IMSLS_REGRESSION_FORECASTS`, *float* `xlead` [] (Input)

Array of length `n_predict` \times `n_cols` containing the regression variables to be used in obtaining forecasts. Specific columns of `xlead` may be selected using the optional argument `IMSLS_REGRESSION_INDICES`.

If optional arguments `IMSLS_FORECASTS` and `IMSLS_REGRESSION` are present, then `IMSLS_REGRESSION_FORECASTS` is required.

Default: Not used.

IMSLS_INITIAL_EST_INPUT, *int n_params*, *float params[]* (Input)

Array of length `n_params` containing initial estimates for the parameters. Note that `n_params` must be equal to $p+q+P+Q$. The order of each parameter must be as stated, i.e. first the p autoregressive (AR) and q moving average (MA) parameters, followed by the P seasonal AR and Q seasonal MA parameters.

Default: Initial estimates are set by the program.

IMSLS_NO_TREND, (Input)

If **IMSLS_NO_TREND** is specified, the function will not include a trend variable. A trend variable has the effect of fitting an intercept term in the regression. If the difference operator `model[1] = d > 0`, the effect of trend on the model in the original, undifferenced space is polynomial of order d .

Default: The function includes a trend variable.

IMSLS_MAX_ITERATIONS, *int max_iterations* (Input)

Maximum number of iterations.

Default: `max_iterations = 100`

IMSLS_PRINT_LEVEL, *int iprint* (Input)

Printing option.

<code>iprint</code>	Action
0	No printing
1	Prints final results only.
2	Prints intermediate and final results.

Default: `iprint = 0`

IMSLS_DIFFERENCES, *float **differences* (Output)

Address of a pointer to an internally allocated array of length `n_obs - model[1] - seasonal_model[1]*sper` containing the differenced series.

IMSLS_DIFFERENCES_USER, *float differences[]* (Output)

Storage for the array `differences` is provided by user.

IMSLS_FORECASTS, *int n_predict, float **forecasts, float **forecast_variances* (Output)

Addresses of pointers to internally allocated arrays of length `n_predict` containing the forecasts and forecast variances for time points $t = n+1, n+2, \dots, n+n_predict$, where $n = n_obs$.

If optional arguments **IMSLS_FORECASTS** and **IMSLS_REGRESSION** are present, then **IMSLS_REGRESSION_FORECASTS** is required.

IMSLS_FORECASTS_USER, *int n_predict, float forecasts[], float forecast_variances[]* (Output)

Storage arrays `forecasts` and `forecast_variances` are provided by the user. See **IMSLS_FORECASTS**.

IMSLS_REGRESSION_COEF, *float **coefficients* (Output)

Address of a pointer to an internally allocated array of length `n_regressors+t` containing the estimated regression coefficients, where $t = 0$ if **IMSLS_NO_TREND** is specified, otherwise $t = 1$.

IMSLS_REGRESSION_COEF_USER, *float coefficients[]* (Output)

Storage array `coefficients` is provided by user. See **IMSLS_REGRESSION_COEF**.

IMSLS_SE_ARMA, *float **arma_std_errors* (Output)

Address of a pointer to an internally allocated array of length $p+q+P+Q$ containing the standard errors of the ARMA parameter estimates, where $p = \text{model}[0]$, $q = \text{model}[2]$, $P = \text{seasonal_model}[0]$ and $Q = \text{seasonal_model}[2]$.

IMSLS_SE_ARMA_USER, *float arma_std_errors[]* (Output)

Storage array `arma_std_errors` is provided by user. See **IMSLS_SE_ARMA**.

IMSLS_VAR_NOISE, *float *avar* (Output)

White noise variance estimate. For a regression-only model ($p=q=P=Q=0$), `avar` is the mean squared regression residual for `w_t`, the time series after applying any differencing.

IMSLS_SE_COEF, *float **regcoef_std_errors* (Output)

Address of a pointer to an internally allocated array of length `n_regressors+ t` containing the standard errors of the regression coefficients, where $t = 0$ if **IMSLS_NO_TREND** is specified, otherwise $t = 1$.

IMSLS_SE_COEF_USER, *float regcoef_std_errors[]* (Output)

Storage array `regcoef_std_errors` is provided by user. See **IMSLS_SE_COEF**.

IMSLS_COEF_COVARIANCES, *float **coef_covar* (Output)

Address of a pointer to an internally allocated array of length $(n_regressors+t) \times (n_regressors+t)$ containing the variances and covariances of the regression coefficients, where $t = 0$ if **IMSLS_NO_TREND** is specified, otherwise $t = 1$.

IMSL_S_COEF_COVARIANCES_USER, *float* `coef_covar[]` (Output)

Storage array `coef_covar` is provided by user. See `IMSL_S_COEF_COVARIANCES`.

IMSL_S_AIC, *float* `*aic` (Output)

Akaike's Information Criterion for the fitted ARMA model.

IMSL_S_LOG_LIKELIHOOD, *float* `*log_likelihood` (Output)

Value of $\ln(\textit{likelihood})$ for fitted model.

IMSL_S_RETURN_USER, *float* `*constant`, *float* `ar[]`, *float* `ma[]` (Output)

If specified, `constant` is the constant parameter estimate, `ar` is an array of length $p+P$ containing the final autoregressive parameter estimates, and `ma` is an array of length $q+Q$ containing the final moving average parameter estimates.

Description

Function `imsls_f_arima` fits a seasonal or non-seasonal ARIMA time series with the possible inclusion of one or more regression variables.

Suppose Y_t , $t = 1, \dots, N$, is a time series such that the d -th difference is stationary. Further, suppose a_t is a series of uncorrelated, mean 0 random variables with variance σ_a^2 .

First, for the non-seasonal case, the Auto-Regressive Integrated Moving Average (ARIMA) model for $\{Y_t, a_t\}$ can be expressed as

$$\phi(B)(1 - B)^d Y_t = \theta(B)a_t$$

where B is the backshift operator, $Bz_t = z_{t-1}$, $B^2 z_t = z_{t-2}$,

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 + \dots - \phi_p B^p,$$

and

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \dots - \theta_q B^q$$

The notation for this model is ARIMA(p, d, q) where p is the order of the autoregressive polynomial $\phi(B)$, d is the order of the differencing needed to make Y_t stationary, and q is the order of the moving-average polynomial $\theta(B)$.

The ARIMA model can be extended to include K regression variables $X_{1t}, X_{2t}, \dots, X_{Kt}$, by using the residuals (of the multiple regression of Y_t on $X_{1t}, X_{2t}, \dots, X_{Kt}$) in place of Y_t in the above ARIMA model:

$$\phi(B)(1-B)^d \left(Y_t - \beta_0 - \sum_{i=1}^K \beta_i X_{it} \right) = \theta(B) a_t$$

Equivalently,

$$\phi(B)w_t = \theta(B)a_t$$

where

$$w_t = (1-B)^d \left(Y_t - \beta_0 - \sum_{i=1}^K \beta_i X_{it} \right)$$

is the differenced residual series. Note that if `IMSLS_NO_TREND` is specified, β_0 will be 0.

To estimate the $p + q + K$ (and $+ 1$ for trend) parameters of the specified regression ARIMA model, `imsls_f_arima` uses the iterative generalized least squares method (IGLS) as described in Otto, Bell, and Burman (1987).

The IGLS method iterates between two steps, one step to estimate the regression parameters via generalized least squares (GLS) and the second step to estimate the ARMA parameters. In particular, at iteration m , the first step finds

$$\hat{\beta}_m = (\hat{\beta}_{m0}, \hat{\beta}_{m1}, \dots, \hat{\beta}_{mK})'$$

by solving the GLS problem with weight matrix

$$V(i,j) = \gamma_w(|j-i|), \quad i, j = 1, \dots, N$$

where

$$\gamma_w(j-i) = E[w_{t-j} w_{t-i} | \hat{\phi}_{m-1}, \hat{\theta}_{m-1}]$$

That is, $\hat{\beta}_m$ minimizes $(Y - X\beta)' V^{-1} (Y - X\beta)$, where $Y = (Y_1, \dots, Y_N)'$, X is an N by $K + 1$ matrix with i -th column, $X_i = (X_{i1}, \dots, X_{iN})'$, $i = 1, \dots, K$, and $X_0 = (1, \dots, 1)'$, and V is an N by N weight matrix defined using the theoretical autocovariances of the series

$$w_{m-1,t} = (1 - B)^d \left(Y_t - \beta_{m-1,0} - \sum_{i=1}^K \hat{\beta}_{m-1,i} X_{it} \right)$$

The series $w_{m-1,t}$ is modeled as an ARMA(p,q) process with parameters $\hat{\phi}_{m-1} = (\hat{\phi}_{m-1,1}, \dots, \hat{\phi}_{m-1,p})'$ and $\hat{\theta}_{m-1} = (\hat{\theta}_{m-1,1}, \dots, \hat{\theta}_{m-1,q})'$. At iteration m , the second step is then to obtain new estimates of $\hat{\phi}_m$ and $\hat{\theta}_m$ for the updated series, $w_{m,t}$. The function then finds the maximum likelihood estimates, $\hat{\phi}_m$ and $\hat{\theta}_m$, by minimizing the negative log-likelihood over the parameter vector. The log-likelihood is calculated using the innovations algorithm as described in [Brockwell & Davis](#), Chapter 8.

The method is essentially the same for a seasonal ARIMA model. The notation for a seasonal model is ARIMA(p, d, q)(P, D, Q) $_s$, where P is the order of the seasonal autoregressive polynomial, D (along with d) is the order of the differencing needed for stationarity, Q is the order of the seasonal moving-average polynomial, and s is the seasonal period. In particular, the seasonal ARIMA model has the form

$$\phi(B) \Phi(B^s) w_t = \theta(B) \Theta(B^s) a_t$$

where $w_t = (1 - B)^d (1 - B^s)^D Y_t$, $d, D \geq 0$ are the difference orders, $s \geq 1$ is an integer specifying the seasonal period, and a_t is the white noise process as before. The seasonal autoregressive and moving average polynomials are given by

$$\Phi(B^s) = 1 - \Phi_1 B^s - \Phi_2 B^{2s} - \dots - \Phi_p B^{ps}$$

and

$$\Theta(B^s) = 1 - \Theta_1 B^s - \Theta_2 B^{2s} - \dots - \Theta_Q B^{Qs}$$

respectively.

Remarks

When forecasts are requested (`n_predict > 0`), `imsls_f_arima` requires that future values of the independent variables be provided in optional argument `IMSLS_REGRESSION_FORECASTS`. In effect, `imsls_f_arima` assumes the future X 's are known without error, which is valid for any deterministic function of time such as a seasonal indicator. Also, in economics, certain factors that are considered to be leading indicators are treated as deterministic for the purpose of predicting changes in the economy. Users may consider using a more general transfer function model if this is an unreasonable assumption. Function `imsls_f_arima` calculates forecast variances using the asymptotic result found in [Fuller \(1996\)](#), Theorem 2.9.4. To obtain the standard errors of the ARMA parameters, `imsls_f_arima` calls function `imsls_f_arma` for the final w series.

Examples

Example 1

The data set consists of annual mileage per passenger vehicle and annual US population (in 10000's) spanning the years 1980 to 2006 (U.S. Energy Information Administration, 2008). Consider modeling the annual mileage using US population as a regression variable.

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

int main()
{
    int nobs = 24;
    int model[3] = { 1, 0, 0 };
    int n_predict = 5;
    float avar, llike, *result = NULL, *regcoef = NULL, *regstderr = NULL;
    float *coefcovar = NULL, *armastderr = NULL, *fcst = NULL;
    float *fcst_var = NULL;

    float y[29] = {
        9062.0, 8813.0, 8873.0, 9050.0, 9118.0,
        9248.0, 9419.0, 9464.0, 9720.0, 9972.0,
        10157.0, 10504.0, 10571.0, 10857.0, 10804.0,
        10992.0, 11203.0, 11330.0, 11581.0, 11754.0,
        11848.0, 11976.0, 11831.0, 12202.0, 12325.0,
        12460.0, 12510.0, 12485.0, 12293.0
    };

    float regX[29] = { 22722.4681, 22946.5714, 23166.4458,
        23379.1990, 23582.4902, 23792.3795, 24013.2887,
        24228.8918, 24449.8982, 24681.923, 24962.2814,
        25298.0941, 25651.4224, 25991.8588, 26312.5821,
        26627.8393, 26939.4284, 27264.6925, 27585.4104,
        27904.0168, 28217.1936, 28503.9803, 28772.6647,
        29021.0914, 29289.2127, 29556.0549, 29836.2973,
        30129.0332, 30405.9724
    };

    result = imsls_f_arima(nobs, y, model,
        IMSLS_REGRESSION, 1, regX,
        IMSLS_REGRESSION_FORECASTS, &regX[nobs],
        IMSLS_FORECASTS, n_predict, &fcst, &fcst_var,
        IMSLS_VAR_NOISE, &avar,
        IMSLS_LOG_LIKELIHOOD, &llike,
        IMSLS_REGRESSION_COEF, &regcoef,
        IMSLS_SE_COEF, &regstderr,
        IMSLS_COEF_COVARIANCES, &coefcovar,
        IMSLS_SE_ARMA, &armastderr,
        IMSLS_PRINT_LEVEL, 1,
        0);

    if (result) imsls_free(result);
    if (fcst) imsls_free(fcst);
    if (fcst_var) imsls_free(fcst_var);
    if (regcoef) imsls_free(regcoef);
}
```

```

    if (regstderr) imsls_free(regstderr);
    if (coefcovar) imsls_free(coefcovar);
    if (armastderr) imsls_free(armastderr);
}

```

Output

```
Final results for ARIMA model (p,d,q)x(P,D,Q)_s = (1,0,0)x(0,0,0)_0
```

```
Final AR parameter estimates/ std errors
      0.56471      0.13500
```

```
-2*ln(maximum log likelihood) = 299.944427
```

```
White noise variance = 15425.566406
```

```
Regression estimates:
```

	COEFFICIENTS	Regression STD Errors
0	-3480.87573	689.33929
1	0.54236	0.02674

```
Forecasts with standard deviation
```

T	Y fcst	Y fcst std dev
24	12368.37207	124.19970
25	12524.92871	142.63528
26	12683.60254	148.03239
27	12846.14355	149.71263
28	12998.47363	150.24451

Example 2

The data set consists of simulated weekly observations containing a strong annual seasonality. The seasonal variables are constructed and sent into `imsls_f_arima` as regression variables.

```

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

int main()
{
    int nobs = 100, n_predict = 4, n_regressors = 2;
    int i, model[3] = { 2, 0, 0 };
    float PI, *coefcovar = NULL, *regcoef = NULL, *regstderr = NULL;
    float *result = NULL, *armastderr = NULL, *fcst = NULL;
    float *fcstvar = NULL;
    float avar, llike;
    float x[104][2];

    float y[104] = {
        32.27778, 32.63300, 33.13768, 34.4517,
        34.63824, 37.31262, 37.35704, 37.03092,
        36.39894, 35.75541, 35.10829, 34.70107,
        34.69592, 32.75326, 30.85370, 31.10936,

```

```

29.47493, 29.14361, 28.50466, 30.09714,
28.49403, 27.23268, 23.49674, 22.71225,
21.42798, 18.68601, 17.40035, 16.06832,
15.31862, 14.75179, 13.40089, 13.01101,
12.44863, 11.27890, 11.51770, 14.31982,
14.67036, 14.76331, 15.35644, 17.04353,
18.39931, 18.21919, 18.72777, 19.61794,
22.31733, 23.79600, 25.41326, 25.60497,
27.93579, 29.21765, 29.60981, 28.46994,
28.78081, 30.96402, 35.49537, 35.75124,
36.18933, 37.2627, 35.02454, 33.57089,
35.00683, 34.83886, 34.19827, 33.73966,
34.49709, 34.07127, 32.74709, 31.97856,
31.3029, 30.21916, 27.46015, 26.78431,
25.32815, 23.97863, 21.83837, 21.00647,
20.58846, 19.94578, 17.38271, 17.12572,
16.71847, 17.45425, 16.15050, 13.07448,
12.54188, 12.42137, 13.51771, 14.84232,
14.28870, 13.39561, 15.48938, 16.47175,
17.62758, 16.57677, 18.20737, 20.8491,
20.15616, 20.93857, 23.73973, 25.30449,
26.51106, 29.43261, 32.02672, 32.18846
};

/*
 * The data are simulated weekly observations
 * with an annual seasonal cycle
 */
PI = acos(-1.0);

for (i = 0; i < nobs + n_predict; i++)
{
    x[i][0] = sin(2.0 * PI * i / 52.0);
    x[i][1] = cos(2.0 * PI * i / 52.0);
}

result = imsls_f_arima(nobs, y, model,
    IMSLS_REGRESSION, 2, x,
    IMSLS_REGRESSION_FORECASTS, &x[100][0],
    IMSLS_FORECASTS, n_predict, &fcst, &fcstvar,
    IMSLS_VAR_NOISE, &avar,
    IMSLS_LOG_LIKELIHOOD, &llike,
    IMSLS_REGRESSION_COEF, &regcoef,
    IMSLS_SE_COEF, &regstderr,
    IMSLS_COEF_COVARIANCES, &coefcovar,
    IMSLS_SE_ARMA, &armastderr,
    IMSLS_PRINT_LEVEL, 1,
    0);

if (coefcovar) imsls_free(coefcovar);
if (regcoef) imsls_free(regcoef);
if (regstderr) imsls_free(regstderr);
if (result) imsls_free(result);
if (armastderr) imsls_free(armastderr);
if (fcst) imsls_free(fcst);
if (fcstvar) imsls_free(fcstvar);
}

```

Output

Final results for ARIMA model (p,d,q)x(P,D,Q)_s = (2,0,0)x(0,0,0)_0

Final AR parameter estimates/ std errors

0.71727	0.09837
-0.26694	0.09828

-2*ln(maximum log likelihood) = 270.166840

White noise variance = 0.868012

Regression estimates:

	COEFFICIENTS	Regression STD Errors
0	24.81011	0.17172
1	8.91972	0.24034
2	6.84813	0.24701

Forecasts with standard deviation

T	Y fcst	Y fcst std dev
100	26.74694	0.93167
101	28.07994	1.14655
102	29.33769	1.16952
103	30.53127	1.16959

Example 3

The data set consists of the number of monthly accidental deaths in the USA from 1973 to 1978 (Brockwell & Davis, 2006). The following example fits a (0,1,1)x(0,1,1) ARIMA model with seasonal period = 12. With monthly data, a seasonal period of 12 corresponds to an annual seasonal cycle.

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

int main()
{
    int nobs = 72, iprint = 1;
    int model[3] = { 0, 1, 1 };
    int seas_model[3] = { 0, 1, 1 };
    int sper = 12;
    int n_predict = 24;
    float params[2] = { .25, .25 };
    float avar, llike;
    float *ses = NULL, *result = NULL;

    float deaths[72] = {
        9007, 8106, 8928, 9137, 10017, 10826, 11317, 10744, 9713, 9938,
        9161, 8927, 7750, 6981, 8038, 8422, 8714, 9512, 10120, 9823,
        8743, 9129, 8710, 8680, 8162, 7306, 8124, 7870, 9387, 9556,
        10093, 9620, 8285, 8433, 8160, 8034, 7717, 7461, 7776, 7925,
        8634, 8945, 10078, 9179, 8037, 8488, 7874, 8647, 7792, 6957,
        7726, 8106, 8890, 9299, 10625, 9302, 8314, 8850, 8265, 8796,
        7836, 6892, 7791, 8129, 9115, 9434, 10484, 9827, 9110, 9070,
```

```
      8633, 9240
    };

    result = imsls_f_arima(nobs, deaths, model,
        IMSLS_SEASONAL_MODEL, sper, seas_model,
        IMSLS_LOG_LIKELIHOOD, &llike,
        IMSLS_NO_TREND,
        IMSLS_INITIAL_EST_INPUT, 2, params,
        IMSLS_VAR_NOISE, &avar,
        IMSLS_PRINT_LEVEL, iprint,
        IMSLS_SE_ARMA, &ses, 0);

    if (result) imsls_free(result);
    if (ses) imsls_free(ses);
}
```

Output

```
Final results for ARIMA model (p,d,q)x(P,D,Q)_s = (0,1,1)x(0,1,1)_12

Final MA parameter estimates/ std errors
      0.42643          0.11604

Final Seasonal MA parameter estimates/ std errors
      0.55822          0.13508

-2*ln(maximum log likelihood) = 851.065125

White noise variance = 99502.765625
```

regression_arima



[more...](#)

Fits a univariate ARIMA (p, d, q) time series model with the inclusion of one or more regression variables.

Synopsis

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

```
float *imsls_f_regression_arima (int n_obs, float y[ ], int model[ ], ..., 0)
```

The type *double* function is `imsls_d_regression_arima`.

Required Arguments

int `n_obs` (Input)

Number of observations.

float `y[]` (Input)

Array of length `n_obs` containing the observations.

int `model[]` (Input)

Array of length 3 containing the model order parameters p, d, q .

	Description
0	Order of the autoregressive part, p , where $p \geq 0$.
1	Order of the non-seasonal difference operator, d , where $d \geq 0$.
2	Order of the moving average part, q , where $q \geq 0$.

Return Value

Pointer to an array of length $1 + p + q$ with the estimated constant, autoregressive (AR), and moving average (MA) parameters.

Synopsis with Optional Arguments

```

#include <imsls.h>

float *imsls_f_regression_arima (int n_obs, float y[], int model[],

    IMSLS_REGRESSION, int n_regressors, float x[],

    IMSLS_REGRESSION_FORECASTS, float xlead[],

    IMSLS_REGRESSION_INDICES, int n_indices, int indices[],

    IMSLS_NO_TREND,

    IMSLS_MAX_ITERATIONS, int max_iterations,

    IMSLS_PRINT_LEVEL, int iprint,

    IMSLS_FORECASTS, int n_predict,
        float **forecasts, float **forecast_variances,

    IMSLS_FORECASTS_USER, int n_predict,
        float forecasts[], float forecast_variances[],

    IMSLS_REGRESSION_COEF, float **coefficients,

    IMSLS_REGRESSION_COEF_USER, float coefficients[],

    IMSLS_SE_ARMA, float **arma_std_errors,

    IMSLS_SE_ARMA_USER, float arma_std_errors[],

    IMSLS_VAR_NOISE, float *avar,

    IMSLS_SE_COEF, float **regcoef_std_errors,

    IMSLS_SE_COEF_USER, float regcoef_std_errors[],

    IMSLS_COEF_COVARIANCES, float **coef_covar,

    IMSLS_COEF_COVARIANCES_USER, float coef_covar[],

    IMSLS_AIC, float *aic,

    IMSLS_LOG_LIKELIHOOD, float *log_likelihood,

    IMSLS_RETURN_USER, float *constant, float ar[], float ma[],

    0)

```

Optional Arguments

`IMSLS_REGRESSION`, *int* `n_regressors`, *float* `x[]` (Input)

Array of length `n_obs × n_regressors` containing the regression variables. Specific columns of `x` may be selected using the optional argument `IMSLS_REGRESSION_INDICES`.

If optional arguments `IMSLS_FORECASTS` and `IMSLS_REGRESSION` are present, then `IMSLS_REGRESSION_FORECASTS` is required.

Default: `n_regressors = 0` (No regression variables are included).

`IMSLS_REGRESSION_FORECASTS`, *float* `xlead[]` (Input)

Array of length `n_predict × n_regressors` containing the regression variables to be used in obtaining forecasts. Specific columns of `xlead` may be selected using the optional argument `IMSLS_REGRESSION_INDICES`.

If optional arguments `IMSLS_FORECASTS` and `IMSLS_REGRESSION` are present, then `IMSLS_REGRESSION_FORECASTS` is required.

Default: Not used.

`IMSLS_REGRESSION_INDICES`, *int* `n_indices`, *int* `indices[]` (Input)

Argument `n_indices` specifies the length of array `indices` and the number of regression variables to be included in the ARIMA fit. Argument `indices` contains the indices of the regression variables in matrices `x` and `xlead`.

Default: All regression variables in `x` and `xlead` will be used.

`IMSLS_NO_TREND`, (Input)

If `IMSLS_NO_TREND` is specified, the function will not include a trend variable. A trend variable has the effect of fitting an intercept term in the regression. If the difference operator `model[1] = d > 0`, the effect of no trend on the model in the original, undifferenced space is polynomial of order `d`.

Default: The function will include a trend variable.

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)

Maximum number of iterations.

Default: `max_iterations = 50`

IMSLS_PRINT_LEVEL, *int* `iprint` (Input)
 Printing option.

<code>iprint</code>	Action
0	No printing
1	Prints final results only.
2	Prints intermediate and final results.

Default: `iprint = 0`

IMSLS_FORECASTS, *int* `n_predict`, *float *** `forecasts`, *float *** `forecast_variances` (Output)
 Addresses of pointers to internally allocated arrays of length `n_predict` containing the forecasts and forecast variances for time points $t = n+1, n+2, \dots, n+n_predict$, where $n = n_obs$.

If optional arguments IMSLS_FORECASTS and IMSLS_REGRESSION are present, then IMSLS_REGRESSION_FORECASTS is required.

IMSLS_FORECASTS_USER, *int* `n_predict`, *float* `forecasts[]`, *float* `forecast_variances[]` (Output)
 Storage arrays `forecast` and `forecast_variance` are provided by user. See IMSLS_FORECASTS.

IMSLS_REGRESSION_COEF, *float *** `coefficients` (Output)
 Address of a pointer to an internally allocated array of length `n_regressors+t` containing the estimated regression coefficients, where $t = 0$ if IMSLS_NO_TREND is specified, otherwise $t = 1$.

IMSLS_REGRESSION_COEF_USER, *float* `coefficients[]` (Output)
 Storage array `coefficients` is provided by user. See IMSLS_REGRESSION_COEF.

IMSLS_SE_ARMA, *float *** `arma_std_errors` (Output)
 Address of a pointer to an internally allocated array of length $p+q$ containing the standard errors of the ARMA parameter estimates, where $p = \text{model}[0]$ and $q = \text{model}[2]$.

IMSLS_SE_ARMA_USER, *float* `arma_std_errors[]` (Output)
 Storage array `arma_std_errors` is provided by user. See IMSLS_SE_ARMA.

IMSLS_VAR_NOISE, *float ** `avar` (Output)
 White noise variance estimate. If `model[0]+model[2]=0` and `n_regressors > 0`, `avar` is the mean squared regression residual.

IMSLs_SE_COEF, *float **regcoef_std_error* (Output)

Address of a pointer to an internally allocated array of length `n_regressors + t` containing the standard errors of the ARMA parameter estimates, where $t = 0$ if `IMSLs_NO_TREND` is specified, otherwise $t = 1$.

IMSLs_SE_COEF_USER, *float regcoef_std_errors[]* (Output)

Storage array `regcoef_std_errors` is provided by user. See `IMSLs_SE_COEF`.

IMSLs_COEF_COVARIANCES, *float **coef_covar* (Output)

Address of a pointer to an internally allocated array of length $(n_regressors + t) \times (n_regressors + t)$ containing the variances and covariances of the regression coefficients, where $t = 0$ if `IMSLs_NO_TREND` is specified, otherwise $t = 1$.

IMSLs_COEF_COVARIANCES_USER, *float coef_covar[]* (Output)

Storage array `coef_covar` is provided by user. See `IMSLs_COEF_COVARIANCES`.

IMSLs_AIC, *float *aic* (Output)

Akaike's Information Criterion for the fitted ARMA model.

IMSLs_LOG_LIKELIHOOD, *float *log_likelihood* (Output)

Value of $-2(\ln(\text{likelihood}))$ for fitted model.

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

Function `imsls_f_regression_arima` fits an ARIMA(p, d, q) to a univariate time series with the possible inclusion of one or more regression variables.

Suppose $Y_t, t = 1, \dots, N$, is a time series such that the d -th difference is stationary. Further, suppose a_t is a series of uncorrelated, mean 0 random variables with variance σ_a^2 .

The Auto-Regressive Integrated Moving Average (ARIMA) model for $\{Y_t, a_t\}$ can be expressed as

$$\phi(B)(1 - B)^d Y_t = \theta(B)a_t$$

where B is the backshift operator, $Bz_t = z_{t-1}$, $B^2 z_t = z_{t-2}$,

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 + \dots - \phi_p B^p,$$

and

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \dots - \theta_q B^q$$

The notation for this model is ARIMA(p, d, q) where p is the order of the autoregressive polynomial $\phi(B)$, d is the order of the differencing needed to make Y_t stationary, and q is the order of the moving-average polynomial $\theta(B)$.

The ARIMA model can be extended to include K regression variables $X_{1t}, X_{2t}, \dots, X_{Kt}$, by using the residuals (of the multiple regression of Y_t on $X_{1t}, X_{2t}, \dots, X_{Kt}$) in place of Y_t in the above ARIMA model.

$$\phi(B)(1-B)^d \left(Y_t - \beta_0 - \sum_{i=1}^K \beta_i X_{it} \right) = \theta(B) a_t$$

Equivalently,

$$\phi(B) w_t = \theta(B) a_t$$

where

$$w_t = (1-B)^d \left(Y_t - \beta_0 - \sum_{i=1}^K \beta_i X_{it} \right)$$

is the differenced residual series.

To estimate the $(p + q + K)$ parameters of the specified regression ARIMA model, `imsls_f_regression_arima` uses the iterative generalized least squares method (IGLS) as described in Otto, Bell, and Burman (1987).

The IGLS method iterates between two steps, one step to estimate the regression parameters via generalized least squares (GLS) and the second step to estimate the ARMA parameters. In particular, at iteration m , the first step finds

$$\hat{\beta}_m = (\hat{\beta}_{m0}, \hat{\beta}_{m1}, \dots, \hat{\beta}_{mK})'$$

by solving the GLS problem with weight matrix

$$V(i, j) = \gamma_w(|j - i|), \quad i, j = 1, \dots, N$$

where

$$\gamma_w(j-i) = E[w_{t-j}w_{t-i}|\hat{\phi}_{m-1}, \hat{\theta}_{m-1}]$$

That is, $\hat{\beta}_m$ minimizes $(Y - X\beta)'V^{-1}(Y - X\beta)$, where $Y = (Y_1, \dots, Y_N)'$, X is an N by K matrix with i -th column, $X_i = (X_{i1}, \dots, X_{iN})'$, $i = 1, \dots, K$, and $X_0 = (1, \dots, 1)'$, and V is an N by N weight matrix defined using the theoretical autocovariances of the series

$$w_{m-1,t} = (1 - B)^d \left(Y_t - \beta_{m-1,0} - \sum_{i=1}^K \hat{\beta}_{m-1,i} X_{it} \right)$$

The series $w_{m-1,t}$ is modeled as an ARMA(p,q) process with parameters $\hat{\phi}_{m-1} = (\hat{\phi}_{m-1,1}, \dots, \hat{\phi}_{m-1,p})'$ and $\hat{\theta}_{m-1} = (\hat{\theta}_{m-1,1}, \dots, \hat{\theta}_{m-1,q})'$. At iteration m , the second step is then to obtain new estimates of $\hat{\phi}_m$ and $\hat{\theta}_m$ for the updated series, $w_{m,t}$. To find the estimates $\hat{\phi}_m$ and $\hat{\theta}_m$, `imsls_f_regression_arima` uses the exact likelihood method as described in Akaike, Kitagawa, Arahata and Tada (1979) and used in function, [imsls_f_max_arma](#).

Remarks

When forecasts are requested (`n_predict > 0`), `imsls_f_regression_arima` requires that future values of the independent variables be provided in optional argument `IMSLS_REGRESSION_FORECASTS`. In effect, `imsls_f_regression_arima` assumes the future X 's are known without error, which is valid for any deterministic function of time such as a seasonal indicator. Also, in economics, certain factors that are considered to be leading indicators are treated as deterministic for the purpose of predicting changes in the economy. Users may consider using a more general transfer function model if this is an unreasonable assumption. Function `imsls_f_regression_arima` calculates forecast variances using the asymptotic result found in Fuller (1996), Theorem 2.9.4. To obtain the standard errors of the ARMA parameters, `imsls_f_regression_arima` calls function `imsls_f_arma` for the final w series.

Examples

Example 1

The data set consists of annual mileage per passenger vehicle and annual US population (in 1000's) spanning the years 1980 to 2006 (U.S. Energy Information Administration, 2008). Consider modeling the annual mileage using US population as a regression variable.

```

#include <imsls.h>
int main()
{
    int nobs= 24, model[3] = {1, 0, 0};
    int indices[1] = {0}, n_predict=5;
    float avar, llike, *result;
    float *regcoef, *regstderr, *coefcovar, *armastderr;
    float *fcst, *fcst_var;
    float y[29] = {
        9062.0, 8813.0, 8873.0, 9050.0, 9118.0,
        9248.0, 9419.0, 9464.0, 9720.0, 9972.0,
        10157.0, 10504.0, 10571.0, 10857.0, 10804.0,
        10992.0, 11203.0, 11330.0, 11581.0, 11754.0,
        11848.0, 11976.0, 11831.0, 12202.0, 12325.0,
        12460.0, 12510.0, 12485.0, 12293.0
    };
    float regX[29][2] = {
        {22722.4681, 9062.0},
        {22946.5714, 8813.0},
        {23166.4458, 8873.0},
        {23379.1990, 9050.0},
        {23582.4902, 9118.0},
        {23792.3795, 9248.0},
        {24013.2887, 9419.0},
        {24228.8918, 9464.0},
        {24449.8982, 9720.0},
        {24681.923, 9972.0},
        {24962.2814, 10157.0},
        {25298.0941, 10504.0},
        {25651.4224, 10571.0},
        {25991.8588, 10857.0},
        {26312.5820999999, 10804.0},
        {26627.8393, 10992.0},
        {26939.4284, 11203.0},
        {27264.6925, 11330.0},
        {27585.4104, 11581.0},
        {27904.0168, 11754.0},
        {28217.1936, 11848.0},
        {28503.9803, 11976.0},
        {28772.6647, 11831.0},
        {29021.0914, 12202.0},
        {29289.2127, 12325.0},
        {29556.0549, 12460.0},
        {29836.2973, 12510.0},
        {30129.0332, 12485.0},
        {30405.9724, 12293.0}
    };
};

result = imsls_f_regression_arima (nobs, y, model,
    IMSLS_REGRESSION,2, regX,
    IMSLS_REGRESSION_FORECASTS, &regX[24][0],
    IMSLS_FORECASTS, n_predict, &fcst, &fcst_var,
    IMSLS_REGRESSION_INDICES, 1, indices,
    IMSLS_VAR_NOISE, &avar,
    IMSLS_LOG_LIKELIHOOD, &llike,
    IMSLS_REGRESSION_COEF, &regcoef,
    IMSLS_SE_COEF, &regstderr,
    IMSLS_COEF_COVARIANCES, &coefcovar,

```

```

    IMSLS_SE_ARMA, &armastderr,
    IMSLS_PRINT_LEVEL, 1,
    0);
}

```

Output

Final results for regression ARIMA model (p,d,q) = 1, 0, 0s

Final AR parameter estimates/ std errors
 0.73000 0.13498

-2*ln(maximum log likelihood) = 231.835464

White noise variance = 15427.915039

Regression estimates:

	COEFFICIENTS	Regression STD Errors
0	-3483.13306	687.21167
1	0.54244	0.02666

Forecasts with standard deviation

T	Y fcst	Y fcst std dev
24	12360.51563	124.20916
25	12514.80664	153.78410
26	12673.78906	167.42434
27	12837.66895	174.25776
28	12991.60547	177.79208

Example 2

The data set consists of simulated weekly observations containing a strong annual seasonality. The seasonal variables are constructed and sent into `regression_arima` as regression variables.

```

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

int main()
{
    int nobs=100, n_predict=4, n_regressors=2;
    int i, model[3] = {2,0,0};
    float PI, *coefcovar, *regcoef, *regstderr, *result;
    float *armastderr, *fcst, *fcstvar;
    float avar, llike;
    float x[104][2];

    float y[104] = {
        32.27778, 32.63300, 33.13768, 34.4517,
        34.63824, 37.31262, 37.35704, 37.03092,
        36.39894, 35.75541, 35.10829, 34.70107,
        34.69592, 32.75326, 30.85370, 31.10936,
        29.47493, 29.14361, 28.50466, 30.09714,
        28.49403, 27.23268, 23.49674, 22.71225,
        21.42798, 18.68601, 17.40035, 16.06832,

```

```

15.31862, 14.75179, 13.40089, 13.01101,
12.44863, 11.27890, 11.51770, 14.31982,
14.67036, 14.76331, 15.35644, 17.04353,
18.39931, 18.21919, 18.72777, 19.61794,
22.31733, 23.79600, 25.41326, 25.60497,
27.93579, 29.21765, 29.60981, 28.46994,
28.780810, 30.96402, 35.49537, 35.75124,
36.18933, 37.2627, 35.02454, 33.57089,
35.00683, 34.83886, 34.19827, 33.73966,
34.49709, 34.07127, 32.74709, 31.97856,
31.3029, 30.21916, 27.46015, 26.78431,
25.32815, 23.97863, 21.83837, 21.00647,
20.58846, 19.94578, 17.38271, 17.12572,
16.71847, 17.45425, 16.15050, 13.07448,
12.54188, 12.42137, 13.51771, 14.84232,
14.28870, 13.39561, 15.48938, 16.47175,
17.62758, 16.57677, 18.20737, 20.8491,
20.15616, 20.93857, 23.73973, 25.30449,
26.51106, 29.43261, 32.02672, 32.18846
};

/*
* The data are simulated weekly observations
* with an annual seasonal cycle
*/
PI = imsl_f_constant("PI",0);

for (i=0; i<nobs+n_predict;i++)
{
    x[i][0] = sin(2*PI*i/ 52.0);
    x[i][1] = cos(2*PI*i/ 52.0);
}

result = imsls_f_regression_arima (nobs, y, model,
    IMSLS_REGRESSION,2, x,
    IMSLS_REGRESSION_FORECASTS, &x[100][0],
    IMSLS_FORECASTS, n_predict, &fcst, &fcstvar,
    IMSLS_VAR_NOISE, &avar,
    IMSLS_LOG_LIKELIHOOD, &llike,
    IMSLS_REGRESSION_COEF, &regcoef,
    IMSLS_SE_COEF, &regstderr,
    IMSLS_COEF_COVARIANCES, &coefcovar,
    IMSLS_SE_ARMA, &armastderr,
    IMSLS_PRINT_LEVEL, 1,
    0);
}

```

Output

```

Final AR parameter estimates/ std errors
      0.71855      0.09838
     -0.25989      0.09828

-2*ln(maximum log likelihood) = -13.621020

White noise variance = 0.868007

Regression estimates:

```

	COEFFICIENTS	Regression STD	Errors
0	24.81011	0.17177	
1	8.91971	0.24042	
2	6.84814	0.24709	
Forecasts with standard deviation			
T	Y fcst	Y fcst std dev	
100	26.74492	0.93167	
101	28.07804	1.14725	
102	29.33707	1.35615	
103	30.53160	1.52323	

auto_uni_ar



[more...](#)

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 $1 \leq \text{maxlag} \leq \text{n_obs} / 2$.
- int* ***p** (Output)
Number of autoregressive parameters in the model with minimum AIC.

Return Value

Vector of length $1 + \text{maxlag}$ containing the estimates for the constant and the autoregressive parameters in the model with minimum AIC. The estimates are located in the first $1 + \text{p}$ 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:

iprint	Action
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:

method	Action
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

$$AIC = -2 \ln(\text{likelihood}) + 2(p + 1)$$

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

$$AIC = (n_obs - maxlag) \ln(\hat{\sigma}^2) + 2(p + 1) + (n_obs - maxlag) (\ln(2\pi) + 1),$$

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

$$(n_obs - maxlag) (\ln(2\pi) + 1),$$

the calculation is simplified to

$$AIC = (n_obs - maxlag) \ln(\hat{\sigma}^2) + 2(p + 1)$$

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., $p = \text{maxlag}$, 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 $\text{method} = 0$, estimates of the autoregressive coefficients for the model with minimum AIC are calculated using method of moments. If $\text{method} = 1$, the coefficients are determined by the method of least squares applied in the form described by Kitagawa and Akaike (1978). Otherwise, if $\text{method} = 2$, 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 $\phi_0, \phi_1, \phi_2, \phi_3$ for each of the three parameter estimation methods available.

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

```

#include <stdio.h>

int 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)
    {
        imsls_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);
}

```

Output

```

AIC Automatic Order selection
AR coefficients estimated using method of moments
Order selected: 3
AIC =    633.0114, 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 =    633.0114, 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 = 633.0114, Variance = 218.8337
Constant estimate is 11.3902.

Final AR coefficients estimated by maximum likelihood

1	1.553
2	-1.001
3	0.205

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 $d_1, \dots, d_m, s_1, \dots, s_m$ ($m=n_differences$) and p in the AR model:

$$\phi_p(B) \left(\Delta_{s_1}^{d_1} \Delta_{s_2}^{d_2} \dots \Delta_{s_m}^{d_m} Z_t - \mu \right) = a_t$$

where $\{Z_t\}$ is the original time series, B is the backward shift operator defined by $B^k Z_t = Z_{t-k}$, $k \geq 0$, a_t is

Gaussian white noise with $E[a_t] = 0$ and $VAR[a_t] = \sigma^2$,

$\phi_p(B) = 1 - \phi_1 B - \phi_2 B^2 - \dots - \phi_p B^p$, $0 \leq p \leq \text{maxlag}$, $\Delta_s^d = (1 - B^s)^d$, with $s > 0, d \geq 0$, 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 (Input)

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, \dots, 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, \dots, 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

$$W_t(s,d) = \Delta_{s_1}^{d_1} \Delta_{s_2}^{d_2} \dots \Delta_{s_m}^{d_m} Z_t = \prod_{i=1}^m (1 - B^{s_i})^{d_i} Z_t = \prod_{i=1}^m \sum_{j=0}^{d_i} \binom{d_i}{j} (-1)^j B^{j-s_i} Z_t$$

where $s := (s_1, \dots, s_m)$, $d := (d_1, \dots, d_m)$ 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,

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

is identified.

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

int 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)
{
    imsls_free(s);
    s = NULL;
}

if (d)
{
    imsls_free(d);
    d = NULL;
}

if (z)
{
    imsls_free(z);
    z = NULL;
}

if (difference)
{
    imsls_free(difference);
    difference = NULL;
}
}
```

Output

```
nlost = 13
s = (1, 12)
d = (1, 1)
Order of optimum AR process: 1
aic = 949.780334
```

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

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), $0 < \text{delta} < 1$.

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. 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 $(p, 0, q) \times (0, d, 0)_S$:

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

Here, $\Delta_s^d = (1 - B^s)^d$, $\theta(B) = 1 - \theta_1 B - \dots - \theta_q B^q$, $\phi(B) = 1 - \phi_1 B - \dots - \phi_p B^p$. 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, \dots, 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, $I_t(t_j) = 1$, $I_t(t_j) = 0$ otherwise. **Note** that $L_j(B)$ operates on I_t via $B^k I_t = I_{t-k}$, $k = 0, 1, \dots$.

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 characterized by different values for $L_j(B)$:

1. $L_j(B) = \frac{\theta(B)}{\Delta_s^d \phi(B)}$ 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. In order to simulate a measurement error, the actual time series value at time point $t=30$, which is $0.25570e + 01$, was replaced by $0.35570e + 01$. Function `imsls_f_ts_outlier_identification` is used to fit an AR(2) model of the form

$(1 - \phi_1 B - \phi_2 B^2) Y_t = \theta_0 + a_t, t = 1, 2, \dots, 144, \{a_t\}$ Gaussian White noise, to the given series. Function

`imsls_f_ts_outlier_identification` computes parameters $\theta_0 = 1.052683$, $\phi_1 = 1.389253$ and $\phi_2 = -0.752184$ and identifies an additive outlier at time point $t=30$.

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

int main(){
    float series[144]={
        0.24300e+01,0.25060e+01,0.27670e+01,0.29400e+01,0.31690e+01,0.34500e+01,
        0.35940e+01,0.37740e+01,0.36950e+01,0.34110e+01,0.27180e+01,0.19910e+01,
        0.22650e+01,0.24460e+01,0.26120e+01,0.33590e+01,0.34290e+01,0.35330e+01,
        0.32610e+01,0.26120e+01,0.21790e+01,0.16530e+01,0.18320e+01,0.23280e+01,
        0.27370e+01,0.30140e+01,0.33280e+01,0.34040e+01,0.29810e+01,0.25570e+01,
        0.25760e+01,0.23520e+01,0.25560e+01,0.28640e+01,0.32140e+01,0.34350e+01,
        0.34580e+01,0.33260e+01,0.28350e+01,0.24760e+01,0.23730e+01,0.23890e+01,
        0.27420e+01,0.32100e+01,0.35200e+01,0.38280e+01,0.36280e+01,0.28370e+01,
        0.24060e+01,0.26750e+01,0.25540e+01,0.28940e+01,0.32020e+01,0.32240e+01,
        0.33520e+01,0.31540e+01,0.28780e+01,0.24760e+01,0.23030e+01,0.23600e+01,
        0.26710e+01,0.28670e+01,0.33100e+01,0.34490e+01,0.36460e+01,0.34000e+01,
        0.25900e+01,0.18630e+01,0.15810e+01,0.16900e+01,0.17710e+01,0.22740e+01,
        0.25760e+01,0.31110e+01,0.36050e+01,0.35430e+01,0.27690e+01,0.20210e+01,
        0.21850e+01,0.25880e+01,0.28800e+01,0.31150e+01,0.35400e+01,0.38450e+01,
        0.38000e+01,0.35790e+01,0.32640e+01,0.25380e+01,0.25820e+01,0.29070e+01,
        0.31420e+01,0.34330e+01,0.35800e+01,0.34900e+01,0.34750e+01,0.35790e+01,
        0.28290e+01,0.19090e+01,0.19030e+01,0.20330e+01,0.23600e+01,0.26010e+01,
        0.30540e+01,0.33860e+01,0.35530e+01,0.34680e+01,0.31870e+01,0.27230e+01,
        0.26860e+01,0.28210e+01,0.30000e+01,0.32010e+01,0.34240e+01,0.35310e+01
    };
}
```

```

int i, model[4] = {2,0,1,0}, n_obs = 114;
int *outlier_stat = NULL, num_outliers;
float *parameters = NULL, *result = NULL;
float res_sigma, aic;

/* Simulate measurement error */
series[29] = 0.35570e+01;
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("\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(" t=%2d\t\t Type=%d\n", outlier_stat[2*i],
        outlier_stat[2*i+1]);
printf("\n\nRSE: %lf\n", res_sigma);
printf("AIC: %lf\n", aic);
printf("\nExtract from the series:\n\n");
printf ("time point    original series    outlier free series\n\n");
for (i=0; i<36; i++)
    printf ("%2d %21.4f %21.4f\n", i+1, series[i], result[i]);
}

```

Output

ARMA parameters:

0	1.052683
1	1.389253
2	-0.752184

Number of outliers: 1

Outlier statistics:

Time point	Outlier type
t=30	Type=1

RSE: 0.225020

AIC: 202.958511

Extract from the series:

time point	original series	outlier free series
1	2.4300	2.4300
2	2.5060	2.5060
3	2.7670	2.7670
4	2.9400	2.9400
5	3.1690	3.1690
6	3.4500	3.4500

7	3.5940	3.5940
8	3.7740	3.7740
9	3.6950	3.6950
10	3.4110	3.4110
11	2.7180	2.7180
12	1.9910	1.9910
13	2.2650	2.2650
14	2.4460	2.4460
15	2.6120	2.6120
16	3.3590	3.3590
17	3.4290	3.4290
18	3.5330	3.5330
19	3.2610	3.2610
20	2.6120	2.6120
21	2.1790	2.1790
22	1.6530	1.6530
23	1.8320	1.8320
24	2.3280	2.3280
25	2.7370	2.7370
26	3.0140	3.0140
27	3.3280	3.3280
28	3.4040	3.4040
29	2.9810	2.9810
30	3.5570	2.7403
31	2.5760	2.5760
32	2.3520	2.3520
33	2.5560	2.5560
34	2.8640	2.8640
35	3.2140	3.2140
36	3.4350	3.4350

Example 2

This example is an artificial realization of an ARMA(1,1) process via formula

$$Y_t - 0.8Y_{t-1} = 10.0 + a_t + 0.5a_{t-1}, \quad t = 1, \dots, 300, \quad \text{Gaussian white noise, } E[Y_t]=50.0.$$

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 <stdio.h>

int main()
{
    int i, n_obs = 300, num_outliers;
    int outlier_stat[300], model[4] = {1,1,1,0};
    float res_sigma, aic;
    float parameters[300], result[300], omega[300];

    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,
```



```
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[2*i], outlier_stat[2*i+1]);
printf("\nOmega statistics:\n");
printf("Time point\tomega\n");
for (i=0; i<num_outliers; i++)
    printf("%d%21.6f\n", outlier_stat[2*i], omega[i]);
printf("\nRSE: %lf\n", res_sigma);
printf("AIC: %lf\n\n", aic);
}
```

Output

```
ARMA parameters:
0          10.833087
1           0.785139
2          -0.496548
```

```
Number of outliers: 2
```

```
Outlier statistics:
Time point  Outlier type
150         1
200         3
```

```
Omega statistics:
Time point  omega
150         4.477888
200         3.381441
```

```
RSE: 1.007223
AIC: 1417.044434
```

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 $1,2,\dots,n_predict$.

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 $\{Y_t^*\}_{t=1,\dots,n}$:

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

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:

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

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

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

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

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

Here,

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

and

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

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 $t = t_j$, $I_t(t_j) = \begin{cases} 0 & \text{for } t < t_j, \\ 0 & \text{for } t \neq t_j, t_j + k, k \geq 0, \\ \omega_j & \text{for } t = t_j, \\ \omega_j \delta^k & \text{for } t = t_j + k, k \geq 1. \end{cases}$

Innovational outliers (IO)	$\omega_j L_j(B) I_t(t_j) =$	$\begin{cases} 0 & \text{for } t < t_j, \\ 0 & \text{for } t \neq t_j, t_j + k, k \geq 0, \\ \omega_j & \text{for } t = t_j, \\ \omega_j \delta^k & \text{for } t = t_j + k, k \geq 1. \end{cases}$
Additive outliers (AO)		
Level shifts (LS)		
Temporary changes (TC)		

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

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

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

$$Y_t - Y_{t-1} + 0.24Y_{t-2} = 10.0 + a_t + 0.5a_{t-1}, \{a_t\} \text{ a Gaussian white noise process.}$$

Outliers were artificially added to the outlier free series $\{Y_t\}_{t=1, \dots, 280}$ 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 $\{Z_t\}_{t=1, \dots, 280}$. For both series, forecasts were determined for time points $t=281, \dots, 290$ and compared with the actual values of the series.

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

int 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,  
42.1471291, 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] = result[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);
}

```

Output

ARMA parameters:

0	8.892076
1	0.943928
2	-0.150295
3	-0.559073

Number of outliers: 2

Outlier statistics:

Time point	Outlier type
150	2
200	1

RSE:1.004306

AIC:1323.617310

```

* * * Forecast Table for outlier contaminated series * * *
  Orig. Series      forecast      prob. limits      psi weights
      1          2          3          4
1      42.6384      42.3158      1.9684      1.5030
2      41.7089      42.7934      3.5535      1.2684
3      43.9399      43.2822      4.3430      0.9714
4      45.4284      43.6718      4.7453      0.7263
5      44.4558      43.9662      4.9560      0.5396
6      45.1762      44.1854      5.0685      0.4002
7      45.3489      44.3481      5.1293      0.2966
8      45.1893      44.4688      5.1625      0.2199
9      46.3755      44.5582      5.1806      0.1629
10     45.6083      44.6245      5.1905      0.1208

```

* * * Forecast Table for outlier free series * * *				
Outlier free series	forecast		prob. limits	psi weights
	1	2	3	4
1	40.1384	40.5903	1.9684	1.5030
2	39.2089	41.0679	3.5535	1.2684
3	41.4399	41.5567	4.3430	0.9714
4	42.9284	41.9463	4.7453	0.7263
5	41.9558	42.2406	4.9560	0.5396
6	42.6762	42.4599	5.0685	0.4002
7	42.8489	42.6226	5.1293	0.2966
8	42.6893	42.7433	5.1625	0.2199
9	43.8755	42.8327	5.1806	0.1629
10	43.1083	42.8990	5.1905	0.1208

auto_arima



[more...](#)

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, \dots, t_{n_obs} , the actual length of the series, including missing observations is $n = t_{n_obs} - t_1 + 1$.

int `tpoints[]` (Input)

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

float `x[]` (Input)

A vector of length `n_obs` containing the observed time series values $Y_1^*, Y_2^*, \dots, Y_{n_obs}^*$. This series can contain outliers and missing observations. Outliers are identified by this function 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 $t_i + 1, t_i + 2, \dots, t_{i+1} - 1$. Therefore, the gap free series is assumed to be defined for equidistant time points. 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 $d = \text{model}[3] = 0$, then an ARMA(p, q) model or AR(p) model is fitted to the outlier-free version of the observed series Y_t^* . If $d = \text{model}[3] > 0$, these parameters are computed for an ARMA(p, q) representation of the seasonally adjusted series

$$Z_t^* = A_s^d \cdot Y_t^* = (1 - B_s)^d \cdot Y_t^*, \text{ where } B_s Y_t^* = Y_{t-s}^* \text{ and } s = \text{model}[2] \geq 1.$$

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_AICC, float *aicc,
IMSLS_BIC, float *bic,
IMSLS_MODEL_SELECTION_CRITERION, int criterion,
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_SUPPLY_WORK_ARRAYS, int liwork, int iwork[], int lwork, float work[],
IMSLS_RETURN_USER, float parameters[],
0)

```

Optional Arguments

IMSLS_METHOD, *int* method (Input)

The method used in model selection:

	Description
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.

For more information, see the “[Description](#)” section.

Default: method = 1

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), $0 < \text{delta} < 1$.

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.

IMSL_S_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.

IMSL_S_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}

IMSL_S_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}

IMSL_S_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 ($t = t_1, t_1 + 1, t_1 + 2, \dots, t_{n_obs}$) 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.

IMSL_S_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, $n = t_{n_obs} - t_1 + 1 \geq n_obs$. See IMSLS_OUTLIER_STATISTICS. If num_outliers = 0, outlier_stat stays unchanged.

IMSL_S_AIC, *float* *aic (Output)

The AIC (Akaike's Information Criterion) value for the optimum model. Uses an approximation of the maximum log-likelihood based on an estimate of the innovation variance of the series.

IMSLS_AICC, *float *aicc* (Output)

The AICC (corrected AIC) value for the optimum model. Uses an approximation of the maximum log-likelihood based on an estimate of the innovation variance of the series.

IMSLS_BIC, *float *bic* (Output)

The BIC (Bayesian Information Criterion) value for the optimum model. Uses an approximation of the maximum log-likelihood based on an estimate of the innovation variance of the series.

IMSLS_MODEL_SELECTION_CRITERION, *int criterion* (Input)

The information criterion used for optimum model selection.

criterion	selected information criterion
0	Akaike's Information Criterion (AIC)
1	Akaike's Corrected Information Criterion (AICC)
2	Bayesian Information Criterion (BIC)

Default: `criterion = 0`.

IMSLS_OUT_FREE_SERIES, *float **outfree_series* (Output)

Address of a pointer to an internally allocated array of length n by 2, where $n = t_{n_obs} - t_1 + 1$. 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 $n = t_{n_obs} - t_1 + 1$. 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_{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 $t = t_{n_obs} + 1, t_{n_obs} + 2, \dots, t_{n_obs} + n_predict$. 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 $t = t_{n_obs} + 1, t_{n_obs} + 2, \dots, t_{n_obs} + n_predict$. 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_SUPPLY_WORK_ARRAYS`, *int liwork, int iwork[], int lwork, float work[],* (Input/Output)

The use of this optional argument will increase efficiency and avoid memory fragmentation run-time failures for large problems by allowing the user to provide the sizes and locations of the working arrays `work` and `iwork`. It is also useful if many time series have to be processed sequentially because it can significantly reduce the amount of memory that has to be reallocated. This optional argument can be used in conjunction with `method = 1` and `method = 2`. With `maxt` as the maximum number of threads that will be active and `nobs_act` the length of the time series (including missing values), it is required that

$$liwork \geq 2 \times maxt \times (2 + nobs_act).$$

The minimum length of array `work` depends on the choice of the method. For method 1, it is required that

$$lwork \geq maxt \times (3 \times nobs_act + 1 + maxlag).$$

Method 2 requires

$$lwork \geq maxt \times (3 \times nobs_act + 1 + ub_p + ub_q),$$

where `ub_p` and `ub_q` denote the maximum values in arrays `p_initial` and `q_initial`, respectively.

Without the use of OpenMP and parallel threading, `maxt` = 1.

`IMSLS_RETURN_USER`, *float x[]* (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`, $p = \text{model}[0]$ and $q = \text{model}[1]$.

Description

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:

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

where

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \dots - \phi_p B^p, \quad \theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \dots - \theta_q B^q, \quad \Delta_s^d = (1 - B^s)^d$$

and

$$B^k Y_t = Y_{t-k}.$$

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:

$$Y_t^* = Y_t + \text{outlier_effect}_t.$$

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

1. innovational
2. additive
3. level shift

4. temporary change *and*
5. 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 $p = 0, \dots, \text{maxlag}$.

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 `IMSL_S_INITIAL` and `IMSL_D_INITIAL` are not defined, then values $s > 0$ and $d \geq 0$ must be specified in `model[2]` and `model[3]`. If `IMSL_S_INITIAL` and `IMSL_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:

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 <code>delta</code> . The default of <code>delta=0.7</code> 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 D from Box, Jenkins and Reinsel (1994), the hourly viscosity readings of a chemical process. [Method 1](#) without seasonal adjustment is chosen to find an appropriate AR(p) model for the first 304 observations of this series, measured at time points $t = 1$ to $t = 304$. A forecast is then done at origin $t = 304$ for lead times 1 to 6 and compared with the actual time series values which are stored in array `actual`.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int n_obs, n_predict, i, num_outliers;
    int *outlier_stat = NULL, model[4], times[304];
    float aic, res_sigma, *parameters = NULL;
    float outlier_forecast[18], forecast_table[24];

    /* Values of series D at time points t=1,...,t=304 */
    float x[304] = {
        8.0,8.0,7.4,8.0,8.0,8.0,8.0,8.8,8.4,8.4,8.0,8.2,8.2,8.2,8.4,
        8.4,8.4,8.6,8.8,8.6,8.6,8.6,8.6,8.8,8.9,9.1,9.5,8.5,8.4,
        8.3,8.2,8.1,8.3,8.4,8.7,8.8,8.8,9.2,9.6,9.0,8.8,8.6,8.6,8.8,
        8.8,8.6,8.6,8.4,8.3,8.4,8.3,8.3,8.1,8.2,8.3,8.5,8.1,8.1,7.9,
        8.3,8.1,8.1,8.1,8.4,8.7,9.0,9.3,9.3,9.5,9.3,9.5,9.5,9.5,9.5,
        9.5,9.5,9.9,9.5,9.7,9.1,9.1,8.9,9.3,9.1,9.1,9.3,9.5,9.3,9.3,
        9.3,9.9,9.7,9.1,9.3,9.5,9.4,9.0,9.0,8.8,9.0,8.8,8.6,8.6,8.0,
```

```

8.0,8.0,8.0,8.6,8.0,8.0,8.0,7.6,8.6,9.6,9.6,10.0,9.4,9.3,9.2,
9.5,9.5,9.5,9.9,9.9,9.5,9.3,9.5,9.5,9.1,9.3,9.5,9.3,9.1,9.3,
9.1,9.5,9.4,9.5,9.6,10.2,9.8,9.6,9.6,9.4,9.4,9.4,9.4,9.6,9.6,
9.4,9.4,9.0,9.4,9.4,9.6,9.4,9.2,8.8,8.8,9.2,9.2,9.6,9.6,9.8,
9.8,10.0,10.0,9.4,9.8,8.8,8.8,8.8,8.8,9.6,9.6,9.6,9.2,9.2,9.0,
9.0,9.0,9.4,9.0,9.0,9.4,9.4,9.6,9.4,9.6,9.6,9.6,10.0,10.0,9.6,
9.2,9.2,9.2,9.0,9.0,9.6,9.8,10.2,10.0,10.0,10.0,9.4,9.2,9.6,9.7,
9.7,9.8,9.8,9.8,10.0,10.0,8.6,9.0,9.4,9.4,9.4,9.4,9.4,9.6,10.0,
10.0,9.8,9.8,9.7,9.6,9.4,9.2,9.0,9.4,9.6,9.6,9.6,9.6,9.6,9.6,
9.0,9.4,9.4,9.4,9.6,9.4,9.6,9.6,9.8,9.8,9.8,9.6,9.2,9.6,9.2,
9.2,9.6,9.6,9.6,9.6,9.6,9.6,10.0,10.0,10.4,10.4,9.8,9.0,9.6,9.8,
9.6,8.6,8.0,8.0,8.0,8.0,8.4,8.8,8.4,8.4,9.0,9.0,9.4,10.0,10.0,
10.0,10.2,10.0,10.0,9.6,9.0,9.0,8.6,9.0,9.6,9.6,9.0,9.0,8.9,8.8,
8.7,8.6,8.3,7.9};
/* Actual values of series D at time points t=305,...,t=310 */
float actual[6] = {8.5,8.7,8.9,9.1,9.1,9.1};

char *col_labels[] = {
    "Lead Time",
    "Orig. Series",
    "Forecast",
    "Dev. for Prob. Limits",
    "Psi"};

n_predict = 6;
n_obs = 304;

/* Define times from t=1 to t=304 */
for (i=0;i<n_obs;i++) times[i] = i+1;

parameters = imsls_f_auto_arima(n_obs, times, x,
    IMSLS_MODEL, model,
    IMSLS_AIC, &aic,
    IMSLS_MAX_LAG, 5,
    IMSLS_CRITICAL, 3.8,
    IMSLS_NUM_OUTLIERS, &num_outliers,
    IMSLS_OUTLIER_STATISTICS, &outlier_stat,
    IMSLS_RESIDUAL_SIGMA, &res_sigma,
    IMSLS_NUM_PREDICT, n_predict,
    IMSLS_OUTLIER_FORECAST_USER, 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  Outlier type\n");
for (i=0; i<num_outliers; i++)
    printf("%d%lld\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] = actual[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("** * * Forecast Table * * *",
        n_predict, 4, forecast_table,
        IMSLS_COL_LABELS, col_labels,
        IMSLS_WRITE_FORMAT, "%11.4f", 0);
}

```

Output

Method 1: Automatic ARIMA model selection, no differencing

Model chosen: $p=1, q=0, s=1, d=0$

Number of outliers: 1

Outlier statistics:

Time point	Outlier type
217	3

AIC = 678.224731

RSE = 0.290680

Parameters:

parameters[0] = 1.044163

parameters[1] = 0.887724

* * * Forecast Table * * *					
Lead Time	Orig. Series	Forecast	Dev. for Prob.	Limits	Psi
1	8.5000	8.0572		0.5697	0.8877
2	8.7000	8.1967		0.7618	0.7881
3	8.9000	8.3206		0.8843	0.6996
4	9.1000	8.4306		0.9699	0.6210
5	9.1000	8.5282		1.0325	0.5513
6	9.1000	8.6148		1.0792	0.4894

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 $p = 3, q = 1, s = 1, d = 0$ is chosen as optimum.

```

#include <imsls.h>
#include <stdio.h>
int main()
{
    int n_obs, n_predict, i, num_outliers;
    int model[4], times[304];
    int n_p_initial = 4, n_q_initial = 4;
    int n_s_initial = 2, n_d_initial = 3;
    int s_initial[2] = {1,2}, d_initial[3] = {0,1,2};
    int p_initial[4] = {0,1,2,3}, q_initial[4] = {0,1,2,3};
    int outlier_stat[608];
    float aic, res_sigma;
    float parameters[7], outlier_forecast[18], forecast_table[24];
}

```

```

/* Values of series D at time points t=1,...,t=304 */
float x[310] = {
    8.0,8.0,7.4,8.0,8.0,8.0,8.0,8.8,8.4,8.4,8.0,8.2,8.2,8.2,8.4,
    8.4,8.4,8.6,8.8,8.6,8.6,8.6,8.6,8.8,8.9,9.1,9.5,8.5,8.4,
    8.3,8.2,8.1,8.3,8.4,8.7,8.8,8.8,9.2,9.6,9.0,8.8,8.6,8.6,8.8,
    8.8,8.6,8.6,8.4,8.3,8.4,8.3,8.3,8.1,8.2,8.3,8.5,8.1,8.1,7.9,
    8.3,8.1,8.1,8.1,8.4,8.7,9.0,9.3,9.3,9.5,9.3,9.5,9.5,9.5,9.5,
    9.5,9.5,9.9,9.5,9.7,9.1,9.1,8.9,9.3,9.1,9.1,9.3,9.5,9.3,9.3,
    9.3,9.9,9.7,9.1,9.3,9.5,9.4,9.0,9.0,8.8,9.0,8.8,8.6,8.6,8.0,
    8.0,8.0,8.0,8.6,8.0,8.0,8.0,7.6,8.6,9.6,9.6,10.0,9.4,9.3,9.2,
    9.5,9.5,9.5,9.9,9.9,9.5,9.3,9.5,9.5,9.1,9.3,9.5,9.3,9.1,9.3,
    9.1,9.5,9.4,9.5,9.6,10.2,9.8,9.6,9.6,9.4,9.4,9.4,9.4,9.6,9.6,
    9.4,9.4,9.0,9.4,9.4,9.6,9.4,9.2,8.8,8.8,9.2,9.2,9.6,9.6,9.8,
    9.8,10.0,10.0,9.4,9.8,8.8,8.8,8.8,8.8,9.6,9.6,9.6,9.2,9.2,9.0,
    9.0,9.0,9.4,9.0,9.0,9.4,9.4,9.6,9.4,9.6,9.6,9.6,10.0,10.0,9.6,
    9.2,9.2,9.2,9.0,9.0,9.6,9.8,10.2,10.0,10.0,10.0,9.4,9.2,9.6,9.7,
    9.7,9.8,9.8,9.8,10.0,10.0,8.6,9.0,9.4,9.4,9.4,9.4,9.6,10.0,
    10.0,9.8,9.8,9.7,9.6,9.4,9.2,9.0,9.4,9.6,9.6,9.6,9.6,9.6,9.6,
    9.0,9.4,9.4,9.4,9.6,9.4,9.6,9.6,9.8,9.8,9.8,9.6,9.2,9.6,9.2,
    9.2,9.6,9.6,9.6,9.6,9.6,9.6,10.0,10.0,10.4,10.4,9.8,9.0,9.6,9.8,
    9.6,8.6,8.0,8.0,8.0,8.0,8.4,8.8,8.4,8.4,9.0,9.0,9.4,10.0,10.0,
    10.0,10.2,10.0,10.0,9.6,9.0,9.0,8.6,9.0,9.6,9.6,9.0,9.0,8.9,8.8,
    8.7,8.6,8.3,7.9};

/* Actual values of series D at time points t=305,...,t=310 */
float actual[6] = {8.5,8.7,8.9,9.1,9.1,9.1};

char *col_labels[] = {
    "Lead Time",
    "Orig. Series",
    "Forecast",
    "Dev. for Prob. Limits",
    "Psi"};

n_predict = 6;
n_obs = 304;

/* Define times from t=1 to t=304 */
for (i=0;i<n_obs;i++) times[i] = i+1;

imsls_f_auto_arima(n_obs, times, x,
    IMSLS_MODEL, model,
    IMSLS_AIC, &aic,
    IMSLS_CRITICAL, 3.8,
    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,
    IMSLS_RESIDUAL_SIGMA, &res_sigma,
    IMSLS_NUM_PREDICT, n_predict,
    IMSLS_OUTLIER_FORECAST_USER, outlier_forecast,
    IMSLS_RETURN_USER, parameters,
    0);

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  Outlier type\n");
for (i=0; i<num_outliers; i++)
    printf("%d%11d\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] = actual[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(" * * * Forecast Table * * *",
    n_predict, 4, forecast_table,
    IMSLS_COL_LABELS, col_labels,
    IMSLS_WRITE_FORMAT, "%11.4f", 0);
}

```

Output

Method 2: Grid search, differencing allowed

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

Number of outliers: 1

Outlier statistics:

Time point	Outlier type
217	3

AIC = 675.885986
RSE = 0.286720

Parameters:
parameters[0] = 1.892720
parameters[1] = 0.184380
parameters[2] = 0.641278
parameters[3] = -0.029176
parameters[4] = -0.743030

* * * Forecast Table * * *					
Lead Time	Orig. Series	Forecast	Dev. for Prob.	Limits	Psi
1	8.5000	8.0471		0.5620	0.9274
2	8.7000	8.2004		0.7664	0.8123
3	8.9000	8.3347		0.8921	0.7153
4	9.1000	8.4534		0.9785	0.6257
5	9.1000	8.5569		1.0397	0.5504
6	9.1000	8.6483		1.0847	0.4819

Example 3

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

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

int main()
{
    int n_obs, n_predict, i, num_outliers;
    int *outlier_stat = NULL;
    int model[4] = {3,1,1,0}, times[304];
    float aic, res_sigma, *parameters = NULL;
    float outlier_forecast[18], forecast_table[24];

    /* Values of series D at time points t=1,...,t=304 */
    float x[304] = {
        8.0,8.0,7.4,8.0,8.0,8.0,8.0,8.8,8.4,8.4,8.0,8.2,8.2,8.2,8.4,
        8.4,8.4,8.6,8.8,8.6,8.6,8.6,8.6,8.8,8.9,9.1,9.5,8.5,8.4,
        8.3,8.2,8.1,8.3,8.4,8.7,8.8,8.8,9.2,9.6,9.0,8.8,8.6,8.6,8.8,
        8.8,8.6,8.6,8.4,8.3,8.4,8.3,8.3,8.1,8.2,8.3,8.5,8.1,8.1,7.9,
        8.3,8.1,8.1,8.1,8.4,8.7,9.0,9.3,9.3,9.5,9.3,9.5,9.5,9.5,9.5,
        9.5,9.5,9.9,9.5,9.7,9.1,9.1,8.9,9.3,9.1,9.1,9.3,9.5,9.3,9.3,
        9.3,9.9,9.7,9.1,9.3,9.5,9.4,9.0,9.0,8.8,9.0,8.8,8.6,8.6,8.0,
        8.0,8.0,8.0,8.6,8.0,8.0,8.0,7.6,8.6,9.6,9.6,10.0,9.4,9.3,9.2,
        9.5,9.5,9.5,9.9,9.9,9.5,9.3,9.5,9.5,9.1,9.3,9.5,9.3,9.1,9.3,
        9.1,9.5,9.4,9.5,9.6,10.2,9.8,9.6,9.6,9.4,9.4,9.4,9.4,9.6,9.6,
        9.4,9.4,9.0,9.4,9.4,9.6,9.4,9.2,8.8,8.8,9.2,9.2,9.6,9.6,9.8,
        9.8,10.0,10.0,9.4,9.8,8.8,8.8,8.8,8.8,9.6,9.6,9.6,9.2,9.2,9.0,
        9.0,9.0,9.4,9.0,9.0,9.4,9.4,9.6,9.4,9.6,9.6,9.6,10.0,10.0,9.6,
        9.2,9.2,9.2,9.0,9.0,9.6,9.8,10.2,10.0,10.0,10.0,9.4,9.2,9.6,9.7,
        9.7,9.8,9.8,9.8,10.0,10.0,8.6,9.0,9.4,9.4,9.4,9.4,9.4,9.6,10.0,
        10.0,9.8,9.8,9.7,9.6,9.4,9.2,9.0,9.4,9.6,9.6,9.6,9.6,9.6,9.6,
        9.0,9.4,9.4,9.4,9.6,9.4,9.6,9.6,9.8,9.8,9.8,9.6,9.2,9.6,9.2,
        9.2,9.6,9.6,9.6,9.6,9.6,9.6,10.0,10.0,10.4,10.4,9.8,9.0,9.6,9.8,
        9.6,8.6,8.0,8.0,8.0,8.0,8.4,8.8,8.4,8.4,9.0,9.0,9.4,10.0,10.0,
        10.0,10.2,10.0,10.0,9.6,9.0,9.0,8.6,9.0,9.6,9.6,9.0,9.0,8.9,8.8,
        8.7,8.6,8.3,7.9};

    /* Actual values of series D at time points t=305,...,t=310 */
    float actual[6] = {8.5,8.7,8.9,9.1,9.1,9.1};

    char *col_labels[] = {
        "Lead Time",
        "Orig. Series",
        "Forecast",
        "Dev. for Prob. Limits",
        "Psi"};

    n_predict = 6;
    n_obs = 304;

    /* Define times from t=1 to t=304 */
    for (i=0;i<n_obs;i++) times[i] = i+1;

    parameters = imsls_f_auto_arima(n_obs, times, x,
        IMSLS_MODEL, model,
```

```

    IMSLS_AIC, &aic,
    IMSLS_CRITICAL, 3.8,
    IMSLS_METHOD, 3,
    IMSLS_NUM_OUTLIERS, &num_outliers,
    IMSLS_OUTLIER_STATISTICS, &outlier_stat,
    IMSLS_RESIDUAL_SIGMA, &res_sigma,
    IMSLS_NUM_PREDICT, n_predict,
    IMSLS_OUTLIER_FORECAST_USER, 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  Outlier type\n");
for (i=0; i<num_outliers; i++)
    printf("%d%lld\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] = actual[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("* * * Forecast Table * * *",
    n_predict, 4, forecast_table,
    IMSLS_COL_LABELS, col_labels,
    IMSLS_WRITE_FORMAT, "%11.4f", 0);
}

```

Output

Method 3: Specified ARIMA model

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

Number of outliers: 1

Outlier statistics:

Time point	Outlier type
217	3

AIC = 675.885925

RSE = 0.286720

Parameters:

parameters[0] = 1.892720

parameters[1] = 0.184380

parameters[2] = 0.641278

parameters[3] = -0.029176

parameters[4] = -0.743030

* * * Forecast Table * * *				
Lead Time	Orig. Series	Forecast	Dev. for Prob.	Psi
			Limits	
1	8.5000	8.0471	0.5620	0.9274
2	8.7000	8.2004	0.7664	0.8123
3	8.9000	8.3347	0.8921	0.7153
4	9.1000	8.4534	0.9785	0.6257
5	9.1000	8.5569	1.0397	0.5504
6	9.1000	8.6483	1.0847	0.4819

auto_parm

Estimates structural breaks in non-stationary univariate time series.

Synopsis

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

```
int *imsls_f_auto_parm(int nobs, float y[ ], int *npcs, ..., 0)
```

The type *double* function is `imsls_d_auto_parm`.

Required Arguments

int **nobs** (Input)

The number of observations in the time series (y).

float **y[]** (Input)

An array of length **nobs** containing the time series.

int ***npcs** (Input/Output)

The number of requested/estimated pieces or segments of the time series. **npcs** is considered input only when `IMSLS_AR_MODEL` is provided.

Return Value

A pointer to an array (**arp**) of length $\text{npcs} \times 2$ containing the break points and AR orders for the derived model. If `IMSLS_AR_MODEL` is used, the return value is `NULL`.

Column Index	Description
0	Structural break points τ_j
1	AR order (p_j) for each segment

Synopsis with Optional Arguments

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

```

int *imsls_f_auto_parm(int nobs, float y[], int *npcs,
    IMSLS_MAX_AR_ORDER, int max_ar_order,
    IMSLS_METHOD, int method,
    IMSLS_MODEL_SELECTION_CRITERION, int criterion,
    IMSLS_MAXIMUM_LIKELIHOOD, int likelihood,
    IMSLS_AR_MODEL, int arp[],
    IMSLS_PRINT, int print,
    IMSLS_RANDOM_SEED, int seed,
    IMSLS_PROB_DISTRIBUTION, float pdistn[],
    IMSLS_MIN_OBSERVATIONS, int mspan[],
    IMSLS_GA_PARAMETERS, float gaparm[],
    IMSLS_ISLAND, int island[],
    IMSLS_MAX_MIGRATIONS, int maxmig,
    IMSLS_STOP_ITERATIONS, int stopiters,
    IMSLS_SELECTION_CRITERION_VALUE, float *value,
    IMSLS_AR_FIT, float **arfit,
    IMSLS_AR_FIT_USER, float arfit[],
    IMSLS_AR_STATS, float **arstat,
    IMSLS_AR_STATS_USER, float arstat[],
    0)

```

Optional Arguments

`IMSLS_MAX_AR_ORDER`, *int* `max_ar_order` (Input)
 Maximum order to consider for each AR model.
 Default: `max_ar_order` = 20.

IMSLS_METHOD, *int method* (Input)
Method of estimation.

method	Method Used
0	Yule -Walker
1	Least Squares
2	Burg

Default: `method = 0`.

IMSLS_MODEL_SELECTION_CRITERION, *int criterion* (Input)
Selection criterion.

criterion	Criterion Used
0	Minimum Description Length (MDL)
1	Akaike's Information Criterion (AIC)

Default: `criterion = 0`.

IMSLS_MAXIMUM_LIKELIHOOD, *int likelihood* (Input)
Likelihood computation method.

likelihood	Computation Method
0	Exact
1	Approximate

Default: `likelihood = 0`.

IMSLS_AR_MODEL, *int arp[]* (Input)

A user specified array of length `npcs × 2` containing the break points and AR orders. When this argument is used, only the AR parameters and quality of the fit are determined.

Column Index	Description
0	Structural break points τ_j
1	AR order (p_j) for each segment

IMSLS_PRINT, *int* **print** (Input)
Printing option.

print	Action
0	No printing
1	Prints final results only
2	Prints intermediate and final results

Default: **print** = 0.

IMSLS_RANDOM_SEED, *int* **seed** (Input)

Seed of the random number generator. For the same data and parameter settings, `imsls_f_auto_parm` will return the same results each time if a seed is set. If **seed** = 0, the system clock will be used to generate a seed. The result will be nondeterministic.

Default: **seed** = 0.

Note: The following *input* arguments are for setting up and running the embedded Genetic Algorithm. In most situations, the default values should be used for these arguments. Users may wish to change some or all for testing or research purposes.

IMSLS_PROB_DISTRIBUTION, *float* **pdistn**[] (Input)

Array of length `max_ar_order + 1` giving the probability distribution over the AR order variable $p = 0, \dots, \text{max_ar_order}$. $i = 0, \dots, \text{max_ar_order}$ is used to randomly assign an AR order to breakpoint position for a given chromosome. `pdistn[i] >= 0` and if $\sum \text{pdistn}$ is not equal to 1, the values will be normalized, i.e., `pdistn[i] = pdistn[i] / $\sum \text{pdistn}$` .

Default: `pdistn[i] = 1 / (max_ar_order + 1)` for all i .

IMSLS_MIN_OBSERVATIONS, *int* **mspan**[] (Input)

Array of length `max_ar_order + 1` containing minimum number of observations required for valid estimates of AR model with order $p = 0, \dots, \text{max_ar_order}$.

Default: `mspan [p] = 2 × (number of parameters) + 2 = 2 × (p + 2) + 2`.

IMSLI_S_GA_PARAMETERS, *float* gaparm[] (Input)

Array of length 4 containing parameters that control the behavior of the genetic algorithm. These values should be strictly greater than zero and less than one to avoid unexpected results.

Index	Behavior
0	Probability used to set initial break points in a chromosome. Default: $\min(\text{mspan}) / \text{nobs}$.
1	Probability of crossover used to decide between a crossover and a mutation. Default: $1 - \min(\text{mspan}) / \text{nobs}$.
2	In the mutation operation, probability an AR(p) model is enforced at the current position. Default: 0.4.
3	In the mutation operation, probability a break point is disallowed at the current position. Default: 0.3.

Note: gaparm[2] and gaparm[3] must be valid probabilities and their sum must be between 0 and 1. $1 - \text{gaparm}[2] - \text{gaparm}[3]$ is the probability that the chromosome j inherits the parent's chromosome j .

IMSLI_S_ISLAND, *int* island[] (Input)

Array of length 5 containing the migration policy parameters.

	Policy
0	Number of islands. Default: 40.
1	Number of generations that pass before migration occurs. Note that the convergence of the algorithm is revised whether migrations take place or not (see argument island[4]). Default: 5.
2	Number of subjects that migrate at each migration event. Default: 2.
3	Population size (number of chromosomes) per island. Default: 40.
4	Migration flag. If 1, migration is performed. Default: 1.

IMSL_MAX_MIGRATIONS, *int maxmig* (Input)

Maximum number of times that migrations may take place before the function is stopped if convergence has not occurred.

Default: `maxmig = 20`.

IMSL_STOP_ITERATIONS, *int stopiters* (Input)

Number of iterations. The function will declare convergence and stop the iterations if the criterion value (MDL/AIC) has not changed after `stopiters` consecutive migrations. Otherwise, the algorithm will declare non-convergence and stop after `maxmig` migrations have taken place. See also **IMSL_MAX_MIGRATIONS** and `island[1]`. Note that logically, `stopiters < maxmig`.

Default: `stopiters = 10`.

IMSL_SELECTION_CRITERION_VALUE, *float *value* (Output)

Final value of the selection criterion.

IMSL_AR_FIT, *float **arfit* (Output)

Address of a pointer to an internally allocated array of length `npcs × max_ar_order` containing the AR coefficient estimates ϕ for each segment. `arfit[i*max_ar_order+j]` is the j -th coefficient for segment i where $i = 0, \dots, \text{npcs} - 1$ and $j = 0, \dots, \text{arp}[i \times 2 + 1]$.

Note that the intercept is not reported.

IMSL_AR_FIT_USER, *float arfit []* (Output)

Storage for array `arfit` is provided by the user. If `npcs` is output, the user should allocate $(\text{nobs} - 1) \times \text{max_ar_order}$ to assure sufficient space. See **IMSL_AR_FIT**.

IMSL_AR_STATS, *float **arstat* (Output)

Address of a pointer to an internally allocated array of length `npcs × 2`.

	Description
0	Likelihood values for each of the fitted AR models
1	Residual variances for each of the fitted AR models

IMSL_AR_STATS_USER, *float arstat []* (Output)

Storage for array `arstat` is provided by the user. If `npcs` is output, the user should allocate $(\text{nobs} - 1) \times 2$ to assure sufficient space. See **IMSL_AR_STATS**.

Description

Function `imsls_f_auto_parm` estimates the structural breaks of a non-stationary time series using, with permission from the authors, the method developed by Davis et al (2006). `imsls_f_auto_parm` estimates a partition of the time index and models the time series in each segment as a separate auto-regressive (AR(ρ)) process. The function returns the estimated breakpoints, the estimated AR(ρ) models, and supporting statistics.

For the observed time series $Y_t, t = 1, \dots, n$, the problem is to find m , the number of breaks, their locations, $1 < \tau_1 < \tau_2 < \dots < \tau_m < n$, and $p_j, j = 1, \dots, m+1$, the order of the AR process in which the j -th segment is modeled. That is, $Y_t = X_{t,j}$ for $\tau_{j-1} \leq t < \tau_j$ (for convenience, $\tau_0 := 1$ and $\tau_{m+1} := n+1$), where $\{X_{t,j}\}$ is an AR process of order p_j

$$X_{t,j} = \phi_{j,1}X_{t-1,j} + \dots + \phi_{j,p_j}X_{t-j,p_j} + \sigma_t \varepsilon_t$$

and ε_t , the noise sequence, is independent and identically distributed with mean 0 and variance 1. Note that a series with m breaks will have $m+1$ segments ($m+1 = \text{npcs}$).

The vector $\{m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}\}$ completely specifies a piecewise AR model. To estimate this vector `imsls_f_auto_parm` optimizes, with respect to this vector, one of two selection criteria: the first is a Minimum Description Length (MDL) criterion, and the second is the Akaike's Information Criterion (AIC). The MDL is defined as

$$\begin{aligned} MDL(m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}) \\ = \ln m + (m-1) \ln n + \sum_{j=1}^{m+1} \frac{2+p_j}{2} \ln n_j - \ln L \end{aligned}$$

while the AIC criterion is given by

$$\begin{aligned} AIC(m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}) &= 2(\text{number of parameters}) - 2 \ln L \\ &= 2(1 + m + \sum_{j=1}^{m+1} (2 + p_j)) - 2 \ln L \end{aligned}$$

where, given a candidate value of the vector $\{m, \tau_1, \dots, \tau_m, p_1, \dots, p_{m+1}\}$, L is the likelihood of the fitted piecewise AR model evaluated at the parameter estimates,

$$\{\hat{\gamma}_1, \hat{\phi}_{1,1}, \dots, \hat{\phi}_{1,p_1}, \hat{\sigma}_1^2, \dots, \hat{\gamma}_{m+1}, \hat{\phi}_{m+1,1}, \dots, \hat{\phi}_{m+1,p_{m+1}}, \hat{\sigma}_{m+1}^2\}.$$

The parameters $\{\gamma_j, \phi_{1,1}, \dots, \phi_{1,p_j}, \sigma_j^2\}$ of the j -th AR segment are estimated by the choice of one of three estimation methods: Yule–Walker, Burg, or Least Squares.

For simplicity, assume the mean of each series $\{X_{t,j}\}$ is 0 and that the errors are Gaussian. Then, the piecewise AR model has Gaussian likelihood

$$L = \prod_{j=1}^{m+1} (2\pi)^{-n_j/2} |V_j|^{-1/2} \exp\{-Y_j^T V_j Y_j\}$$

where V_j is the variance-covariance of the j -th AR segment (of order p_j) and Y_j is the vector of observations of the j -th segment, i.e., $Y_j = (Y_{\tau_j}, Y_{\tau_j+1}, \dots, Y_{\tau_{j+1}-1})^T$.

To find the minimizer $\{m, \tau_j, p_j\}^*$ of either MDL or AIC, `imsls_f_auto_parm` employs a Genetic Algorithm with islands, migration, cross-over and mutations. See Davis et.al. (2006) for further details.

Remarks

Function `imsls_f_auto_parm` approximates locally stationary time series by independent auto-regressive processes. Experimental results suggest that `imsls_f_auto_parm` gives reasonable estimates of the structural breaks of a given time series, even if the segment series are not autoregressive. Also, based on experimental results, MDL gives better results than AIC as a selection criterion.

If `seed` is set out of range, an informational (error) message is issued indicating that the seed will be reset to 123457. Also if `maxmig` migrations are reached in the genetic algorithm before the selection criterion value converges an informational message is issued suggesting the increase of `maxmig` or the use of the double precision function.

Example

The examples below illustrate different scenarios using `imsls_f_auto_parm`. The example series used in each case is the airline demand data (Box, Jenkins and Reinsel, 1994), which gives monthly total demand for the period January 1949 through December 1960. Each scenario sets the optional argument, `seed = 123457`.

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

```

int main()
{
#define N 144
    int n=N, npcs=0, iseed=123457, *arp=NULL, iper, iord, nlost;
    int maxarorder, *arpc=NULL, arp2[4]={0,2,59,1};
    float x[N], *arfit=NULL, *arstat=NULL, sc, dx[N];

    /* get data */
    imsls_f_data_sets(4, IMSLS_RETURN_USER, x, 0);

    /* Example 1: Use default values */
    printf ("Example 1: Use defaults\n\n");

    arp = imsls_f_auto_parm(n, x, &npcs,
        IMSLS_PRINT, 1,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_SELECTION_CRITERION_VALUE, &sc,
        IMSLS_AR_FIT, &arfit,
        IMSLS_AR_STATS, &arstat,
        0);

    imsls_free(arp);
    imsls_free(arfit);
    imsls_free(arstat);

    /* Example 2: differenced series set period for the difference.
    iper is in years for this data set */
    printf ("\n\nExample 2: differenced series\n\n");

    iper = 1;
    /* set the order for the difference. */
    iord = 1;
    /* get differenced series dx */

    imsls_f_difference(n, x, 1, &iper,
        IMSLS_ORDERS, &iord,
        IMSLS_LOST, &nlost,
        IMSLS_RETURN_USER, dx,
        0);

    arp = imsls_f_auto_parm(n-nlost, &dx[nlost], &npcs,
        IMSLS_PRINT, 1,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_SELECTION_CRITERION_VALUE, &sc,
        IMSLS_AR_FIT, &arfit,
        IMSLS_AR_STATS, &arstat,
        0);

    imsls_free(arp);
    imsls_free(arfit);
    imsls_free(arstat);

    /* Example 3: original series, lower order allowed
    lower maximum AR order */
    printf ("\n\nExample 3: original series, lower order allowed\n\n");

    maxarorder=5;

    arp = imsls_f_auto_parm(n, x, &npcs,
        IMSLS_MAX_AR_ORDER, maxarorder,

```

```

        IMSLS_PRINT, 1,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_SELECTION_CRITERION_VALUE, &sc,
        IMSLS_AR_FIT, &arfit,
        IMSLS_AR_STATS, &arstat,
        0);

    imsls_free(arp);
    imsls_free(arfit);
    imsls_free(arstat);

/* Example 4: differenced series, lower order allowed */
    printf("\n\nExample 4: differenced series, lower order allowed\n\n");

    maxarorder=5;

    arp = imsls_f_auto_parm(n-nlost, &dx[nlost], &npcs,
        IMSLS_MAX_AR_ORDER, maxarorder,
        IMSLS_PRINT, 1,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_SELECTION_CRITERION_VALUE, &sc,
        IMSLS_AR_FIT, &arfit,
        IMSLS_AR_STATS, &arstat,
        0);

    imsls_free(arp);
    imsls_free(arfit);
    imsls_free(arstat);

/* Example 5: original series, force fit the segments
Fit only at the break points */
    printf("\n\nExample 5: original series, force fit the segments\n\n");

    npcs=2;
    arpnnull = imsls_f_auto_parm(n, x, &npcs,
        IMSLS_AR_MODEL, arp2,
        IMSLS_PRINT, 1,
        IMSLS_RANDOM_SEED, iseed,
        IMSLS_SELECTION_CRITERION_VALUE, &sc,
        IMSLS_AR_FIT, &arfit,
        IMSLS_AR_STATS, &arstat,
        0);
}

```

Output

Example 1: Use defaults

```

===== final results =====
number of pieces:          2

selection criteria value: 684.243164

total time: 3.203000      conv:          1

===== final model estimates =====
break point  order  est. coeff.  likelihood  resid. var

```

arp[0]	arp[1]	arfit[0- 0]	arstat[0]	arstat[1]
0	1	0.77542	186.945	355.025
arp[2]	arp[3]	arfit[20-32]	arstat[2]	arstat[3]
43	13	1.03700		
		-0.07801		
		-0.03891		
		-0.03452		
		0.11961		
		-0.12851		
		0.01990		
		-0.04885		
		0.08089		
		-0.13117		
		0.22122		
		0.53862		
		-0.61515		
			486.666	691.486
Example 2: differenced series				
===== final results =====				
number of pieces:		1		
selection criteria value: 624.283508				
total time: 3.031000		conv: 1		
===== final model estimates =====				
break point	order	est. coeff.	likelihood	resid. var
arp[0]	arp[1]	arfit[0-11]	arstat[0]	arstat[1]
0	12	-0.02842		
		-0.22436		
		-0.16846		
		-0.24267		
		-0.10573		
		-0.22429		
		-0.12126		
		-0.26446		
		-0.07087		
		-0.24327		
		-0.07136		
		0.57129		
			619.321	297.352
Example 3: original series, lower order allowed				
===== final results =====				
number of pieces:		2		
selection criteria value: 705.296631				
total time: 2.312000		conv: 1		
===== final model estimates =====				
break point	order	est. coeff.	likelihood	resid. var
arp[0]	arp[1]	arfit[0- 0]	arstat[0]	arstat[1]
0	1	0.89533		

```

                270.393      333.563
arp[ 2]      arp[ 3]      arfit[ 5- 6]      arstat[ 2]      arstat[ 3]
  62          2          1.19788
                -0.35922
                424.270      1632.335

Example 4: differenced series, lower order allowed

===== final results =====
number of pieces:          2

selection criteria value: 698.359497

total time: 2.219000      conv:          1

===== final model estimates =====
break point  order      est. coeff.      likelihood      resid. var
arp[ 0]      arp[ 1]      arfit[ 0- 0]      arstat[ 0]      arstat[ 1]
  0          0          -----
                335.565      357.388
arp[ 2]      arp[ 3]      arfit[ 5- 5]      arstat[ 2]      arstat[ 3]
  76          1          0.33310
                352.175      1786.345

Example 5: original series, force fit the segments

===== final results =====
number of pieces: 2
selection criteria value: 712.521
===== final model estimates =====
break point  order      est. coeff.      likelihood      resid. var
arp[ 0]      arp[ 1]      arfit[ 0- 1]      arstat[ 0]      arstat[ 1]
  0          2          1.12156
                -0.24876
                258.192      313.889
arp[ 2]      arp[ 3]      arfit[20-20]      arstat[ 2]      arstat[ 3]
  59          1          0.88605
                443.696      1937.633

```

Warning Errors

IMSLS_MAX_MIGRATIONS_EXCEEDED

“maxmig” migrations or “stopiters” iterations were reached in the genetic algorithm before the selection criterion value converged. Try increasing “maxmig”, “stopiters” or using the double precision routine.

bayesian_seasonal_adj

Decomposes a time series into trend, seasonal, and an error component.

Synopsis

```
#include <imsls.h>

float imsls_f_bayesian_seasonal_adj(int nobs, float w[ ], ..., 0)
```

The type *double* function is `imsls_d_bayesian_seasonal_adj`.

Required Arguments

int **nobs** (Input)
The number of equally spaced series values.

float **w[]** (Input)
An array of length **nobs** containing the stationary time series.

Return Value

The average Akaike Bayesian Information Criterion for the estimated model. NaN is returned on error.

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_bayesian_seasonal_adj(int nobs, float w[ ],
    IMSLS_TREND_ORDER, int t_order,
    IMSLS_SEASONAL_ORDER, int s_order,
    IMSLS_NUM_PREDICT, int n_predict,
    IMSLS_PERIOD, int period,
    IMSLS_SPAN, int span,
    IMSLS_RIGIDITY, float rigidity,
```

```
IMSLS_MODEL, int model,  
IMSLS_PRINT_LEVEL, int print_level,  
IMSLS_NONSEASONAL_TREND, float **trend,  
IMSLS_NONSEASONAL_TREND_USER, float trend[ ],  
IMSLS_SEASONAL, float **seasonal,  
IMSLS_SEASONAL_USER, float seasonal[ ],  
IMSLS_IRREGULAR_COMPONENTS, float **irr_comp,  
IMSLS_IRREGULAR_COMPONENTS_USER, float irr_comp[ ],  
IMSLS_SERIES_SMOOTHED, float **smoothed,  
IMSLS_SERIES_SMOOTHED_USER, float smoothed[ ],  
0)
```

Optional Arguments

IMSLS_TREND_ORDER, *int* t_order (Input)

The order of trend differencing where $t_order \geq 0$.

Default: `t_order = 2`.

IMSLS_SEASONAL_ORDER, *int* s_order (Input)

The order of seasonal differencing where $s_order \geq 1$.

Default: `s_order = 1`.

IMSLS_NUM_PREDICT, *int* n_predict (Input)

The number of values to forecast where $n_predict \geq 0$.

Default: `n_predict = 0`.

IMSLS_PERIOD, *int* period (Input)

The number of seasons within a period where $period \geq 1$.

Default: `period = 12`.

IMSLS_SPAN, *int* span (Input)

The number of periods to be processed at one time where $1 \leq span \leq nobs/period$.

Default: `span = nobs/period`.

IMSLS_RIGIDITY, *float rigidity* (Input)

Controls the rigidity of the seasonal pattern where $0 \leq \text{rigidity} \leq 1.0$.

Default: `rigidity = 1.0`.

IMSLS_MODEL, *int model* (Input)

Model option.

<code>model</code>	Action
0	Non-additive logistic model
1	Log additive logistic model

Default: `model = 0`.

IMSLS_PRINT_LEVEL, *int print_level* (Input)

Printing option.

<code>print_level</code>	Action
0	No printing
1	Prints final results only
2	Prints intermediate and final results

Default: `print_level = 0`.

IMSLS_NONSEASONAL_TREND, *float **trend* (Output)

Address of a pointer to an internally allocated array of length `nobs + n_predict` containing the estimated trend component for each data value followed by the trend estimates for the `n_predict` forecasted values.

IMSLS_NONSEASONAL_TREND_USER, *float trend[]* (Output)

Storage array `trend` is provided by the user. See `IMSLS_NONSEASONAL_TREND`.

IMSLS_SEASONAL, *float **seasonal* (Output)

Address of a pointer to an internally allocated array of length `nobs + n_predict` containing the estimated seasonal components for each data value followed by the estimates for the `n_predict` forecasted seasonal values.

IMSLS_SEASONAL_USER, *float seasonal[]* (Output)

Storage array `seasonal` is provided by the user. See `IMSLS_SEASONAL` above.

IMSLS_IRREGULAR_COMPONENTS, *float **irr_comp* (Output)

Address of a pointer to an internally allocated array of length `nobs` containing the estimated irregular components.

`IMSLI_S_IRREGULAR_COMPONENTS_USER`, *float irr_comp[]* (Output)

Storage array `irr_comp` is provided by the user. See `IMSLI_S_IRREGULAR_COMPONENTS` above.

`IMSLI_S_SERIES_SMOOTHED`, *float **smoothed* (Output)

Address of a pointer to an internally allocated array of length `nobs + n_predict` containing the estimated smoothed component for each of the time series values followed by the `n_predict` forecast values.

`IMSLI_S_SERIES_SMOOTHED_USER`, *float smoothed[]* (Output)

Storage array `smoothed` is provided by the user. See `IMSLI_S_SERIES_SMOOTHED` above.

Description

Function `imsls_f_bayesian_seasonal_adj` is based upon the algorithm published by Akaike (1980). This algorithm uses a Bayesian approach to the problem of fitting the following autoregressive model for a time series W_t decomposed into a trend and a seasonal component.

Adopting the notation described earlier in the [Usage Notes](#) section of this chapter, if

$$t \in \mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

then a seasonal autoregressive model can be represented by the following relationship:

$$W_t = T_t + S_t + A_t$$

where W_t is the stationary time series with mean μ , T_t denotes an underlying trend, S_t denotes a seasonal component and A_t denotes a noise or irregular component.

A non-Bayesian approach to this problem would be to estimate the trend and seasonal components by minimizing

$$\sum_{t=1}^N (W_t - T_t - S_t)^2 + d^2 (\nabla^k T_t + r^2 \nabla_p^l S_t + z^2 (\sum_{j=0}^{p-1} S_{t-j}))^2$$

where the difference operator is denoted by ∇ and defined as $\nabla T_t = (T_t - T_{t-1})$, for $k \geq 1$, $\nabla^k T_t = \nabla(\nabla^{k-1} T_t)$, with $\nabla^0 T_t = T_t$. Similarly, $\nabla_p S_t = (S_t - S_{t-p})$ and $\nabla_p^l S_t = \nabla_p(\nabla_p^{l-1} S_t)$, $l \geq 1$, with $\nabla_p^0 S_t = S_t$.

The period of the seasonal component, p , the trend order, k , and the seasonal order l correspond to parameter options `IMSLI_S_PERIOD`, `IMSLI_S_TREND_ORDER`, and `IMSLI_S_SEASONAL_ORDER` respectively. d , z , and r are constants determined as follows.

In `imsls_f_bayesian_seasonal_adj`, the approach is to select the parameter ***d***, which controls the smoothness of the trend and seasonality estimates, using Bayesian methods. The prior distribution controls the smoothness of the trend and seasonal components by assuming low-order Gaussian autoregressive models for some differences of these components. The choice of the variance of the Gaussian distribution is realized by maximizing the log likelihood of the Bayesian model.

The other smoothing parameters, ***r*** and ***z***, are determined by the value of `rigidity`. The default value for `rigidity` is 1. Larger values of `rigidity` produce a more rigid seasonal pattern. Normally, a series is first fit using the default value for `rigidity`. The smoothness of the trend and seasonality estimates are examined and then `rigidity` is either increased or decreased depending upon whether more or less seasonal smoothing is needed.

Additionally, `imsls_f_bayesian_seasonal_adj` selects the optimum autoregressive model as the model that minimizes the Akaike Bayesian Information Criterion (ABIC).

$$ABIC = -2 \ln(\text{likelihood})$$

where the *likelihood* in this case is the mixed Bayesian maximum likelihood. Smaller values of *ABIC* represent a better fit. The basic minimization procedure is applied to blocks of data of length `span*period`. The final return value of the criterion is averaged over these blocks. By default, the data is treated in one block.

Example

This example uses unadjusted unemployment for women over 20 years of age in the U.S. for 1991-2001, as reported by the U.S. Bureau of Labor Statistics (www.bls.gov).

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

#define NDATA 132
#define NFOCAST 12

int main(){
    float y[]={2968.0, 3009.0, 2962.0, 2774.0, 3040.0, 3165.0,
        3104.0, 3313.0, 3178.0, 3142.0, 3129.0, 3107.0, 3397.0,
        3447.0, 3328.0, 3229.0, 3286.0, 3577.0, 3799.0, 3867.0,
        3655.0, 3360.0, 3310.0, 3369.0, 3643.0, 3419.0, 3108.0,
        3118.0, 3146.0, 3385.0, 3458.0, 3468.0, 3330.0, 3244.0,
        3135.0, 3005.0, 3462.0, 3272.0, 3275.0, 2938.0, 2894.0,
        3106.0, 3150.0, 3289.0, 3136.0, 2829.0, 2776.0, 2467.0,
        2944.0, 2787.0, 2749.0, 2762.0, 2578.0, 2900.0, 3100.0,
        3102.0, 2934.0, 2864.0, 2652.0, 2456.0, 3088.0, 2774.0,
        2701.0, 2555.0, 2677.0, 2741.0, 3052.0, 2966.0, 2772.0,
        2723.0, 2705.0, 2640.0, 2898.0, 2788.0, 2718.0, 2406.0,
        2520.0, 2645.0, 2708.0, 2811.0, 2666.0, 2380.0, 2292.0,
        2187.0, 2750.0, 2595.0, 2554.0, 2213.0, 2218.0, 2449.0,
        2532.0, 2639.0, 2449.0, 2326.0, 2302.0, 2065.0, 2447.0,
        2398.0, 2381.0, 2250.0, 2086.0, 2397.0, 2573.0, 2475.0,
        2299.0, 2054.0, 2127.0, 1935.0, 2425.0, 2245.0, 2298.0,
```

```

2005.0, 2208.0, 2379.0, 2459.0, 2539.0, 2182.0, 1959.0,
2012.0, 1834.0, 2404.0, 2329.0, 2285.0, 2175.0, 2245.0,
2492.0, 2636.0, 2892.0, 2784.0, 2771.0, 2878.0, 2856.0};

int nobs = NDATA, focast = NFOCAST;
float trend[NDATA+NFOCAST], seasonal[NDATA+NFOCAST];
float irr_comp[NDATA], abic=0.0;

char *months[] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"};
char *years[] = {"", "1991", "1992", "1993", "1994", "1995",
"1996", "1997", "1998", "1999", "2000", "2001", "2002"};

abic = imsls_f_bayesian_seasonal_adj(nobs, y,
IMSLS_TREND_ORDER, 2,
IMSLS_SEASONAL_ORDER, 1,
IMSLS_NUM_PREDICT, focast,
IMSLS_NONSEASONAL_TREND_USER, &trend,
IMSLS_SEASONAL_USER, &seasonal,
IMSLS_IRREGULAR_COMPONENTS_USER, &irr_comp,
0);

printf("Average ABIC = %f\n", abic);

imsls_f_write_matrix("TREND with last 12 values forecasted",
12, 12, trend,
IMSLS_TRANSPOSE,
IMSLS_ROW_LABELS, months,
IMSLS_COL_LABELS, years,
IMSLS_WRITE_FORMAT, "%8.1f",
0);

imsls_f_write_matrix("SEASONAL with last 12 values forecasted",
12, 12, seasonal,
IMSLS_TRANSPOSE,
IMSLS_ROW_LABELS, months,
IMSLS_COL_LABELS, years,
IMSLS_WRITE_FORMAT, "%6.1f",
0);

imsls_f_write_matrix("IRREGULAR=Original data-TREND-SEASONAL",
11, 12, irr_comp,
IMSLS_TRANSPOSE,
IMSLS_ROW_LABELS, months,
IMSLS_COL_LABELS, years,
IMSLS_WRITE_FORMAT, "%6.1f",
0);
}

```

Output

Average ABIC = 1297.640259

	TREND with last 12 values forecasted						
	1991	1992	1993	1994	1995	1996	1997
Jan	2879.8	3318.9	3422.6	3228.7	2827.3	2795.7	2743.1
Feb	2918.2	3359.9	3387.8	3206.0	2815.8	2785.6	2720.6

Mar	2955.1	3399.1	3355.5	3177.0	2812.4	2777.9	2694.2		
Apr	2990.0	3436.1	3329.5	3142.1	2814.7	2773.2	2665.2		
May	3022.7	3469.0	3309.6	3103.9	2819.3	2771.3	2636.0		
Jun	3052.8	3496.1	3294.8	3064.8	2825.5	2771.0	2607.4		
Jul	3082.8	3514.5	3283.8	3025.7	2830.9	2772.4	2580.8		
Aug	3116.2	3521.9	3276.0	2987.3	2833.2	2773.6	2557.2		
Sep	3153.4	3517.8	3270.3	2949.0	2831.7	2774.7	2536.3		
Oct	3193.8	3503.7	3264.8	2911.3	2826.1	2774.5	2518.0		
Nov	3235.6	3482.4	3256.8	2876.6	2816.6	2770.4	2503.0		
Dec	3277.6	3455.2	3245.3	2847.6	2805.9	2760.2	2491.5		
	1998	1999	2000	2001	2002				
Jan	2481.3	2352.1	2235.9	2206.0	3166.1				
Feb	2469.7	2345.6	2237.6	2239.0	3275.1				
Mar	2455.4	2338.1	2239.3	2281.4	3384.1				
Apr	2439.0	2328.3	2238.9	2333.1	3493.0				
May	2422.8	2315.5	2235.2	2393.9	3602.0				
Jun	2408.5	2301.4	2226.3	2464.7	3710.9				
Jul	2396.8	2286.0	2212.8	2545.7	3819.9				
Aug	2387.7	2269.8	2196.6	2636.7	3928.9				
Sep	2380.3	2255.5	2181.3	2735.7	4037.8				
Oct	2373.6	2244.6	2171.9	2840.4	4146.8				
Nov	2366.6	2238.3	2172.1	2948.2	4255.8				
Dec	2359.1	2235.6	2183.3	3057.2	4364.7				
SEASONAL with last 12 values forecasted									
	1991	1992	1993	1994	1995	1996	1997	1998	1999
Jan	162.9	165.6	169.3	172.0	173.8	176.3	176.4	177.3	176.4
Feb	51.4	51.5	50.5	49.5	48.6	48.8	50.0	51.1	51.0
Mar	-24.0	-23.9	-23.4	-18.8	-16.3	-13.0	-8.7	-4.9	-3.0
Apr	-191.0	-190.1	-189.1	-188.0	-186.6	-187.8	-188.5	-187.9	-186.6
May	-140.6	-143.3	-145.4	-147.4	-148.1	-147.1	-147.1	-147.9	-147.7
Jun	67.2	66.6	65.8	64.4	63.3	62.2	62.7	63.6	64.9
Jul	176.9	180.1	181.6	183.0	185.5	186.6	185.8	186.1	187.2
Aug	251.9	253.0	252.6	253.3	253.1	252.8	253.7	254.4	255.2
Sep	76.4	77.2	77.1	77.3	75.5	73.3	72.6	70.7	68.9
Oct	-79.5	-80.5	-80.1	-80.8	-81.5	-84.3	-87.6	-90.1	-93.4
Nov	-119.8	-120.7	-120.5	-120.2	-120.4	-119.7	-119.7	-118.1	-117.6
Dec	-235.7	-237.9	-243.0	-247.9	-250.5	-251.2	-254.2	-256.2	-257.5
	2000	2001	2002						
Jan	177.2	177.6	177.5						
Feb	50.8	51.6	51.5						
Mar	-1.9	-1.7	-1.8						
Apr	-187.2	-186.5	-186.6						
May	-145.7	-145.6	-145.7						
Jun	65.6	65.1	65.0						
Jul	186.4	184.7	184.7						
Aug	256.8	256.9	256.9						
Sep	67.3	67.0	67.0						
Oct	-95.1	-94.6	-94.6						
Nov	-117.3	-116.6	-116.6						
Dec	-258.1	-257.1	-257.1						
IRREGULAR=Original data-TREND-SEASONAL									
	1991	1992	1993	1994	1995	1996	1997	1998	1999
Jan	-74.7	-87.5	51.1	61.2	-57.1	116.0	-21.5	91.4	-81.5
Feb	39.4	35.6	-19.3	16.5	-77.4	-60.4	17.3	74.2	1.5
Mar	30.9	-47.2	-224.1	116.8	-47.1	-63.9	32.5	103.5	45.9
Apr	-25.0	-17.0	-22.5	-16.1	133.9	-30.5	-70.6	-38.0	108.3

May	157.8	-39.7	-18.2	-62.6	-93.1	52.8	31.2	-56.9	-81.9
Jun	45.0	14.3	24.4	-23.1	11.2	-92.1	-25.1	-23.2	30.7
Jul	-155.6	104.4	-7.4	-58.6	83.6	93.0	-58.6	-50.9	99.9
Aug	-55.0	92.1	-60.6	48.4	15.7	-60.4	0.1	-3.1	-49.9
Sep	-51.8	60.0	-17.4	109.6	26.8	-76.0	57.1	-2.1	-25.3
Oct	27.7	-63.3	59.3	-1.5	119.4	32.8	-50.3	42.5	-97.1
Nov	13.2	-51.7	-1.3	19.5	-44.3	54.3	-91.3	53.5	6.3
Dec	65.1	151.7	2.7	-132.8	-99.4	131.0	-50.3	-37.9	-43.1
	2000	2001							
Jan	11.9	20.4							
Feb	-43.5	38.4							
Mar	60.5	5.3							
Apr	-46.7	28.5							
May	118.5	-3.3							
Jun	87.1	-37.7							
Jul	59.9	-94.4							
Aug	85.6	-1.6							
Sep	-66.6	-18.7							
Oct	-117.8	25.3							
Nov	-42.7	46.4							
Dec	-91.2	56.0							

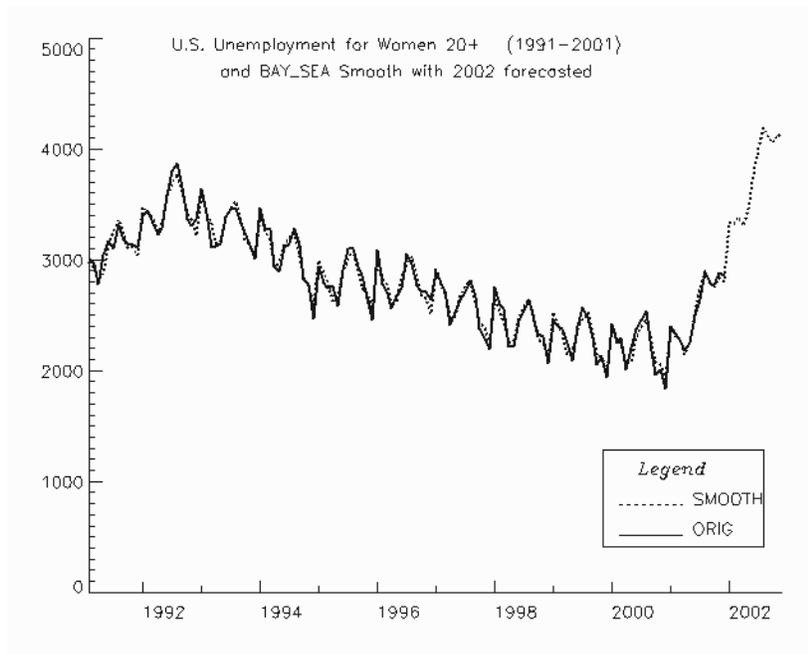


Figure 8, Sample Smoothed Predictions from bayesian_seasonal_adj

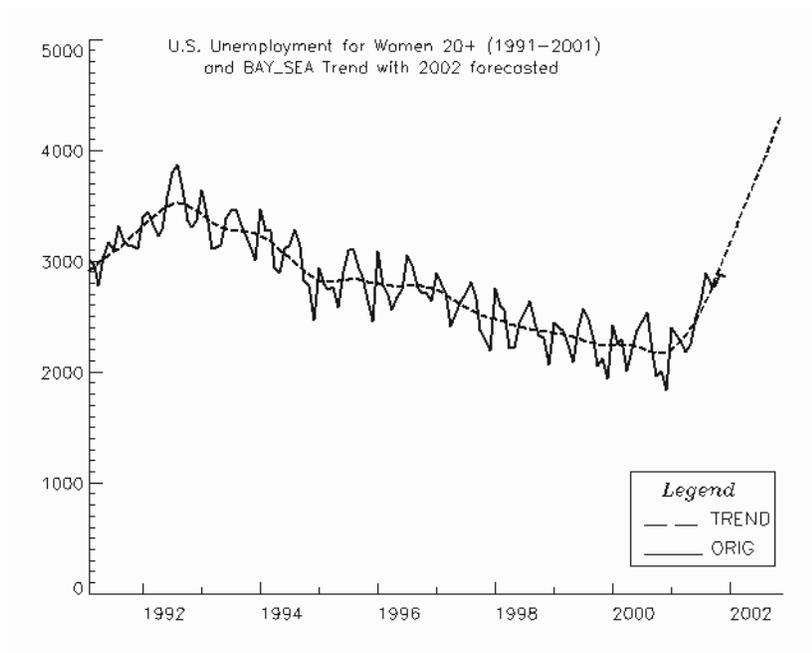


Figure 9, Sample Trend Predictions from bayesian_seasonal_adj

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 `imsls_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

`IMSL_S_SHIFT`, *float shift* (Input)

Shift parameter in the Box-Cox (power) transformation. Parameter `shift` must satisfy the relation $\min(z(t)) + \text{shift} > 0$.

Default: `shift = 0.0`.

`IMSL_S_INVERSE_TRANSFORM` (Input)

If `IMSL_S_INVERSE_TRANSFORM` is specified, the inverse transform is performed.

`IMSL_S_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 $n = \text{n_observations}$ observations $\{Z_t\}$ for $t = 1, 2, \dots, n$.

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

$$X_t = \begin{cases} \frac{(Z_t + \xi)^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \ln(Z_t + \xi) & \lambda = 0 \end{cases}$$

where $Z_t + \xi > 0$ for all t . Since

$$\lim_{\lambda \rightarrow 0} \frac{(Z_t + \xi)^\lambda - 1}{\lambda} = \ln(Z_t + \xi)$$

the family of power transformations is continuous.

Let $\lambda = \text{power}$ and $\xi = \text{shift}$; 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>

int 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);

    imsls_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 original data values.

```
#include <imsls.h>

int 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);

    imsls_free(x);
    imsls_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 $i = 1, \dots, \text{"n_observations"}$. "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[#]" = #. $\exp(\text{"z[#]"})$ will underflow.
IMSLS_BCTR_I_OVERFLOW	Inverse transformation. "power" = #. The maximum element of "z[#]" = #. $\exp(\text{"z[#]"})$ will overflow.
IMSLS_BCTR_I_ABS_UNDERFLOW	Inverse transformation. "power" = #. The element of "z" with the smallest absolute value is "z[#]" = #. $\text{"z[#]" } ^ {1/\text{"power"}}$ will underflow.
IMSLS_BCTR_I_ABS_OVERFLOW	Inverse transformation. "power" = #. The element of "z" with the largest absolute value is "z[#]" = #. $\text{"z[#]" } ^ {1/\text{"power"}}$ will overflow.

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* *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 (Input)

or

IMSLS_SET_FIRST_TO_NAN (Input)

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

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:

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

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

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

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

$$\Delta_s^d Z_t$$

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

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

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

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` 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 $t = 14, 15, \dots, 24$.

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

int 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

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.

autocorrelation



[more...](#)

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 $k = 1, \dots, \text{lagmax}$.

Synopsis with Optional Arguments

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

```
float *imsls_f_autocorrelation (int n_observations, float x[ ], int lagmax,
```

```

IMSLS_PRINT_LEVEL, int iprint,
IMSLS_X_MEAN_IN, float x_mean_in,
IMSLS_X_MEAN_OUT, float *x_mean_out,
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_RETURN_USER, float autocorrelations[ ],
0)

```

Optional Arguments

IMSLS_PRINT_LEVEL, *int iprint* (Input)
Printing option.

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.

Default = 0.

IMSLS_X_MEAN_IN, *float x_mean_in* (Input)
User input estimate of the mean of the time series **x**.

IMSLS_X_MEAN_OUT, *float *x_mean_out* (Output)
If specified, **x_mean_out** is the computed arithmetic mean of the time series **x**.

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`.

<code>se_option</code>	
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_RETURN_USER`, *float* `autocorrelations` [] (Output)

If specified, `autocorrelations` is 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 $k = 1, \dots, \text{lagmax}$.

Description

Function `imsls_f_autocorrelation` estimates the autocorrelation function of a stationary time series given a sample of $n = \text{n_observations}$ observations $\{X_t\}$ for $t = 1, 2, \dots, n$.

Let $\hat{\mu}$ be the estimate of the mean μ of the time series $\{X_t\}$ where

$$\hat{\mu} = \begin{cases} \mu, & \mu \text{ known (x_mean_in)} \\ \frac{1}{n} \sum_{t=1}^n X_t & \mu \text{ unknown (x_mean_out)} \end{cases}$$

The autocovariance function $\sigma(k)$ is estimated by

$$\hat{\sigma}(k) = \frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu})(X_{t+k} - \hat{\mu}), \quad k = 0, 1, \dots, K$$

where $K = \text{lagmax}$. Note that

$$\hat{\sigma}(0)$$

is an estimate of the sample variance. The autocorrelation function $\rho(k)$ is estimated by

$$\hat{\rho}(k) = \frac{\hat{\sigma}(k)}{\hat{\sigma}(0)}, \quad k = 0, 1, \dots, K$$

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 `IMSLI5_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

$$\text{var}\{\hat{\rho}(k)\} = \frac{1}{n} \sum_{i=-\infty}^{\infty} \left[\rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k) \right]$$

where

$$\hat{\rho}(k)$$

assumes μ is unknown. For computational purposes, the autocorrelations $r(k)$ are replaced by their estimates

$$\hat{\rho}(k)$$

for $|k| \leq K$, 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

$$\text{var}\{\hat{\rho}(k)\} = \frac{n-k}{n(n+2)}$$

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>

int 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

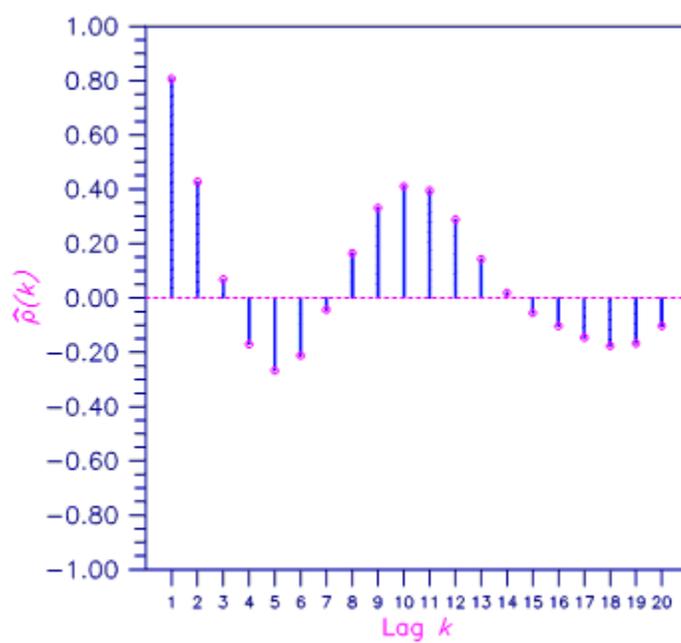


Figure 10, Sample Autocorrelation Function

crosscorrelation



[more...](#)

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 $2 \times \text{lagmax} + 1$ 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 $\text{lag}(k - \text{lagmax})$ where $k = 0, 1, \dots, 2 * \text{lagmax}$. To release this space, use `imsls_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_PRINT_LEVEL, int iprint,
    IMSLS_INPUT_MEANS, float x_mean_in, float y_mean_in,
    IMSLS_OUTPUT_MEANS, float *x_mean_out, float *y_mean_out,
    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_RETURN_USER, float crosscorrelations[ ],
    0)
```

Optional Arguments

IMSLS_PRINT_LEVEL, int iprint (Input)

Printing option.

iprint	
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, cross-correlations, and standard errors of cross-correlations.

Default = 0.

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**.

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 $2 \times \text{lagmax} + 1$ containing 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	
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 $2 \times \text{lagmax} + 1$ 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 $2 \times \text{lagmax} + 1$ 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 $(k - \text{lagmax})$, where $k = 0, 1, \dots, 2 \times \text{lagmax}$.

IMSLS_CROSS_COVARIANCES_USER, *float cross_covariances[]* (Output)

If specified, **cross_covariances** is a user-specified array of length $2 \times \text{lagmax} + 1$ containing the cross-covariances between the time series **x** and **y**. See **IMSLS_CROSS_COVARIANCES**.

IMSLS_RETURN_USER, *float crosscorrelations[]* (Output)

If specified, **crosscorrelations** is an array of length $2 \times \text{lagmax} + 1$ 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 $(k - \text{lagmax})$ where $k = 0, 1, \dots, 2 \times \text{lagmax}$.

Description

Function **imsls_f_crosscorrelation** estimates the cross-correlation function of two jointly stationary time series given a sample of $n = \text{n_observations}$ observations $\{X_t\}$ and $\{Y_t\}$ for $t = 1, 2, \dots, n$.

Let $\hat{\mu}_x$ be the estimate of the mean μ_x of the time series $\{X_t\}$ where

$$\hat{\mu}_X = \begin{cases} \mu_X & \mu_X \text{ known (x_mean_in)} \\ \frac{1}{n} \sum_{t=1}^n X_t & \mu_X \text{ unknown (x_mean_out)} \end{cases}$$

The autocovariance function of $\{X_t\}$, $\sigma_x(k)$, is estimated by

$$\hat{\sigma}_X(k) = \frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu}_X)(X_{t+k} - \hat{\mu}_X), \quad k = 0, 1, \dots, K$$

where $K = \text{lagmax}$. Note that

$$\hat{\sigma}_X(0)$$

is equivalent to the sample variance `x_variance`. The autocorrelation function $\rho_x(k)$ is estimated by

$$\hat{\rho}_X(k) = \frac{\hat{\sigma}_X(k)}{\hat{\sigma}_X(0)} \quad k = 0, 1, \dots, K$$

Note that

$$\hat{\rho}_X(0) \equiv 1$$

by definition. Let

$$\hat{\mu}_Y, \hat{\sigma}_Y(k), \text{ and } \hat{\rho}_Y(k)$$

be similarly defined.

The cross-covariance function $\sigma_{XY}(k)$ is estimated by

$$\hat{\sigma}_{XY}(k) = \begin{cases} \frac{1}{n} \sum_{t=1}^{n-k} (X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \dots, K \\ \frac{1}{n} \sum_{t=1-k}^n (X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \dots, -K \end{cases}$$

The cross-correlation function $\rho_{XY}(k)$ is estimated by

$$\hat{\rho}_{XY}(k) = \frac{\hat{\sigma}_{XY}(k)}{[\hat{\sigma}_X(0)\hat{\sigma}_Y(0)]^{1/2}} \quad k = 0, \pm 1, \dots, \pm K$$

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

$$\begin{aligned} \text{var}\{\hat{\rho}_{XY}(k)\} = & \frac{1}{n-k} \sum_{i=-\infty}^{\infty} \left[\rho_X(i)\rho_Y(i) + \rho_{XY}(i-k)\rho_{XY}(i+k) \right. \\ & \left. - 2\rho_{XY}(k) \left\{ \rho_X(i)\rho_{XY}(i+k) + \rho_{XY}(-i)\rho_Y(i+k) \right\} \right. \\ & \left. + \rho_{XY}^2(k) \left\{ \rho_X(i) + \frac{1}{2}\rho_X^2(i) + \frac{1}{2}\rho_Y^2(i) \right\} \right] \end{aligned}$$

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 $|k| \leq K$, 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

$$\text{var}\{\hat{\rho}_{XY}(k)\} = \frac{1}{n-k} \sum_{i=-\infty}^{\infty} \rho_X(i)\rho_Y(i) \quad k \geq 0$$

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 $k \geq 0$. 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 -10 through lag 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

int 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

[more...](#)

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 $n_channel_x \times n_channel_y \times (2 \times lagmax + 1)$ 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

$i = 1, \dots, n_channel_x$

$j = 1, \dots, n_channel_y$

$k = 0, 1, \dots, 2 \times lagmax$, and

$m = (n_channel_x \times n_channel_y \times k + (i \times n_channel_x + j))$

To release this space, use **imsls_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_PRINT_LEVEL, int iprint,
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[],
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_RETURN_USER, float crosscorrelations[],
0)
```

Optional Arguments

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)

Printing option.

<code>iprint</code>	
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.

Default = 0.

`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`.

`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 $n_channel_x \times n_channel_y \times (2 \times lagmax + 1)$ containing the cross-covariances between the channels of \mathbf{x} and \mathbf{y} . The m th element of this array contains the cross-covariance between channel i of the \mathbf{x} series and channel j of the \mathbf{y} series at lag $(k - lagmax)$ where

$i = 1, \dots, n_channel_x$

$j = 1, \dots, n_channel_y$

$k = 0, 1, \dots, 2 * lagmax$, 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 $n_channel_x \times n_channel_y \times (2 \times lagmax + 1)$ containing the cross-covariances between the channels of \mathbf{x} and \mathbf{y} . See `IMSLS_CROSS_COVARIANCES`.

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

If specified, `crosscorrelations` is a user-specified array of length $n_channel_x \times n_channel_y \times (2 \times lagmax + 1)$ containing the cross-correlations between the channels of \mathbf{x} and \mathbf{y} . See Return Value.

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

$$X = (X_1, X_2, \dots, X_p)$$

where

$$X_j = (X_{1j}, X_{2j}, \dots, X_{nj})^T, \quad j = 1, 2, \dots, p$$

with $n = n_observations_x$ and $p = n_channel_x$. Similarly, define the multichannel time series Y by

$$Y = (Y_1, Y_2, \dots, Y_q)$$

where

$$Y_j = (Y_{1j}, Y_{2j}, \dots, Y_{mj})^T, \quad j = 1, 2, \dots, q$$

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 mul-

tivariate 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 $\hat{\mu}_X$ be the row vector containing the means of the channels of X . In particular,

$$\hat{\mu}_X = \left(\hat{\mu}_{X_1}, \hat{\mu}_{X_2}, \dots, \hat{\mu}_{X_p} \right)$$

where for $j = 1, 2, \dots, p$

$$\hat{\mu}_{X_j} = \begin{cases} \mu_{X_j} & \mu_{X_j} \text{ known (x_mean_in)} \\ \frac{1}{n} \sum_{t=1}^n X_{tj} & \mu_{X_j} \text{ unknown (x_mean_out)} \end{cases}$$

Let $\hat{\mu}_Y$ be similarly defined for the means of the channels of Y . The cross-covariance of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\sigma}_{X_i Y_j}(k) = \begin{cases} \frac{1}{N} \sum_t \left(X_{ti} - \hat{\mu}_{X_i} \right) \left(Y_{t+k,j} - \hat{\mu}_{Y_j} \right) & k = 0, 1, \dots, K \\ \frac{1}{N} \sum_t \left(X_{ti} - \hat{\mu}_{X_i} \right) \left(Y_{t+k,j} - \hat{\mu}_{Y_j} \right) & k = -1, -2, \dots, -K \end{cases}$$

where $i = 1, \dots, p, j = 1, \dots, q$, and $K = \mathbf{lagmax}$. The summation on t extends over all possible cross-products with N equal to the number of cross-products in the sum

Let

$$\hat{\sigma}_X(\mathbf{0}) = \mathbf{x_variance}$$

be the row vector consisting of the estimated variances of the channels of X . In particular,

$$\hat{\sigma}_X(\mathbf{0}) = \left(\hat{\sigma}_{X_1}(\mathbf{0}), \hat{\sigma}_{X_2}(\mathbf{0}), \dots, \hat{\sigma}_{X_p}(\mathbf{0}) \right)$$

where

$$\hat{\sigma}_{X_j}(\mathbf{0}) = \frac{1}{n} \sum_{t=1}^n \left(X_{tj} - \hat{\mu}_{X_j} \right)^2 \quad j = 1, 2, \dots, p$$

Let

$$\hat{\sigma}_Y(0) = y_variance$$

be similarly defined. The cross-correlation of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\rho}_{X_i Y_j}(k) = \frac{\hat{\sigma}_{X_i Y_j}(k)}{\left[\hat{\sigma}_{X_i}(0) \hat{\sigma}_{Y_j}(0) \right]^{1/2}} \quad k = 0, \pm 1, \dots, \pm K$$

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 -10 through 10 .

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

int 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 $K = \text{lagmax}$ sample autocorrelations

$$\hat{\rho}(k)$$

for $k = 0, 1, \dots, K$. Consider the AR(k) process defined by

$$X_t = \phi_{k1}X_{t-1} + \phi_{k2}X_{t-2} + \dots + \phi_{kk}X_{t-k} + A_t$$

where ϕ_{kj} denotes the j -th coefficient in the process. The set of estimates

$$\left\{ \hat{\phi}_{kk} \right\}$$

for $k = 1, \dots, K$ is the sample partial autocorrelation function. The autoregressive parameters

$$\left\{ \hat{\phi}_{kj} \right\}$$

for $j = 1, \dots, k$ are approximated by Yule-Walker estimates for successive AR(k) models where $k = 1, \dots, K$. Based on the sample Yule-Walker equations

$$\hat{\rho}(j) = \hat{\phi}_{k1}\hat{\rho}(j-1) + \hat{\phi}_{k2}\hat{\rho}(j-2) + \dots + \hat{\phi}_{kk}\hat{\rho}(j-k), \quad j = 1, 2, \dots, k$$

a recursive relationship for $k = 1, \dots, K$ was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & k = 1 \\ \frac{\hat{\rho}(k) - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(k-j)}{1 - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j} \hat{\rho}(j)} & k = 2, \dots, K \end{cases}$$

and

$$\hat{\phi}_{kj} = \begin{cases} \hat{\phi}_{k-1,j} - \hat{\phi}_{kk} \hat{\phi}_{k-1,k-j} & j = 1, 2, \dots, k-1 \\ \hat{\phi}_{kk} & j = k \end{cases}$$

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(ρ), Box and Jenkins (1976, page 65) note

$$\text{var}\{\hat{\phi}_{kk}\} \simeq \frac{1}{n} \quad k \geq p + 1$$

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. Function `imsls_f_partial_autocorrelation` is used to compute the estimated partial autocorrelations.

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

int 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` for an ARMA(*p,q*) model.

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_LAGMIN, int lagmin,
```

```
IMSLS_RETURN_USER, float stat [ ],
0)
```

Optional Arguments

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.

IMSLS_RETURN_USER, *float stat []* (Output)

User defined array for storage of lack-of-fit statistics.

Description

Function `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:

	LAGMIN	LAGMAX	NPFREE
ARMA (p, q)	1	$\sqrt{n_observations}$	$p + q$
Transfer function	0	$\sqrt{n_observations}$	$r + s$

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 $k = L, L + 1, \dots, K$ where $L = \mathbf{lagmin}$ and $K = \mathbf{lagmax}$.

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 $m = \text{npfree}$ 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 $\text{npfree} = p + q$ 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 function `imsls_f_arma`. The autocorrelations of the residuals are estimated using function `imsls_f_autocorrelation`. A portmanteau lack of fit test is computed using 10 lags with `imsls_f_lack_of_fit`.

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

int main()
{
    int p=2, q=1, i, n_observations=100, max_iteations=0, lagmin=1,
        lagmax=10, npfree=4;
    float data[176][2], x[100], *parameters, *correlations,
        *residuals, *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_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", result[0]);  
printf("          P-value of Q = \t %1.5f\n\n", result[1]);  
}
```

Output

```
Lack of Fit Statistic, Q = 23.89239  
          P-value of Q = 0.00055
```

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, \dots, 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, \dots, t_{n_obs} , then missing values between t_i and t_{i+1} , $i = 1, \dots, n_obs - 1$, 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_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:

	Description
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.

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 $2 \cdot \text{maxlag}$, then `maxlag` is reduced to `len/2` for the computation of the missing values within this gap.

Default: `method = 3`

`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_RELATIVE_ERROR`, *float* `relative_error` (Input)

Stopping criterion for use in the nonlinear equation solver used by `method 2`.

Default: `relative_error = 100 \times \text{imsls_f_machine}(4)` for single precision,
`relative_error = 100 \times \text{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`.

IMSL_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 $t_1, t_1 + 1, t_1 + 2, \dots, t_n$. 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 $t_1, t_1 + 1, \dots, t_m$ 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+1} is also missing then the values at time points $t_1, t_1 + 1, \dots, t_{m+1}$ 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 function `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 $\{0, 1, \dots, \text{max_lag}\}$ of possible values and to compute the parameters ϕ_1, \dots, ϕ_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 $y_{t_1}, y_{t_1+1}, \dots, y_{t_m}$ by μ the one-step-ahead forecast at origin t_m , $\hat{y}_{t_m}(1)$, can be computed by the formula

$$\hat{y}_{t_m}(1) = \mu \left(1 - \sum_{j=1}^p \phi_j \right) + \sum_{j=1}^p \phi_j y_{t_m+1-j}$$

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

$$Y_t = \phi_1 Y_{t-1} + a_t, \quad t = 1, 2, 3, \dots$$

We assume that $\{a_t\}$ is a Gaussian white noise process, $a_t \sim N(0, \sigma^2)$. Then, $E[Y_t] = 0$ and

$$\text{VAR}[Y_t] = \sigma^2 / (1 - \phi_1^2) \quad (\text{see Anderson, p. 174}).$$

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 $\text{VAR}[Y_t] = 1$. 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 <stdio.h>
#include <math.h>

int main()
{
    int i, j, k, maxlag = 20, times_1[200], times_2[200], ntemp,
        n_obs, n_miss, ntimes, miss_ind, *times = NULL,
        *missing_index = NULL;
    float x_1[200], x_2[200], *result = NULL;
    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, &times,
            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, &times,
            IMSLS_MAX_LAG, 20,
            IMSLS_TIMES_ARRAY, &times,
            IMSLS_MISSING_INDEX, &missing_index,
            0);
}

```

```

    if (!result) {
        if (times) {
            imsls_free(times);
            times = NULL;
        }
        if (missing_index) {
            imsls_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) {
        imsls_free(result);
        result = NULL;
    }
    if (times) {
        imsls_free(times);
        times = NULL;
    }
    if (missing_index) {
        imsls_free(missing_index);
        missing_index = NULL;
    }
}
}

```

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

hw_time_series



[more...](#)

Calculates parameters and forecasts using the Holt-Winters Multiplicative or Additive forecasting method for seasonal data.

Synopsis

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

```
float *imsls_f_hw_time_series (int nobs, int nseason, float y[ ], ..., 0)
```

The type *double* function is `imsls_d_hw_time_series`.

Required Arguments

int **nobs** (Input)

The number of equally spaced series values.

int **nseason** (Input)

The number of time points in a season, or the length of the season. The function requires that $2 \leq \mathbf{nseason} \leq \mathbf{nobs}$ unless one of the non-seasonal options is specified, in which case **nseason** is ignored. See the optional arguments for details.

float **y[]** (Input)

An array of length **nobs** containing the values of the time series.

Return Value

Pointer to an array of length $(\mathbf{nobs} + 1) * 3$ containing the values of the smoothing parameters followed by the *level*, *trend*, and *seasonal* component sequences. Note that if `IMSLS_NONSEASONAL_TREND` is specified, the array is of length $(\mathbf{nobs} + 1) * 2$ and if `IMSLS_NONSEASONAL_CONSTANT` is specified, it is of length $(\mathbf{nobs} + 1)$.

Synopsis with Optional Arguments

```

#include <imsls.h>

float *imsls_f_hw_time_series (int nobs, int nseason, float y[ ],

    IMSLS_ADDITIVE, int add,

    IMSLS_SERIES_INCREMENT, int incy,

    IMSLS_NONSEASONAL_TREND,

    IMSLS_NONSEASONAL_CONSTANT,

    IMSLS_USE_PARAMS, float params[ ],

    IMSLS_N_SAMPLE, int nsamples,

    IMSLS_BOUNDS, float bounds[ ],

    IMSLS_INIT_INPUT, int init,

    IMSLS_FORECAST_CONFIDENCE, float confid,

    IMSLS_RETURN_USER, float sequences[ ],

    IMSLS_FORECASTS, int nforecasts, float **forecasts,

    IMSLS_FORECASTS_USER, int nforecasts, float forecasts[ ],

    IMSLS_SERIES_SMOOTHED, float **ysmoothed,

    IMSLS_SERIES_SMOOTHED_USER, float ysmoothed[ ],

    IMSLS_COV, float **cov,

    IMSLS_COV_USER, float cov[ ],

    IMSLS_SS_RESIDUAL, float *sumofsquares,

    0)

```

Optional Arguments

IMSLS_ADDITIVE, *int add* (Input)

Specifies the use of the Multiplicative or Additive time series model. **add** must be 1 or 0. If **add** = 1 the Additive model is used.

Default: **add** = 0. The Multiplicative model is the default.

IMSL_Series_Increment, *int* `incy` (Input)

The constant stride through the series data y . The length of y must be at least $(\text{nobs}-1) \times |\text{incy}| + 1$. When $\text{incy} < 0$, the series is incremented in reverse order beginning at index $\text{nobs} * (-\text{incy}) - 1$.

Default: `incy = 1`.

IMSL_Nonseasonal_Trend, (Input)

Remove the seasonal component and fit only the level and trend. If present, the models involve only the *level* (α) and *trend* (β) parameters. The method is equivalent to double exponential smoothing.

Default: The method includes all three components

IMSL_Nonseasonal_Constant, (Input)

Remove the trend and the seasonal components and fit only the level. If present, the models involve only the *level* (α) parameter. The method is simple exponential smoothing.

Default: The method includes all three components.

IMSL_Use_Params, *float* `params []` (Input)

An array containing the values of the smoothing parameters for the *level* (α), the *trend* (β), and the *seasonal* (γ) component sequences. The array should be length 3 unless

IMSL_Nonseasonal_Trend is specified, in which case it is of length 2 containing values for *level* (α) and the *trend* (β). Likewise, if **IMSL_Nonseasonal_Constant** is specified, `params` is of length 1 and contains the value for the level parameter (α) only.

Default: Parameter values are selected by minimizing the mean squared one step ahead forecast error.

IMSL_N_Sample, *int* `nsamples` (Input)

Number of evaluations of the residual norm that are sampled to obtain starting values for the smoothing parameters, (α , β , γ).

Default: `nsamples = nobs`.

IMSL_Bounds, *float* `bounds []` (Input)

An array of length 6 that contains the lower and upper bounds for each of the smoothing parameters, (α , β , γ). The three lower bounds are followed by the 3 upper bounds. Note that the lower and upper bounds must be in the interval $[0,1]$, inclusive. The array is ignored if **IMSL_Use_Params** is used.

Default: Lower bounds are all 0 and upper bounds are all 1.

IMSLS_INIT_INPUT, *int* **init** (Input)

init must be 1 or 0. If **init** = 1, the initial values for the *level*, *trend*, and *seasonal* component sequences are provided in the user array, **sequences**. The values must be stored in rows 1, 2, ..., **nseason** of the return array. See **IMSLS_RETURN_USER** and the [Description](#) section for more information.

Default: **init** = 0. Initial values are computed by the function.

IMSLS_FORECAST_CONFIDENCE, *float* **confid** (Input)

This value provides the confidence level to use in the calculation of the prediction intervals. If this argument is present and valid ($0.0 < \text{confid} < 100.0$), prediction intervals are provided for each forecast.

Default: Prediction intervals are not provided.

IMSLS_RETURN_USER, *float* **sequences** [] (Input/Output)

Storage for the return value is provided by the user. When **IMSLS_INIT_INPUT** is set to 1, **sequences** must contain initial values for the component sequences on input in rows 1, 2, ..., **nseason** of the array. Rows 0 and **nseason** + 1 to **nobs** are ignored on input. See the [Description](#) section for the required format of the array.

IMSLS_FORECASTS, *int* **nforecasts**, *float* ****forecasts**

int **nforecasts** (Input)

The number of forecasts desired past the series data.

float ****forecasts** (Output)

Address of a pointer to an internally allocated array of length **nforecasts**. The value of the *i*-th row is the forecast (*i* + 1) steps past the series data. If the option **IMSLS_FORECAST_CONFIDENCE** is used, the array will be of length **nforecasts** × 3 and the value of the *i*-th row is the forecast (*i*+1) steps ahead followed by the prediction interval lower and upper bounds.

IMSLS_FORECASTS_USER, *int* **nforecasts**, *float* **forecasts** []

int **nforecasts** (Input)

The number of forecasts desired past the series data.

float **forecasts** [] (Output)

Storage for array **forecasts** is provided by the user. See **IMSLS_FORECASTS**.

`IMSL_Series_Smoothed`, *float **ysmoothed* (Output)

The address of a pointer to the internally allocated array of length `nobs` containing the fitted series values.

`IMSL_Series_Smoothed_User`, *float ysmoothed[]* (Output)

Storage for array `ysmoothed` is provided by the user. See `IMSL_Series_Smoothed`.

`IMSL_Cov`, *float **cov* (Output)

The address of a pointer to an internally allocated array containing the variance-covariance matrix of the smoothing parameters estimated by minimizing the mean squared forecast error. The length of the array is 3×3 unless `IMSL_Nonseasonal_Trend` is specified, in which case it is 2×2 , or unless `IMSL_Nonseasonal_Constant` is specified, in which case it is 1×1 . `IMSL_Cov` is ignored when the option `IMSL_Use_Params` is present.

`IMSL_Cov_User`, *float cov[]* (Output)

Storage for array `cov` is provided by the user. See `IMSL_Cov`.

`IMSL_SS_Residual`, *float *sumofsquares* (Output)

The sum of squares of the one step ahead forecast errors.

Description

Function `imsls_f_hw_time_series` performs the Holt-Winters forecasting method to an equally spaced time series, $\{y_t\}$ where $N = \text{nobs}$ and $t = 1, \dots, N$ (or $t = 0, 1 \cdot \text{incy}, 2 \cdot \text{incy}, \dots, K \cdot \text{incy}$ where $K \cdot \text{incy} \leq N$ and $\text{incy} \geq 1$). The Holt-Winters procedure fits three component sequences known as the *level*, the *trend*, and the *seasonal* sequence. There are two formulations, depending on whether the seasonal component of the time series is thought to be additive or multiplicative. The seasonal component depends on the length of the season, `nseason` = s , where $s = 2, \dots, N$.

Holt-Winters Additive Model

$L_t = \alpha(y_t - S_{t-s}) + (1 - \alpha)(L_{t-1} + b_{t-1})$	the <i>level</i> sequence
$b_t = \beta(L_t - L_{t-1}) + (1 - \beta)b_{t-1}$	the <i>trend</i> sequence
$S_t = \gamma(y_t - L_t) + (1 - \gamma)S_{t-s}$	the <i>seasonal</i> sequence
$F_{t+k} = L_t + kb_t + S_{t+k-s}$	the <i>forecast</i> sequence

Holt-Winters Multiplicative Model

$L_t = \alpha(y_t / S_{t-s}) + (1 - \alpha)(L_{t-1} + b_{t-1})$	the <i>level</i> sequence
$b_t = \beta(L_t - L_{t-1}) + (1 - \beta)b_{t-1}$	the <i>trend</i> sequence
$S_t = \gamma(y_t / L_t) + (1 - \gamma)S_{t-s}$	the <i>seasonal</i> sequence
$F_{t+k} = (L_t + kb_t)S_{t+k-s}$	the <i>forecast</i> sequence

Note that without a seasonal component, both the additive and multiplicative formulations reduce to the same methods. (The seasonal sequence is set to 1 for the multiplicative model, and identically 0 for the additive model.)

Default Starting Values

Initial values are required for these sequences. The software allows the user code to define these initial values (see optional argument `IMSL_S_INIT_INPUT`). If they are not provided, then the first two seasons of data are used:

$L_s = \frac{1}{s} \sum_{i=1}^s y_i, s > 1$	
$b_s = \frac{1}{s} \sum_{i=1}^s \frac{(y_{i+s} - y_i)}{s}, s > 1$	
$S_i = y_i / L_s$ or $y_i - L_s, i = 1, \dots, s$	(Multiplicative or Additive)

The smoothing parameters (α, β, γ) are each in the interval [0,1] and can be specified by the user (see optional argument `IMSL_S_USE_PARAMS`), or automatically set by minimizing the within sample one-step ahead mean squared forecast error. Note that this objective function is not necessarily convex and solutions will depend on starting values. See Chatfield and Yar (1988) for further discussion. Starting values for (α, β, γ) are obtained by minimizing the mean squared error over `nsamples` bootstrap samples of the time series. Experiments suggest that this approach helps prevent poor starting values. Note, that solutions may vary for different settings of the number of random samples, `nsamples`.

The return value of the `imsls_hw_time_series` is the array of length $3*(N + 1) = 3*(nobs + 1)$ containing the smoothing parameter values on the first row, followed by the calculated level, trend, and seasonal sequences. When $N = nobs$ and $s = nseason$, the format of the return value is as follows:

Series Storage			
Row	Value		
0	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}$
1	0	0	S_1
2	0	0	S_2
⋮	⋮	⋮	⋮
s	L_s	b_s	S_s
$s+1$	L_{s+1}	b_{s+1}	S_{s+1}
⋮	⋮	⋮	⋮
N	L_N	b_N	S_N

If one of the nonseasonal options is specified, the return value will be of length $2*(\text{noobs} + 1)$ or $(\text{noobs} + 1)$ accordingly.

The variance-covariance matrix for the parameters (α, β, γ) is

$$\text{cov} = \frac{\sum e_i^2}{N - 2s - 3} (J^T J)^{-1}$$

where e_i is the one-step-ahead forecast error, and J is the Jacobian matrix of the series using the forecast model and calculating forecasts one step ahead of each datum. Prediction intervals are calculated following Chatfield and Yar (1991).

Example

A series of `noobs = 12` seasonal data values are analysed using the Multiplicative and the Additive Holt-Winters method. The season size is `nseason = 4`. The objective is to predict one season ahead, `nforecasts = 4` using each method. The forecasts and prediction interval lower and upper bounds are returned. The mean sum of squares of the one-step ahead forecast errors is shown to be smallest using the Multiplicative model.

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

#define NOBS 12
#define NSEASON 4

int main()
{
```

```

float y[NOBS]={23,25,36,31,26,28,48,36,31,42,53,43};
float *seriesm,*seriesa, *forecasts, *ysmoothed;
float ss,confidence;
int nforecasts=NSEASON,nvars=3;

/* Compute the time series and forecasts
using the Multiplicative model. */
confidence=95.0;

seriesm=imsls_f_hw_time_series(NOBS,NSEASON,y,
    IMSLS_FORECAST_CONFIDENCE,confidence,
    IMSLS_FORECASTS,nforecasts,&forecasts,
    IMSLS_SERIES_SMOOTHED, &ysmoothed,
    IMSLS_SS_RESIDUAL,&ss,
    0);

imsls_f_write_matrix(" Input time series ", 1,NOBS,y,0);
imsls_f_write_matrix(" Smoothed Multiplicative series ",
    1,NOBS, ysmoothed,0);
imsls_f_write_matrix(" Parameters and internal sequences ",
    NOBS+1,nvars,seriesm,0);
imsls_f_write_matrix(" Multiplicative forecasts\nwith"
    " 95% prediction interval",
    nforecasts,3,forecasts,0);
printf("MSS - Multiplicative %3.2f \n",
    ss/(float)(NOBS-NSEASON));

imsls_free(seriesm);
imsls_free(forecasts);
imsls_free(ysmoothed);

/* Compute the time series and forecasts
using the Additive model. */
seriesa=imsls_f_hw_time_series(NOBS, NSEASON, y,
    IMSLS_FORECASTS,nforecasts,&forecasts,
    IMSLS_SS_RESIDUAL, &ss,
    IMSLS_ADDITIVE, 1,
    IMSLS_FORECAST_CONFIDENCE,confidence,
    IMSLS_SERIES_SMOOTHED,&ysmoothed,
    0);

imsls_f_write_matrix(" Smoothed Additive series ",
    1,NOBS, ysmoothed,0);
imsls_f_write_matrix(" Parameters and internal sequences ",
    NOBS+1,nvars,seriesa,0);
imsls_f_write_matrix(" Additive forecasts\nwith"
    " 95% prediction interval",
    nforecasts,3, forecasts,0);
printf("MSS - Additive %3.2f \n",
    ss/(float)(NOBS-NSEASON));
}
#undef NOBS
#undef NSEASON

```

Output

```
Input time series
```

1	2	3	4	5	6
23	25	36	31	26	28
7	8	9	10	11	12
48	36	31	42	53	43
Smoothed Multiplicative series					
1	2	3	4	5	6
23.00	25.00	36.00	31.00	24.15	27.65
7	8	9	10	11	12
41.77	38.04	30.44	33.72	54.51	45.25
Parameters and internal sequences					
	1	2	3		
1	0.04	1.00	0.44		
2	0.00	0.00	0.80		
3	0.00	0.00	0.87		
4	0.00	0.00	1.25		
5	28.75	1.44	1.08		
6	30.28	1.53	0.83		
7	31.82	1.54	0.87		
8	33.55	1.73	1.33		
9	35.20	1.66	1.05		
10	36.89	1.68	0.83		
11	38.93	2.04	0.96		
12	40.93	2.00	1.31		
13	42.85	1.92	1.03		
Multiplicative forecasts with 95% prediction interval					
	1	2	3		
1	37.26	27.54	46.97		
2	44.99	35.24	54.74		
3	63.91	53.99	73.83		
4	52.14	42.17	62.11		
MSS - Multiplicative 15.35					
Smoothed Additive series					
1	2	3	4	5	6
23.00	25.00	36.00	31.00	23.00	26.32
7	8	9	10	11	12
38.58	38.54	34.05	35.73	56.63	43.83
Parameters and internal sequences					
	1	2	3		
1	0.27	0.64	1.00		
2	0.00	0.00	-5.75		
3	0.00	0.00	-3.75		
4	0.00	0.00	7.25		
5	28.75	0.00	2.25		
6	29.56	0.52	-3.56		
7	30.52	0.81	-2.52		
8	33.86	2.43	14.14		
9	35.61	1.99	0.39		
10	36.79	1.47	-5.79		
11	39.94	2.55	2.06		
12	41.51	1.92	11.49		
13	43.22	1.78	-0.22		

Additive forecasts			
with 95% prediction interval			
	1	2	3
1	39.21	27.80	50.62
2	48.84	36.37	61.31
3	60.05	45.74	74.35
4	50.13	33.24	67.01
MSS - Additive 21.18			

Warning Errors

IMSLS_HW_COV_NOT_EST

The covariance matrix could not be estimated. The parameter estimates may be at the upper or lower bound.

Fatal Errors

IMSLS_HW_SEASON_SIZE

The number for a Holt-Winters series season length must be ≥ 2 . Input value is $\#$.

IMSLS_HW_SERIES_SIZE

The number of observations for a Holt-Winters input series must be $\text{"nobs"} > 2 * \text{"nseason"} + 3$. Now have $\text{"nobs"} = \#$ and $\text{"nseason"} = \#$.

IMSLS_HW_SERIES_SIGN

The values of the input series, using the Holt-Winters Multiplicative method, must be positive. The series entry with index = $\#$ now has the non-positive value $\#$.

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 $(p+q+1) \times (p+q+1)$ 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,

$$0 < \sigma^2 < \max_sigma, \beta_i \geq 0, \alpha_i \geq 0 \text{ and}$$

$$\sum_{i=2}^{p+q+1} x(i) = \sum_{i=1}^p \beta_i + \sum_{i=1}^q \alpha_i < 1.$$

The above model is denoted as GARCH(p, q). The β_i and α_i coefficients will be referred to as GARCH and ARCH coefficients, respectively. When $\beta_i = 0, i = 1, 2, \dots, p$, 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 is

$$\log(L) = -\frac{m}{2} \log(2\pi) - \frac{1}{2} \sum_{t=1}^m y_t^2 / \sigma_t^2 - \frac{1}{2} \sum_{t=1}^m \log \sigma_t^2,$$

$$\text{where } \sigma_t^2 = \sigma^2 + \sum_{i=1}^p \beta_i \sigma_{t-i}^2 + \sum_{i=1}^q \alpha_i w_{t-i}^2.$$

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 \mathbf{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 be 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 function `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 <stdio.h>
#include <math.h>

void sgarch (int p, int q, int m, float x[],
             float y[], float z[], float y0[], float sigma[]);

#define MAX(a, b) ((a)>(b)?(a):(b))
#define M      1000
#define N      (P + Q + 1)
#define P      2
#define Q      1

int main ()
{
    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);

    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);
}

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 = MAX (p, q);
    l = 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 (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 = 1; 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];  
}
```

Output

```
Sigma estimate is      1.7629  
ARCH(1) estimate is   0.2517  
GARCH(1) estimate is  0.3340  
GARCH(2) estimate is  0.3034  
  
Log-likelihood function value is -2707.0886  
Akaike Information Criterion value is 5422.1772
```

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 $\text{covb} \times \sigma^2$ is the mean squared error matrix for **b**. Before the first call to `imsls_f_kalman`, $\text{covb} \times \sigma^2$ 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. The estimate of σ^2 is given by $\frac{ss}{n}$. **ss** must be initialized to zero before the first call to `imsls_f_kalman` is made.

*float *alndet* (Input/Output)

Pointer to the natural log of the product of the nonzero eigenvalues of P where $P \times \sigma^2$ 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>

void imsls_f_kalman(int nb, float b[], 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 q_col_dim,
    IMSLS_TOLERANCE, float tolerance,
    IMSLS_V, float **v,
    IMSLS_V_USER, float v[],
    IMSLS_COVV, float **covv,
    IMSLS_COVV_USER, float covv[],
    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.

\mathbf{r} : \mathbf{ny} by \mathbf{ny} array containing the matrix such that $\mathbf{r} \times \sigma^2$ 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 \mathbf{r} are referenced.

IMSLS_Z_COL_DIM, *int* $\mathbf{z_col_dim}$ (Input)

Column dimension of the matrix \mathbf{z} .

Default: $\mathbf{z_col_dim} = \mathbf{nb}$

IMSLS_R_COL_DIM, *int* $\mathbf{r_col_dim}$ (Input)

Column dimension of the matrix \mathbf{r} .

Default: $\mathbf{r_col_dim} = \mathbf{ny}$

IMSLS_T, *float* * \mathbf{t} (Input)

\mathbf{nb} by \mathbf{nb} transition matrix in the state equation

Default: $\mathbf{t} =$ identity matrix

IMSLS_T_COL_DIM, *int* $\mathbf{t_col_dim}$ (Input)

Column dimension of the matrix \mathbf{t} .

Default: $\mathbf{t_col_dim} = \mathbf{nb}$

IMSLS_Q, *float* * \mathbf{q} (Input)

\mathbf{nb} by \mathbf{nb} matrix such that $\mathbf{q} \times \sigma^2$ 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* $\mathbf{q_col_dim}$ (Input)

Column dimension of the matrix \mathbf{q} .

Default: $\mathbf{q_col_dim} = \mathbf{nb}$

IMSLS_TOLERANCE, *float* $\mathbf{tolerance}$ (Input)

Tolerance used in determining linear dependence.

Default: $\mathbf{tolerance} = 100.0 \times \mathbf{imsls_f_machine}(4)$

IMSLS_V, *float* ** \mathbf{v} (Output)

Address to a pointer \mathbf{v} to an array of length \mathbf{ny} containing the one-step-ahead prediction error.

IMSLS_V_USER, *float* $\mathbf{v} []$ (Output)

Storage for \mathbf{v} is provided by the user. See IMSLS_V.

IMSLS_COVV, *float* ** \mathbf{covv} (Output)

The address to a pointer of size \mathbf{ny} by \mathbf{ny} containing a matrix such that $\mathbf{covv} \times \sigma^2$ is the variance-covariance matrix of \mathbf{v} .

`IMSLS_COVV_USER`, *float covv* [] (Output)

Storage for `covv` is provided by the user. See `IMSLS_COVV`.

Description

Function `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 function `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 \mathbf{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 $k = 1, 2, \dots$ 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

$$y_k = Z_k b_k + e_k \quad k = 1, 2, \dots$$

Here, Z_k (input in \mathbf{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*

$$b_{k+1} = T_{k+1} b_k + w_{k+1} \quad k = 1, 2, \dots$$

starting with $b_1 = \boldsymbol{\mu}_1 + w_1$.

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_k s ($k = 1, 2, \dots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\boldsymbol{\sigma}^2 Q_k$, that the n_k -dimensional e_k s ($k = 1, 2, \dots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\boldsymbol{\sigma}^2 R_k$, and that the w_k s and e_k s are independent of each other. Here, $\boldsymbol{\mu}_1$ is the mean of b_1 and is assumed known, $\boldsymbol{\sigma}^2$ is an unknown positive scalar. Q_{k+1} (input in \mathbf{q}) and R_k (input in \mathbf{r}) are assumed known.

Denote the estimator of the realization of the state vector b_k given the observations y_1, y_2, \dots, 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 \mathbf{b} and \mathbf{covb} , respectively. During the k -th invocation, function `ims1s_f_kalman` computes the filtered estimate

$$\hat{\beta}_{k|k}$$

along with $C_{k|k}$. These quantities are given by the *update equations*:

$$\begin{aligned}\hat{\beta}_{k|k} &= \hat{\beta}_{k|k-1} + C_{k|k-1} Z_k^T H_k^{-1} v_k \\ C_{k|k} &= C_{k|k-1} - C_{k|k-1} Z_k^T H_k^{-1} Z_k C_{k|k-1}\end{aligned}$$

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 \mathbf{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 \mathbf{covv} . The “start-up values” needed on the first invocation of `ims1s_f_kalman` are

$$\hat{\beta}_{1|0} = \mu_1$$

and $C_{1|0} = Q_1$ input via \mathbf{b} and \mathbf{covb} , respectively. Computations for the k -th invocation are completed by `ims1s_f_kalman` computing the one-step-ahead estimate

$$\hat{\beta}_{k+1|k}$$

along with $C_{k+1|k}$ given by the *prediction equations*:

$$\begin{aligned}\hat{\beta}_{k+1|k} &= T_{k+1}\hat{\beta}_{k|k} \\ C_{k+1|k} &= T_{k+1}C_{k|k}T_{k+1}^T + Q_{k+1}\end{aligned}$$

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 `IMSLT_T` and `IMSLT_Q` to produce

$$\hat{\beta}_{k|k}$$

and $C_{k|k}$, and second without `IMSLT_UPDATE` to produce

$$\hat{\beta}_{k+1|k}$$

and $C_{k+1|k}$. (Without `IMSLT_T` and `IMSLT_Q`, the prediction equations are skipped. Without `IMSLT_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 `IMSLT_UPDATE` to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of `imsls_f_kalman` without `IMSLT_UPDATE` can compute

$$\hat{\beta}_{j+2|j}, \hat{\beta}_{j+3|j}, \dots, \hat{\beta}_{k|j}$$

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, \dots, 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 \mathbf{n} and \mathbf{ss} .

If σ^2 is known, the R_k s and Q_k s 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 function `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, \dots, y_m differs by no more than an additive constant from

$$L(\theta, \sigma^2; y_1, y_2, \dots, y_m) = -\frac{1}{2}N \ln \sigma^2 - \frac{1}{2} \sum_{k=1}^m \ln[\det(H_k)] - \frac{1}{2} \sigma^{-2} \sum_{k=1}^m v_k^T H_k^{-1} v_k$$

(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 $-2L(\theta, \sigma^2; y_1, y_2, \dots, y_m)$ over all θ and σ^2 produces maximum likelihood estimates. Equivalently, minimization of $-2L_c(\theta; y_1, y_2, \dots, y_m)$ where

$$L_c(\theta; y_1, y_2, \dots, y_m) = -\frac{1}{2}N \ln\left(\frac{SS}{N}\right) - \frac{1}{2} \sum_{k=1}^m \ln[\det(H_k)]$$

produces maximum likelihood estimates

$$\hat{\theta} \text{ and } \hat{\sigma}^2 = SS/N$$

The minimization of $-2L_c(\theta; y_1, y_2, \dots, y_m)$ instead of $-2L(\theta, \sigma^2; y_1, y_2, \dots, y_m)$, reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

$$\hat{\sigma}^2(\theta) = SS(\theta) / N$$

minimizes $-2L(\theta, \sigma^2; y_1, y_2, \dots, y_m)$ for all θ , consequently,

$$\hat{\sigma}^2(\theta)$$

can be substituted for σ^2 in $L(\theta, \sigma^2; y_1, y_2, \dots, y_m)$ to give a function that differs by no more than an additive constant from $L_c(\theta; y_1, y_2, \dots, y_m)$.

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:

- Replace H_k^{-1} by a generalized inverse.
- Replace $\det(H_k)$ by the product of the nonzero eigenvalues of H_k .
- Replace N by $\sum_{k=1}^m \text{rank}(H_k)$

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

$$\begin{aligned} y_k &= b_k + e_k \\ b_{k+1} &= b_k + w_{k+1} \quad k = 1, 2, 3, 4 \end{aligned}$$

where the e_k s are identically and independently distributed normal with mean 0 and variance σ^2 , the w_k s 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 `IMSLT` and `IMSLQ` 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

int 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 function `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 $y_k = \epsilon_k - \theta\epsilon_{k-1}$. Function `imsls_f_random_arma` (see 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_k s are independently distributed bivariate normal with mean 0 and variance $\sigma^2 Q_k$ and

$$Q_1 = \begin{pmatrix} 1 + \theta^2 & -\theta \\ -\theta & \theta^2 \end{pmatrix}$$

$$Q_k = \begin{pmatrix} 1 & -\theta \\ -\theta & \theta^2 \end{pmatrix} \quad k = 2, 3, \dots, 200$$

The warning error that is printed as part of the output is not serious and indicates that `imsl_f_min_uncon_multivar` (see C/Math/Library, Chapter 8, *Optimization*) is generally used for multi-parameter minimization.

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

#define NOBS 200
#define NTHETA 1
#define NB 2
#define NY 1

float fcn(int ntheta, float theta[]);
float *ydata;

int 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
```

vector_autoregression

Estimates a vector auto-regressive time series model with optional moving average components.

Synopsis

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

```
float *imsls_f_vector_autoregression(int n_obs, int n_cols, float y[], int p, ..., 0)
```

The type *double* function is `imsls_d_vector_autoregression`.

Required Arguments

int `n_obs` (Input)

The number of rows in `y`. `n_obs` is equal to the number of observations of each time series.

int `n_cols` (Input)

The number of columns in `y`. `n_cols = K`, the number of individual time series.

float `y[]` (Input)

An array of size `n_obs` by `n_cols` containing the data.

int `p` (Input)

The autoregressive lag order.

Return Value

An array containing the estimated coefficients. The array has length $n_cols \times (\text{trend} + n_xvars + n_cols \times (p + q + 1))$ if a non-trivial A_0 is included in the model. Otherwise, the array has length $n_cols \times (\text{trend} + n_xvars + n_cols \times (p + q))$. `trend = 1` if `IMSLS_TREND` is specified, and 0 otherwise.

The array elements occur in this order:

- If `IMSLS_TREND` is specified, the trend coefficient vector (b_0) of length K ($K = n_cols$).
- The coefficient matrix for the deterministic variables (D), of size K by `n_xvars`, oriented by column.
- The `p` (or `p+1`) autoregressive coefficient matrices (A), each of size K by K , oriented by column.

- The q moving average coefficient matrices (M), each of size K by K , oriented by column.

This arrangement corresponds to the vectorized coefficient matrix,

$$B = \text{vec}[b_0, D, A_0, A_1, \dots, A_p, M_1, M_2, \dots, M_q]$$

or

$$B = \text{vec}[b_0, D, A_1, \dots, A_p, M_1, M_2, \dots, M_q]$$

See also optional argument [IMSLS_RETURN_USER](#) for a different form of these output values. To release this space, use [imsls_free](#). If no value can be computed, or [IMSLS_RETURN_USER](#) is supplied, returns `NULL`.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_vector_autoregression(int n_obs, int n_cols, float y[], int p,
    IMSLS_MA_LAG, int q,
    IMSLS_A0,
    IMSLS_AR_MODEL, int ar[],
    IMSLS_MA_MODEL, int ma[],
    IMSLS_AR_CONSTANTS, float ar_c[],
    IMSLS_MA_CONSTANTS, float ma_c[],
    IMSLS_PRESAMPLE, int n_T,
    IMSLS_MAX_LAG, int max_lag,
    IMSLS_N_STEPS, int max_steps,
    IMSLS_MAX_ITERATIONS, int max_iter,
    IMSLS_TOLERANCE, float tol,
    IMSLS_TREND,
    IMSLS_SCALE,
    IMSLS_CENTER,
    IMSLS_X_DATA, int n_xvars, float x[],
    IMSLS_ERROR_CORRECTION, int irank,
    IMSLS_CAUSALITY, int size_S1, int S1[], int S2[],
    IMSLS_CAUSALITY_STATS, float stats[],
    IMSLS_VAR_INFO, lmsls_f_regression **var_info,
    IMSLS_VARMA_INFO, lmsls_f_regression **varma_info,
    IMSLS_UNIT_ROOT, f_complex **ur_evals,
    IMSLS_UNIT_ROOT_USER, f_complex ur_evals[],
    IMSLS_VECM_COEF, float **vecm_coef,
    IMSLS_VECM_COEF_USER, float vecm_coef[],
    IMSLS_VECM_EIGENVALUES, f_complex **vecm_eigens,
```

```

IMSLS_VECM_EIGENVALUES_USER, f_complex vecm_eigen[ ],
IMSLS_VECM_ALPHABETA, float ** vecm_alphabeta,
IMSLS_VECM_ALPHABETA_USER, float vecm_alphabeta[ ],
IMSLS_FORECASTS, float ** forecasts,
IMSLS_FORECASTS_USER, float forecasts[ ],
IMSLS_CRITERIA, float criteria[ ],
IMSLS_LOG_LIKELIHOOD, float *ll,
IMSLS_RETURN_USER, float b0[ ], float d[ ], float a[ ], float m[ ],
0)

```

Optional Arguments

IMSLS_MA_LAG, *int* **q** (Input)

Fit a moving average component of order **q**.

Default: **q** = 0.

IMSLS_A0, (Input)

Indicates that the model has a non-trivial, lower-triangular leading AR operator, A_0 . See the [Description](#) section for more details. By default, $A_0 = I_K$, where $K = \mathbf{n_cols}$ and I_K is the K by K identity matrix.

IMSLS_AR_MODEL, *int* **ar**[] (Input)

An array used to specify restrictions on the AR coefficient matrices. If **IMSLS_A0** is present, **ar** is of length K by K by $(\mathbf{p}+1)$. If **IMSLS_A0** is not present, **ar** is K by K by \mathbf{p} . Each element of **ar** should be one of $\{-1, 0, 1\}$. If **IMSLS_A0** is present, indicating a non-trivial A_0 , the ordering of AR corresponds to the parameters as follows:

$$\begin{aligned}
 & [\alpha_{11,0}, \alpha_{21,0}, \dots, \alpha_{K1,0}, \dots, \alpha_{1K,0}, \alpha_{2K,0}, \dots, \alpha_{KK,0}, \\
 & \alpha_{11,1}, \alpha_{21,1}, \dots, \alpha_{K1,1}, \dots, \alpha_{1K,1}, \alpha_{2K,1}, \dots, \alpha_{KK,1}, \dots, \\
 & \alpha_{1K,p}, \alpha_{2K,p}, \dots, \alpha_{KK,p}]
 \end{aligned}$$

If A_0 is trivial (equal to the identity matrix), the ordering is the same but the elements $\alpha_{ki,0}$ are left out and the first element is $\alpha_{11,1}$. If **ar**[**i**] = 1 or **ar**[**i**] = -1, $A(k,i) = \alpha_{ki,j}$ is estimated. If **ar**[**i**] = 0, the parameter is restricted to be equal to 0 or some other constant. Constants other than 0 can be specified in the optional argument **IMSLS_AR_CONSTANTS**. By default, all parameters are estimated.

Default: **ar**[**i**] = 1.

IMSLS_MA_MODEL, *int* **ma**[] (Input)

An array of length K by K by $(\mathbf{q}+1)$ if **IMSLS_A0**, or K by K by \mathbf{q} if not **IMSLS_A0**. **ma** is used to specify restrictions on the moving-average coefficients. **ma** is constructed analogously to the **ar**

restriction matrix detailed above. If `IMSLS_A0`, the first K by K by 1 entries in `ma` are only used when `ar` is not provided, because it is assumed that $M_0 = A_0$.

Default: `ma[i] = 1`.

`IMSLS_AR_CONSTANTS`, *float* `ar_c[]` (Input)

An array of length K by $(p+1)$ (or K by p) specifying the constants when certain of the AR parameters are restricted. Note that a lower triangular A_0 is always assumed, with 1's on the diagonal.

However, constants for the lower triangle can be specified using this optional argument.

Default `ar_c[i] = 0`.

`IMSLS_MA_CONSTANTS`, *float* `ma_c[]` (Input)

An array of length K by $(q+1)$ (or K by q) specifying the constants when certain of the MA parameters are restricted. Note that the leading MA coefficient matrix $M_0 = A_0$ and thus there is no specification for it.

Default `ma_c[i] = 0`.

`IMSLS_PRESAMPLE`, *int* `n_T` (Input).

Specifies the number of rows of `y` to use as a presample in the estimation procedure. `n_T` must be strictly greater than 0.

Default `n_T = max_lag`.

`IMSLS_MAX_LAG`, *int* `max_lag` (Input)

Specifies the maximum AR order or lag to use in the unrestricted VAR model. `max_lag` must be strictly greater than 0.

Default: `max_lag = 6`.

`IMSLS_N_STEPS`, *int* `max_steps` (Input)

Specifies the maximum number of steps ahead. When requested, forecasts are produced for times $t+h$ where $h = 1, 2, \dots, \text{max_steps}$ and $t = n_T, n_T+1, \dots, n_{\text{obs}}$.

Default: `max_steps = 4`.

`IMSLS_MAX_ITERATIONS`, *int* `max_iter` (Input)

Specifies the maximum number of iterations. `max_iter > 0`.

Default: `max_iter = 100`.

`IMSLS_TOLERANCE`, *float* `tol` (Input)

Specifies the error tolerance used in the iterations.

Default: `tol = 100 \times \text{imsls_f_machine}(4)`.

`IMSLS_TREND`, (Input)

Indicates that a constant trend (intercept) term should be included in the model.

Default: No trend.

IMSLS_SCALE, (Input)

Indicates that the data series in \mathbf{y} should be centered and then scaled before the analysis begins.
Default: No scaling.

IMSLS_CENTER, (Input)

Indicates that the data series in \mathbf{y} should be centered before the analysis begins. Note that if **IMSLS_SCALE** is provided, **IMSLS_CENTER** has no effect.
Default: No centering.

IMSLS_X_DATA, *int n_xvars*, *float x[]* (Input)

Input values of the deterministic variables.

int n_xvars (Input)

The number of deterministic variables.

float x[] (Input)

The $\mathbf{n_obs}$ by $\mathbf{n_xvars}$ array containing the values of the deterministic variables. Note that if \mathbf{x} is provided when forecasts are requested, \mathbf{x} must have additional rows that can be used for the forecasts. That is, \mathbf{x} is an array of at least $(\mathbf{n_obs} + \mathbf{max_steps})$ rows by $\mathbf{n_xvars}$.
Default: $\mathbf{n_xvars} = 0$.

IMSLS_ERROR_CORRECTION, *int irank* (Input)

Estimate the error-correcting form of the VAR model assuming the series are integrated of order 1 and there are $0 \leq \mathbf{irank} \leq \mathbf{n_cols}$ co-integrating relationships. When $\mathbf{irank}=0$, the error-correcting model is not estimated.
Default: $\mathbf{irank} = 0$.

IMSLS_CAUSALITY, *int size_S1*, *int S1[]*, *int S2[]* (Input)

Specification for a test of Granger causality.

int size_S1 (Input)

The number of variables with indices given in array $\mathbf{S1}$.
 $0 \leq \mathbf{size_S1} < \mathbf{n_cols}$. No test is performed when $\mathbf{size_S1} = 0$.
Default: $\mathbf{size_S1} = 0$.

float S1[] (Input)

An array of length $\mathbf{size_S1}$ containing the indices of the variables that are not Granger-caused by the variables specified in $\mathbf{S2}$, under the null hypothesis. Each index must be between 1 and $\mathbf{n_cols}$ inclusive and must not equal any index in $\mathbf{S2}$.

float **S2[]** (Input)

An array of length `n_cols-size_S1` containing the indices of the variables that do not Granger-cause the variables specified in **S1**, under the null hypothesis. Each index must be between 1 and `n_cols` inclusive and must not equal any index in **S1**.

IMSLS_CAUSALITY_STATS, *float* **stats[]** (Output)

An array of length 2 containing the test statistic value and associated p-value resulting from the requested Granger causality test. It is an error to request this output without specifying the test using **IMSLS_CAUSALITY**.

IMSLS_VAR_INFO, *imsls_f_regression **var_info* (Output)

Contains the regression information from the first stage fitting of the model, VAR(`max_lag`). This structure may be used as input to [imsls_f_regression_summary](#). See also [imsls_f_regression](#) optional argument **IMSLS_REGRESSION_INFO**.

IMSLS_VARMA_INFO, *imsls_f_regression **varma_info* (Output)

Contains the regression information from the second stage fitting of the model. VARMA(`p`, `q`). This structure may be used as input to [imsls_f_regression_summary](#). See also [imsls_f_regression](#) optional argument **IMSLS_REGRESSION_INFO**.

IMSLS_UNIT_ROOT, *f_complex **ur_evals* (Output)

Address of a pointer to a complex array of length `K` by `(p+q)` containing the eigenvalues of the determinantal polynomial.

IMSLS_UNIT_ROOT_USER, *f_complex ur_evals[]* (Output)

Storage for the array `ur_evals` is provided by the user.

IMSLS_VECM_COEF, *float **vecm_coef* (Output)

Address of a pointer to an array of length `K` by `K` by `(p+1)` or `(K` by `K` by `p`) containing the estimated parameters when the model is put into the error-correcting form.

IMSLS_VECM_COEF_USER, *float vecm_coef[]* (Output)

Storage for the array `vecm_coef` is provided by the user.

IMSLS_VECM_EIGENVALUES, *f_complex **vecm_eigens* (Output)

Address of a pointer to an array of length `K` containing the eigenvalues associated with each cointegrating rank `r`, $0 \leq r \leq K$.

IMSLS_VECM_EIGENVALUES_USER, *f_complex vecm_eigens[]* (Output)

Storage for the array `vecm_eigens` is provided by the user.

IMSLS_VECM_ALPHABETA, *float **vecm_alphabeta* (Output)

An array of length $2 \times (K \text{ by } \text{irank})$ containing the estimates of the coefficient matrix $\Pi = \alpha\beta'$ of the error-correction model. The first K by irank elements correspond to α and the second irank by K elements correspond to β .

IMSLS_VECM_ALPHABETA_USER, *float vecm_alphabeta[]* (Output)

Storage for the array **vecm_alphabeta** is provided by the user.

IMSLS_FORECASTS, *float **forecasts* (Output)

An array of length n_obs by n_cols by max_steps containing the 1, 2, ..., max_steps ahead forecasts based on the final fitted VARMA model.

IMSLS_FORECASTS_USER, *float forecasts[]* (Output)

Storage for the array **forecasts** is provided by the user.

IMSLS_CRITERA, *float criteria[]* (Output)

An array of size 4 containing AIC, BIC, HQ, and FPE fit criteria for the given model.

IMSLS_LOG_LIKELIHOOD, *float *ll* (Output)

Log-likelihood of the estimated VARMA(p,q) model.

IMSLS_RETURN_USER, *float b0[], float d[], float a[], float m[]* (Output)

Storage for the return value is supplied by the user in separate arrays.

float b0[] (Output)

If **IMSLS_TREND** is specified, the estimated constant trend coefficient of length K ($K = \text{n_cols}$) is returned in **b0**. If **IMSLS_TREND** is not specified, **b0** is ignored and can be **NULL**.

float d[] (Output)

If $\text{n_xvars} > 0$, the estimated K by n_xvars coefficient matrix for the deterministic variables is returned in **d**. If $\text{n_xvars} = 0$, **d** is ignored and can be **NULL**.

float a[] (Output)

If **IMSLS_A0** is specified, the estimated $(\text{p}+1)$ autoregressive matrices, each of length K by K , are returned in **a**. If **IMSLS_A0** is not specified, the (p) autoregressive matrices, each of length K by K , are returned in **a**. If $\text{p} = 0$, **a** is ignored and can be **NULL**. Note that if $\text{p} = 0$, q must be greater than 0 (and vice-versa).

float m[] (Output)

The estimated (q) moving average coefficient matrices, each of length K by K , are returned in **m**. If $\text{q} = 0$, **m** is ignored and can be **NULL**. Note that if $\text{q} = 0$, p must be greater than 0 (and vice-versa).

Description

This function estimates a vector-autoregression moving average model using one and two-stage multivariate least squares regressions.

The general model can be written in operator notation as

$$A(L)(y_t - \boldsymbol{\mu}) = M(L)u_t + Dx_t$$

where y_t is a K -dimensional real-valued time series with stationary mean $\boldsymbol{\mu}$, u_t is a K -dimensional white noise series with a non-singular covariance matrix, $\boldsymbol{\Sigma}_u$, and x_t represents a matrix of deterministic components such as trend or seasonal variables. For autoregression (**ar**) lag parameter, $p \geq 0$, and moving average (**ma**) lag parameter, $q \geq 0$, the operators are defined as

$$A(L) = A_0 - A_1L - A_2L^2 - \dots - A_pL^p$$

$$M(L) = M_0 - M_1L - M_2L^2 - \dots - M_qL^q$$

where L is the lag or backshift operator, defined as

$$Ly_t = y_{t-1}, L^k y_t = y_{t-k}$$

and the A_j, M_j are $K \times K$ matrices of coefficients. That is,

$$A_j(k, i) = [\alpha_{kij}], k, i = 1, \dots, K, j = 0, \dots, p$$

$$M_j(k, i) = [m_{kij}], k, i = 1, \dots, K, j = 0, \dots, q$$

The model has many equivalent forms, such as

$$A_0(y_t - \boldsymbol{\mu}) = A_1(y_{t-1} - \boldsymbol{\mu}) + \dots + A_p(y_{t-p} - \boldsymbol{\mu}) + A_0u_t + M_1u_{t-1} + \dots + M_qu_{t-q}$$

A pure vector autoregression with $q = 0$ is often denoted $VAR(p)$ and the autoregressive moving-average is $VARMA(p, q)$.

The estimation procedures for the most part assume that the underlying time series is stable and invertible. These conditions are satisfied when

$$\det(I_K - A_1z - \dots - A_pz^p) \neq 0 \text{ for } |z| \leq 1 \text{ (stability condition)}$$

$$\det(I_K + M_1z + \dots + M_qz^q) \neq 0 \text{ for } |z| \leq 1 \text{ (invertibility condition)}.$$

In other words, the conditions for a stable and invertible process are that the roots of the determinantal polynomial are outside the (complex) unit circle, or equivalently, that the eigenvalues of the determinantal matrix have modulus less than 1. To provide for unit-root tests, this function returns the eigenvalues of the estimated model via the optional argument **IMSLS_UNIT_ROOT**. In the case of unit-roots, when some fall on the unit circle ($|z| = 1$) there may still be stationary combinations of the variables. This behavior, known as *cointegration*, is discussed below.

Error-Correction form and Cointegration

Another form of expressing the $VARMA(p,q)$ model is the *error-correcting* form:

$$A_0 \Delta y_t = \Pi y_{t-1} + \Gamma(L) \Delta y_t + M(L) u_t$$

where

$$\Pi = -(A_0 - A_1 - \dots - A_p),$$

$$\Gamma(L) = (\Gamma_1 L + \Gamma_2 L^2 + \dots + \Gamma_p L^p), \text{ and}$$

$$M(L) = (M_0 + M_1 L + \dots + M_q L^q).$$

The error-correcting or error-correction model (ECM) was developed for economic variables that tend to track close to each other due to common trends. The concept is closely aligned with that of cointegration. A K -dimensional process is said to be integrated of order 1, denoted as $y_t \sim I(1)$, when y_t is non-stationary while the first-differenced series, Δy_t , is stationary. In this case, there may exist a stationary linear combination in the levels (non-differenced versions). That is, there may exist a

$$z_t = \beta' y_t = \beta_{1t} y_{1t} + \beta_{2t} y_{2t} + \dots + \beta_{Kt} y_{Kt},$$

where at least one $\beta_{it} \neq 0$ such that z_t is stationary.

The interpretation is that cointegrated variables have common trends that cause them to move together in some sense. Consider the price of a commodity in two different locations. At their individual levels, they are non-stationary, unpredictable processes, but market forces keep the two prices from being too far apart so the difference in their prices is stationary.

We obtain an estimate of each coefficient matrix in the ECM form of the model. Then we are interested in the decomposition of $\Pi = \alpha\beta'$ where α, β are $K \times \text{irank}$ (rank irank coefficient matrices). From the ECM form it can be deduced that $z_t = \beta' y_t$ is the cointegrating relationship under the given assumptions. However, the cointegrating rank r is not known in practice. As discussed in [Lütkepohl, 2007, Ch. 7](#), the most common tests for the correct cointegrating rank are

$$H_0 : r = r_0 \text{ vs } H_1 : r_0 < r \leq K$$

and

$$H_0 : r = r_0 \text{ vs } H_1 : r = r_0 + 1$$

The likelihood ratio test statistic for testing the first hypothesis is known as the *trace* test, and the second is known as the maximum eigenvalue test ([Johansen 1988, 1995](#)). In particular,

$$\lambda(r_0, K) = -T \sum_{i=r_0+1}^K \ln(1 - \lambda_i) \quad \text{and}$$

$$\lambda(r_0, r_0 + 1) = -T \ln(1 - \lambda_{r_0+1})$$

where $\lambda_1 \geq \dots \geq \lambda_K$ are eigenvalues of a particular matrix associated with the likelihood function. To allow for various tests, this function returns the eigenvalues $\lambda_1 \geq \dots \geq \lambda_K$ through the optional argument `IMSLS_VECM_EIGENVALUES`.

The Granger-Causality Test

Partition the K -dimensional time series as $y_t' = (z_t, x_t)'$, where z_t is of length m and x_t is of length $K-m$. In general, x_t is said to *Granger-cause* z_t when predictions for z_t can be improved by taking x_t into account, and vice-versa. In the context of a stable and invertible $VAR(p)$ or $VARMA(p, q)$ process, a test for Granger causality amounts to testing for certain 0 restrictions on the AR or MA coefficients. Given a partition $y_t' = (z_t, x_t)'$, the test of

$$H_0 : x_t \text{ does not Granger-cause } z_t$$

$$H_1 : x_t \text{ does Granger-cause } z_t$$

is performed using a version of the Wald statistic which has an approximate F-distribution,

$$\lambda_W / N \approx F(N, T - K(p + q) - 1)$$

where N is the number of 0 restrictions induced by the test. This function returns the test statistic and associated p-value when a test is specified through the optional argument `IMSLS_CAUSALITY`.

Comments

1. There are different notational conventions in the literature. Box, Jenkins, and Reinsel typically use (Φ, Θ) in place of (A, M) and B instead of L for the backshift or lag operator. Because it is the main reference for this implementation, we follow the notation used by [Lütkepohl \(2007\)](#).
2. The two-stage regression approach is robust in the sense that results are produced even when there are roots close to 1 or less than 1 (in modulus). The estimates can be tested for unstable roots using the optional argument `IMSLS_UNIT_ROOT`.
3. In general a collection of K time series $y_t = (y_{1t}, y_{2t}, \dots, y_{kt})'$ is said to be cointegrated of order $(d-b)$ when they are each individually integrated of order d , but there exists a linear combination $z_t = \beta' y_t = \beta_{1t} y_{1t} + \beta_{2t} y_{2t} + \dots + \beta_{kt} y_{kt}$, with $\beta_{it} \neq 0$ for at least one i , which is integrated of order b . In this implementation, only the case where $d = 1$ and $b = 0$ is considered. An even more general situation allows the individual time series to have differing orders of integration. For more details, see, for example, [Engle and Granger, \(1991\)](#).

Examples

Example 1

In this example we use a small data set with 2 time series. A VARMA(1,0) model is requested.

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

int main(){
    int n_obs=20, n_cols=2;
    int p=1;
    float *coef=NULL;

    float y[]={0, 0,
        -0.148105526820657, 0.0507420782620461,
        -1.13674727366735, 0.862892721126079,
        1.54366183541037, -1.13804802266371,
        -0.0923211737957721, 1.65649055346038,
        0.521764564424907, -2.11208211206815,
        0.843683397890515, 2.56656798707681,
        -2.70330819114831, -2.83452914666041,
        4.93683704766295, 3.95965377457266,
        -4.78314880243807, -2.23136673998374,
        6.24911547448071, 0.40543051970714,
        -6.76582567372196, 0.816818897274206,
        6.21142944093012, -4.247694573354,
        -5.29817270483491, 5.08246614505046,
        4.19557583927549, -5.35697380907112,
        -3.21572784971078, 7.89716072829777,
        0.485717553966479, -8.25930665413043,
        2.69490292018773, 10.9017252520684,
        -5.41090143363388, -10.400477539227,
        8.29423327234419, 9.10321370415527};

    coef=imsls_f_vector_autoregression(n_obs, n_cols, y, p, 0);

    imsls_f_write_matrix("A1", 2, 2, coef,
        IMSLS_TRANSPOSE,
        0);

    if(coef){
        imsls_free(coef);
        coef = NULL;
    }
}
```

Output

	A1	
	1	2
1	-1.017	-0.296
2	0.273	-1.053

Example 2

In this example, we fit the unrestricted VARMA(1,1) model and request 1 and 2 step ahead forecasts for each observation.

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

int main(){
    int n_obs=50, n_cols=2, p=1, q=1, max_step=2;
    float forecasts[2*2*50];
    float *coef = NULL;

    char *clabel[] = {"t", "t+1, Y1", "t+1, Y2 ", "t+2, Y1", "t+2, Y2"};

    float y[] =
    {
        0.143280117400598, 0.691517079320758, -1.054002748442,
        1.89368500251305, -0.595435384283415, -1.53385197609914,
        -0.450087415747451, -0.522758296071267, 0.338719815948114,
        1.73890633300759, -0.0420130132503371, -0.00933890926865466,
        -0.939527210947825, -0.279211957506804, 0.985111509818102,
        -0.185903912738057, -0.779058630583496, -1.52829111438157,
        -0.0444517646926054, -0.605935345974391, 1.17870357395841,
        0.979822006409954, -1.75509211790604, -1.02617250037744,
        -2.64443752185661, -0.405444884498378, -0.558146570012133,
        1.17006139568333, -0.14877513906561, 1.67436399145195,
        1.21251151094695, -0.236411746432856, -0.319260279201159,
        1.53774676549506, -0.798919508505848, -1.25907348772775,
        -0.43488363747126, 1.18754392780486, 2.49394567166528,
        -0.505392075680617, -0.939902530453777, -1.3000118234638,
        0.308365204823071, -0.0346715133254558, 0.155821836363299,
        1.53865066350577, -0.446013548645569, 0.421382795732249,
        -0.810472750765929, -0.475790066827614, -1.21547965787564,
        0.873092852598299, 0.314687304446453, -0.166494291509063,
        1.20846773815425, -0.21319122737697, 0.885697825416125,
        1.06749862455588, -0.417811475765902, 1.81350258917296,
        1.06312903931625, -0.357098401483029, -2.54962241395723,
        1.58241298127273, -0.445333714405381, -1.54921054521057,
        0.763932954657703, -0.459132068443737, -0.135020632775658,
        0.768987806710127, -2.21000914520305, -0.416578967811937,
        -0.787803924930718, -0.381994679294779, 0.651886558080692,
        -0.296275077784937, -1.34100151116192, -0.695511844572305,
        -1.38111408367782, -0.453263483406621, 0.546836142779091,
        1.35617195687341, 1.47675672302528, -0.879275538764735,
        1.33724523036386, -1.19560541543992, -0.355724070126155,
        -1.35440645398376, 0.215557787562229, -0.0705280128055462,
        2.64174061572298, 0.622022977819918, 1.85193603585307,
        1.1425463214074, 1.47034707550646, -0.274392879035261,
        -0.0256438865664471, -1.32768501270967, 1.4002259906097,
        -1.13393234222336, 0.420685154939234, -1.14369662556446,
        -0.229525100642792, -1.14556440309372, -0.0726173507068031,
        0.537003424016424, 2.21081287812981, -0.329847605902733,
        0.658135565199115, 1.25161597712012, 0.324279854980608,
        -0.122777642651712, -0.746398274242844, -0.510384971856952,
        -1.52397549713935, -0.590932096161281, 0.351879848132452,
        -0.486815223667361, -0.582122931146207, -0.129708090826525,
        -1.83261132721672, -0.438855077878885, 0.160891671954672,
```

```

0.130903505342111, -1.0930038744212, 0.858667240391389,
-0.650140333935879, -0.192590383440759, -0.495902099869389,
-0.0970274914548782, 0.271597266914655, -1.25629606008009,
-1.82411099200514, -0.0862331652538604, -1.48736902275701,
-0.702589236231933, 1.66371111576656, 0.260453048198016,
0.472465295167692, 2.03959666287312, -1.47220802239244,
0.584452376204567, 0.29538916638251, -0.424471774108761,
1.35117053520231, 0.792966672512426, 0.559666965721712,
1.03877148575442, 0.32764651319845, 0.792431599069095,
1.79713629328279, 2.53306185903747, 0.382061987152509,
-0.55974023663989, 0.261351966632211, 0.928359586004826,
1.05805881312766, -0.448798293155081, -2.8433140252059,
-1.29380365284521, 1.60167210548413, -0.58790657908656,
-0.0697276516437701, 0.669259446155372, -0.756109095074059,
-1.04262502361173, -0.689533522981508, 0.322514092974764,
-0.62456134593389, 0.343601164613668, 0.406496690190247,
-0.579352431691941, -0.38067184267295, 1.15818332237678,
-1.3763494217139, 1.07842256464695, -0.607885118048254,
-0.551750338671028, -0.688013574614753, -0.66192239892944,
0.840882344143739, 0.501181908666563, 0.810882707408453,
-0.373132840815414, -1.53884108045858, -0.0475950419868607,
-1.11456391432642, -1.39312192248506, 0.374292584707849,
0.307055843720151, 0.0883771102062163, 1.51499635303431,
0.544284404231116, 1.62863647405725, 0.666268752934375,
3.15259591439161, 0.535584045927088, 0.438326104669433,
1.25375087298954, 1.2784768691421
};

coef = imsls_f_vector_autoregression(n_obs, n_cols, y, p,
    IMSLS_MA_LAG, q,
    IMSLS_N_STEPS, max_step,
    IMSLS_FORECASTS_USER, forecasts,
    0);

imsls_f_write_matrix(" * * * Forecasts * * *\n", n_obs, n_cols*max_step,
    forecasts,
    IMSLS_COL_LABELS, clabel,
    0);

if(coef){
    imsls_free(coef);
    coef = NULL;
}
}

```

Output

```

* * * Forecasts * * *
t      t+1,Y1      t+1,Y2      t+2, Y1      t+2, Y2
1      0.000      0.000      0.000      0.000
2      0.000      0.000      0.000      0.000
3      0.000      0.000      0.000      0.000
4      0.000      0.000      0.000      0.000
5      0.000      0.000      0.000      0.000
6      0.000      0.000      0.000      0.000
7      0.009      -0.739      0.002      -0.086
8      -0.501      0.893      -0.017      -0.050
9      0.773      -1.779      0.024      -0.479

```

10	-0.935	0.042	-0.038	-0.251
11	0.164	0.959	-0.067	0.306
12	0.393	-1.377	0.009	-0.263
13	-0.672	-0.873	0.013	-0.058
14	-0.447	0.644	0.036	0.368
15	0.187	1.540	0.027	0.689
16	0.897	-0.117	0.191	-0.172
17	0.020	0.621	-0.039	0.556
18	0.008	-1.719	0.079	-0.444
19	-1.059	0.790	-0.056	0.397
20	0.193	0.264	0.035	-0.319
21	0.194	-1.050	-0.121	-0.377
22	-0.415	0.529	-0.042	0.039
23	0.433	0.658	0.048	0.495
24	0.074	0.493	0.044	0.235
25	0.402	-1.035	0.114	-0.230
26	-0.634	0.428	-0.073	0.351
27	0.109	-0.094	0.059	-0.061
28	-0.026	0.133	0.009	-0.164
29	0.067	1.258	-0.078	0.435
30	0.677	0.780	0.105	0.674
31	0.210	-0.299	0.145	-0.250
32	-0.200	-0.262	-0.086	0.671
33	-0.656	-1.024	0.134	-0.676
34	-0.351	-0.130	-0.169	-0.188
35	-0.152	0.708	-0.084	0.377
36	0.351	-1.377	0.122	-0.147
37	-0.826	-0.399	-0.026	-0.125
38	-0.256	0.717	-0.018	-0.026
39	0.716	-0.898	0.071	-0.139
40	-0.316	-1.153	0.058	-0.236
41	-0.804	1.543	-0.100	0.486
42	0.765	0.139	0.071	-0.257
43	0.443	-0.054	0.039	-0.434
44	0.345	-1.797	-0.050	-0.503
45	-1.165	0.730	-0.124	-0.028
46	0.531	1.303	-0.022	0.174
47	0.765	1.889	0.014	0.390
48	1.281	0.188	0.069	-0.089
49	0.306	-1.741	0.017	-0.569
50	-1.140	0.239	-0.172	-0.441

Example 3

In this example we fit the following VARMA(1,1) restricted model on simulated data of two dimensions:

$$\begin{pmatrix} 1 & 0 \\ -\alpha_{21,0} & 1 \end{pmatrix} y_t - \begin{pmatrix} \alpha_{11,1} & \alpha_{12,1} \\ \alpha_{21,1} & \alpha_{22,1} \end{pmatrix} y_{t-1} = \begin{pmatrix} 1 & 0 \\ -\alpha_{21,0} & 1 \end{pmatrix} u_t + \begin{pmatrix} 0 & m_{12,1} \\ m_{21,1} & 0 \end{pmatrix} u_{t-1}$$

In this specification there are 7 free parameters:

$$\gamma = (\alpha_{21,0}, \alpha_{11,1}, \alpha_{21,1}, \alpha_{12,1}, \alpha_{22,1}, \alpha_{21,1}, \alpha_{12,1})'$$

The first stage and the second stage results are shown. From the second stage results, we see that (to two decimals) the estimate is

$$\hat{y} = (-0.03, 0.00, 0.05, -0.01, -0.06, 0.01)'$$

```

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

int main(){
    int k, n_obs=100, n_cols=2, p=1, q=1;
    int ar[]={0, -1, 0, 0, 1, 1, 1, 1};
    int ma[]={0, -1, 0, 0, 0, 1, 1, 0};
    float ar_c[]={1, 0, 0, 1, 0, 0, 0, 0};
    float ma_c[]={1, 0, 0, 1, 0, 0, 0, 0};
    float *coef=NULL;
    int max_lag=6, n_coef1, n_coef2;
    float *coef_ttests1=NULL, *coef_ttests2=NULL;
    Imsls_f_regression *stage1_var_info=NULL;
    Imsls_f_regression *stage2_varma_info=NULL;
    char *clabel[] = {"id", "coef", "SE", "t-stat", "p-value"};

    float y[] =
    {
        0.143280117400598, 0.691517079320758, -1.054002748442,
        1.89368500251305, -0.595435384283415, -1.53385197609914,
        -0.450087415747451, -0.522758296071267, 0.338719815948114,
        1.73890633300759, -0.0420130132503371, -0.00933890926865466,
        -0.939527210947825, -0.279211957506804, 0.985111509818102,
        -0.185903912738057, -0.779058630583496, -1.52829111438157,
        -0.0444517646926054, -0.605935345974391, 1.17870357395841,
        0.979822006409954, -1.75509211790604, -1.02617250037744,
        -2.64443752185661, -0.405444884498378, -0.558146570012133,
        1.17006139568333, -0.14877513906561, 1.67436399145195,
        1.21251151094695, -0.236411746432856, -0.319260279201159,
        1.53774676549506, -0.798919508505848, -1.25907348772775,
        -0.43488363747126, 1.18754392780486, 2.49394567166528,
        -0.505392075680617, -0.939902530453777, -1.3000118234638,
        0.308365204823071, -0.0346715133254558, 0.155821836363299,
        1.53865066350577, -0.446013548645569, 0.421382795732249,
        -0.810472750765929, -0.475790066827614, -1.21547965787564,
        0.873092852598299, 0.314687304446453, -0.166494291509063,
        1.20846773815425, -0.21319122737697, 0.885697825416125,
        1.06749862455588, -0.417811475765902, 1.81350258917296,
        1.06312903931625, -0.357098401483029, -2.54962241395723,
        1.58241298127273, -0.445333714405381, -1.54921054521057,
        0.763932954657703, -0.459132068443737, -0.135020632775658,
        0.768987806710127, -2.21000914520305, -0.416578967811937,
        -0.787803924930718, -0.381994679294779, 0.651886558080692,
        -0.296275077784937, -1.34100151116192, -0.695511844572305,
        -1.38111408367782, -0.453263483406621, 0.546836142779091,
        1.35617195687341, 1.47675672302528, -0.879275538764735,
        1.33724523036386, -1.19560541543992, -0.355724070126155,
        -1.35440645398376, 0.215557787562229, -0.0705280128055462,
        2.64174061572298, 0.622022977819918, 1.85193603585307,
        1.1425463214074, 1.47034707550646, -0.274392879035261,
        -0.0256438865664471, -1.32768501270967, 1.4002259906097,
        -1.13393234222336, 0.420685154939234, -1.14369662556446,
        -0.229525100642792, -1.14556440309372, -0.0726173507068031,
        0.537003424016424, 2.21081287812981, -0.329847605902733,
        0.658135565199115, 1.25161597712012, 0.324279854980608,
        -0.12277642651712, -0.746398274242844, -0.510384971856952,
        -1.52397549713935, -0.590932096161281, 0.351879848132452,
        -0.486815223667361, -0.582122931146207, -0.129708090826525,
    }
}

```

```

-1.83261132721672, -0.438855077878885, 0.160891671954672,
0.130903505342111, -1.0930038744212, 0.858667240391389,
-0.650140333935879, -0.192590383440759, -0.495902099869389,
-0.0970274914548782, 0.271597266914655, -1.25629606008009,
-1.82411099200514, -0.0862331652538604, -1.48736902275701,
-0.702589236231933, 1.66371111576656, 0.260453048198016,
0.472465295167692, 2.03959666287312, -1.47220802239244,
0.584452376204567, 0.29538916638251, -0.424471774108761,
1.35117053520231, 0.792966672512426, 0.559666965721712,
1.03877148575442, 0.32764651319845, 0.792431599069095,
1.79713629328279, 2.53306185903747, 0.382061987152509,
-0.55974023663989, 0.261351966632211, 0.928359586004826,
1.05805881312766, -0.448798293155081, -2.8433140252059,
-1.29380365284521, 1.60167210548413, -0.58790657908656,
-0.0697276516437701, 0.669259446155372, -0.756109095074059,
-1.04262502361173, -0.689533522981508, 0.322514092974764,
-0.62456134593389, 0.343601164613668, 0.406496690190247,
-0.579352431691941, -0.38067184267295, 1.15818332237678,
-1.3763494217139, 1.07842256464695, -0.607885118048254,
-0.551750338671028, -0.688013574614753, -0.66192239892944,
0.840882344143739, 0.501181908666563, 0.810882707408453,
-0.373132840815414, -1.53884108045858, -0.0475950419868607,
-1.11456391432642, -1.39312192248506, 0.374292584707849,
0.307055843720151, 0.0883771102062163, 1.51499635303431,
0.544284404231116, 1.62863647405725, 0.666268752934375,
3.15259591439161, 0.535584045927088, 0.438326104669433,
1.25375087298954, 1.2784768691421
};

coef=imsls_f_vector_autoregression(n_obs, n_cols, y, p,
    IMSLS_MA_LAG, q,
    IMSLS_A0,
    IMSLS_AR_MODEL, ar,
    IMSLS_MA_MODEL, ma,
    IMSLS_MAX_LAG, max_lag,
    IMSLS_AR_CONSTANTS, ar_c,
    IMSLS_MA_CONSTANTS, ma_c,
    IMSLS_VAR_INFO, &stage1_var_info,
    IMSLS_VARMA_INFO, &stage2_varma_info,
    0);

n_coef1=n_cols*max_lag;

for(k=0;k<n_cols;k++){
    imsls_f_regression_summary(stage1_var_info,
        IMSLS_INDEX_REGRESSION, k,
        IMSLS_COEF_T_TESTS, &coef_ttests1,
        0);

    imsls_f_write_matrix(" * * * VAR Stage 1 Coefficients * * *\n",
        n_coef1, 4, coef_ttests1,
        IMSLS_COL_LABELS, clabel,
        0);

    imsls_free(coef_ttests1);
}

/* stage 2 restricted model
estimates 1 A0 parameter, 4 A1 parameters, and 2 M1 parameters */
n_coef2=7;

```

```

imsls_f_regression_summary(stage2_varma_info,
    IMSLS_COEF_T_TESTS, &coef_ttests2,
    0);

imsls_f_write_matrix(" * * * VARMA Stage 2 Coefficients * * *\n",
    n_coef2, 4, coef_ttests2,
    IMSLS_COL_LABELS, clabel,
    0);

if(stage1_var_info){
    imsls_free(stage1_var_info);
    stage1_var_info=NULL;
}
if(stage2_varma_info){
    imsls_free(stage2_varma_info);
    stage2_varma_info=NULL;
}
if(coef){
    imsls_free(coef);
    coef=NULL;
}
if(coef_ttests2){
    imsls_free(coef_ttests2);
    coef_ttests2=NULL;
}
}

```

Output

```
 * * * VAR Stage 1 Coefficients * * *
```

id	coef	SE	t-stat	p-value
1	0.229	0.110	2.075	0.041
2	0.292	0.125	2.345	0.021
3	-0.274	0.110	-2.501	0.014
4	-0.242	0.127	-1.896	0.062
5	0.372	0.116	3.208	0.002
6	0.280	0.128	2.198	0.031
7	-0.027	0.116	-0.232	0.817
8	-0.349	0.128	-2.722	0.008
9	0.240	0.109	2.208	0.030
10	0.296	0.128	2.316	0.023
11	-0.145	0.110	-1.323	0.190
12	-0.144	0.126	-1.136	0.259

```
 * * * VAR Stage 1 Coefficients * * *
```

id	coef	SE	t-stat	p-value
1	0.182	0.099	1.840	0.069
2	0.177	0.112	1.577	0.119
3	-0.287	0.099	-2.907	0.005
4	0.064	0.115	0.561	0.577
5	0.075	0.104	0.721	0.473
6	0.066	0.115	0.579	0.564
7	-0.111	0.104	-1.066	0.290
8	-0.193	0.115	-1.675	0.098
9	-0.066	0.098	-0.675	0.501
10	0.018	0.115	0.160	0.873

11	0.034	0.099	0.348	0.729
12	0.092	0.114	0.813	0.418
* * * VARMA Stage 2 Coefficients * * *				
id	coef	SE	t-stat	p-value
1	-0.099	0.232	-0.429	0.668
2	0.130	0.097	1.346	0.180
3	-0.050	0.208	-0.240	0.811
4	-0.618	0.242	-2.559	0.011
5	0.123	0.115	1.068	0.287
6	0.204	0.257	0.791	0.430
7	0.914	0.270	3.388	0.001

Multivariate Analysis

Functions

Hierarchical Cluster Analysis

Computes matrix of dissimilarities or similarities	dissimilarities	923
Hierarchical cluster analysis	cluster_hierarchical	928
Retrieves cluster numbers in hierarchical cluster analysis	cluster_number	934

K-means Cluster Analysis

Performs a K-means (centroid) cluster analysis	cluster_k_means	939
--	---------------------------------	-----

Principal Component Analysis

Computes principal components	principal_components	945
---	--------------------------------------	-----

Factor Analysis

Extracts factor-loading estimates.	factor_analysis	952
Performs discriminant function analysis	discriminant_analysis	973

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 [Correlation and Covariance](#).)

Step 2

Initial Factor Extraction

[imsls_f_factor_analysis](#)

Step 3

Factor Rotation <code>imsls_f_factor_analysis</code>	
Orthogonal	Oblique
No Prior Info. <code>IMSLS_ORTHOMAX_ROTATION</code>	No Prior Info. <code>IMSLS_OBLIQUE_PROMAX_ROTATION</code> <code>IMSLS_DIRECT_OBLIMIN_ROTATION</code> <code>IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION</code>
Prior Info. <code>IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION</code>	Prior Info. <code>IMSLS_OBLIQUE_PROCRUSTES_ROTATION</code>

Step 4

<p>Factor Structure and Variance</p> <p><code>imsls_f_factor_analysis</code> optional argument <code>IMSLS_FACTOR_STRUCTURE</code></p>

dissimilarities



[more...](#)

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 $m = \text{nrow}$ if optional argument `IMSLS_ROWS` is used, and $m = \text{ncol}$ otherwise.

Synopsis with Optional Arguments

```
#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 (Input)

or

IMSLS_COLUMNS, (Input)

Exactly one of these options can be present to indicate whether distances are computed between rows or columns of \mathbf{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.

<i>imeth</i>	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
5	Angle in radians ($0, \pi$) between the lines through the origin defined by the vectors

<code>imeth</code>	Method
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.

`IMSL_S_SCALE`, *int* `iscale` (Input)

Scaling option. `iscale` is not used for methods 3 through 8.

<code>iscale</code>	Scaling Performed
0	No scaling is performed.
1	Scale each column (row, if <code>IMSL_S_ROWS</code> is used) by the standard deviation of the column (row).
2	Scale each column (row, if <code>IMSL_S_ROWS</code> is used) by the range of the column (row).

Default: `iscale` = 0.

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

Column dimension of `x`.

Default: `x_col_dim` = `ncol`.

`IMSL_S_RETURN_USER`, *float* `dist[]` (Output)

User allocated array of size m by m containing the computed dissimilarities or similarities, where $m = \text{nrow}$ if `IMSL_S_ROWS` is used, and $m = \text{ncol}$ 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 `IMSL_S_COLUMNS` is used. If distances between the rows of the matrix are desired, use optional argument `IMSL_S_ROWS`.

For `imeth` = 0 to 2, each row of \mathbf{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 $z_k = (x_k - y_k)/s_k$, $i = 1, \dots, \mathbf{ndstm}$, 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:

<code>imeth</code>	Metric
0	$\sqrt{\left(\sum_{i=1}^{\mathbf{ndstm}} z_i^2\right)}$ Euclidean distance
1	$\sum_{i=1}^{\mathbf{ndstm}} z_i $ L_1 norm
2	$\max_i z_i $ 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:

<code>imeth</code>	Metric
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	$\cos(\theta) = X^T Y / (\sqrt{X^T X} \sqrt{Y^T Y})$ = 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>

int 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



[more...](#)

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/Output)

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 Arguments

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

```
float *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.

<code>imeth</code>	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 <code>dist</code> are assumed to be Euclidean distances.

Default: `imeth = 0`.

`IMSLS_TRANSFORMATION`, *int itrans* (Input)

Option giving the method to be used for clustering.

<code>imeth</code>	Method
0	No transformation is required. The elements of <code>dist</code> 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.

Default: `itrans = 0`.

`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`.

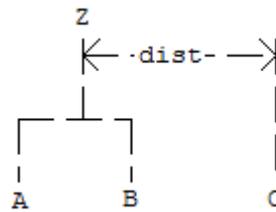
Description

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 $n = \text{npt}$.

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 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 `cleve1`) 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>
#include <stdio.h>

int 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)
An array 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)
An array of length *npt* - 1 containing the right 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

An array of length *npt* containing the cluster membership of each observation.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_cluster_number (int npt, int iclson[], int icrson[], int k,
    IMSLS_OBS_PER_CLUSTER, int **nclus,
    IMSLS_OBS_PER_CLUSTER_USER, int nclus[],
```

```

    IMSLS_RETURN_USER, int iclus[ ],
    0)

```

Optional Arguments

`IMSLS_OBS_PER_CLUSTER, 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_CLUSTER_USER, int nclus[]` (Output)

Storage for array `nclus` is provided by the user. See `IMSLS_OBS_PER_CLUSTER`.

`IMSLS_RETURN_USER, float iclus[]` (Output)

User allocated array of length `npt` containing the cluster membership of each observation.

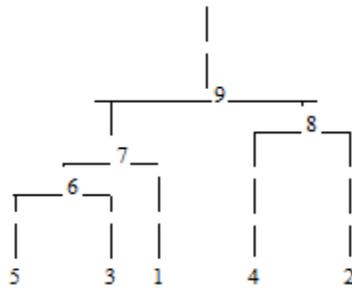
Description

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.

Examples

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>

int 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`, [Utilities](#)) is clustered. First the distance between the irises is 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>
#include <stdlib.h>

#define MAX(A,B) ((A)>(B)?(A):(B))

int 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_HISTORY, int *n_itr, int **cluster_history,

    IMSLS_CLUSTER_HISTORY_USER, int *n_itr, int cluster_history[ ],

    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_HISTORY, *int* *n_itr, *int* **cluster_history (Output)

cluster_history is a pointer to an array of size n_iter by n_observations containing the cluster membership of each observation per iteration. Note that n_iter is the number of completed iterations in the algorithm.

IMSLS_CLUSTER_HISTORY_USER, *int* *n_itr, *int* cluster_history[] (Output)

Storage for array cluster_history is provided by the user. cluster_history is an array of size max_iterations by n_observations containing the cluster membership of each observation per iteration. Note that only the first n_itr rows of cluster_history is set upon return.

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 $p = \text{n_variables}$ 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,

$$\phi = \sum_{i=1}^K \sum_{j=1}^p \sum_{m=1}^{n_i} f_{v_{im}} w_{v_{im}} \delta_{v_{im},j} (x_{v_{im},j} - \bar{x}_{ij})^2$$

where v_{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 v_{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 <imsls.h>

int 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);
}

```

Output

```

                                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

IMSL_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` by `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 (Input)

Treat the input vector `covariances` as a covariance matrix.

Default = `IMSLS_COVARIANCE_MATRIX`.

or

IMSLS_CORRELATION_MATRIX (Input)

Treat the input vector `covariances` as a correlation matrix.

Default = `IMSLS_COVARIANCE_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` by `n_variables` containing the eigenvectors of `covariances`, stored column-wise. 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** by **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 <imsls.h>

int main()
{
#define N_VARIABLES 9

float *values;
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. */
imsls_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 <imsls.h>

int 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 */
imsls_free(values);
imsls_free(eigenvectors);
imsls_free(cum_percent);
imsls_free(std_dev);
imsls_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_PROCRUSTES_ROTATION, float target[ ], float **b, float **t,  
    IMSLS_ORTHOGONAL_PROCRUSTES_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[ ], int norm,
    float target[ ], float b[ ], float t[ ],
    float 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, float 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.

Default: `IMSLS_UNWEIGHTED_LEAST_SQUARES`

or

IMSLS_PRINCIPAL_COMPONENT

Principal component (principal component model) method used to obtain the estimates.

Default: **IMSLS_UNWEIGHTED_LEAST_SQUARES**

or

IMSLS_PRINCIPAL_FACTOR

Principal factor (common factor model) method used to obtain the estimates.

Default: **IMSLS_UNWEIGHTED_LEAST_SQUARES**

or

IMSLS_UNWEIGHTED_LEAST_SQUARES

Unweighted least-squares (common factor model) method used to obtain the estimates. This option is the default.

Default: 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.

Default: **IMSLS_UNWEIGHTED_LEAST_SQUARES**

or

IMSLS_IMAGE

Image-factor analysis (common factor model) method used to obtain the estimates.

Default: **IMSLS_UNWEIGHTED_LEAST_SQUARES**

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.

Default: **IMSLS_UNWEIGHTED_LEAST_SQUARES**

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 $1 - n_factors/2 \times n_variables$ divided by the diagonal elements of the inverse of **covariances**.

`IMSLI_UNIQUE_VARIANCES_OUTPUT`, *float unique_variances[]* (Output)

User-allocated array of length `n_variables` containing the estimated unique variances.

`IMSLI_MAX_ITERATIONS`, *int max_iterations* (Input)

Maximum number of iterations in the iterative procedure.

Default: `max_iterations = 60`

`IMSLI_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`

`IMSLI_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`

`IMSLI_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`

`IMSLI_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.

`IMSLI_EIGENVALUES_USER`, *float eigenvalues[]* (Output)

Storage for array `eigenvalues` is provided by the user. See `IMSLI_EIGENVALUES`.

`IMSLI_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.

`IMSLI_TUCKER_RELIABILITY_COEFFICIENT`, *float *coefficient* (Output)

Tucker reliability coefficient.

`IMSLI_N_ITERATIONS`, *int *n_iterations* (Output)

Number of iterations.

IMSL_FUNCTION_MIN, *float *function_min* (Output)

Value of the function minimum.

IMSL_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).

IMSL_LAST_STEP_USER, *float last_step[]* (Output)

Storage for array **last_step** is provided by the user. See **IMSL_LAST_STEP**.

IMSL_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** by **n_factors** containing the rotated factor loading matrix. **t** contains the address of a pointer to the internally allocated array of length **n_factors** by **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**).

IMSL_ORTHOMAX_ROTATION_USER, *float w, int norm, float b[], float t[]* (Input/Output)

Storage for **b** and **t** are provided by the user. See **IMSL_ORTHOMAX_ROTATION**.

IMSL_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.

IMSL_ORTHOGONAL_PROCRUSTES_ROTATION_USER, *float target[], float b[], float t[]*

(Input/Output)

Storage for **b** and **t** are provided by the user. See

IMSL_ORTHOGONAL_PROCRUSTES_ROTATION.

IMSL_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` \times `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.

`IMSL_S_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
`IMSL_S_DIRECT_OBLIMIN_ROTATION`.

`IMSL_S_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`, contains 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` \times `n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors` \times `n_factors` containing the rotation transformation matrix. `factor_correlations` contains the address of a pointer to the internally allocated array of length `n_factors` \times `n_factors` containing the factor correlations. `target` contains the address of a pointer to the internally allocated array of length `n_variables` \times `n_factors` containing the target matrix for rotation, derived from the orthomax rotation. `w` is used in the orthomax rotation, see the optional argument `IMSL_S_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.

`IMSL_S_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
`IMSL_S_OBLIQUE_PROMAX_ROTATION`.

`IMSL_S_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. Non-negative constant `w` defines the rotation. `pivot`, a vector of length `n_factors`, contains 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` \times `n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors` \times `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 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`

IMSL_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, $p = \text{n_variables}$ and $k = \text{n_factors}$. 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 $\Phi_{U1} = 0.5 (\text{trace} (S - \Sigma)^2)$.

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:

$$\rho = \frac{mM_0 - mM_k}{mM_0 - 1}$$

$$m = d - \frac{2p + 5}{6} - \frac{2k}{6}$$

$$M_0 = \frac{-\ln(|S|)}{p(p-1)/2}$$

$$M_k = \frac{\phi}{((p-k)^2 - p - k)/2}$$

where $|S|$ is the determinant of covariances, $p = \text{n_variables}$, $k = \text{n_variables}$, ϕ is the optimized criterion, and $d = \text{df_covariances}$.

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 mea-

sured. 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 `IMSL_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 \mathbf{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 \mathbf{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 `convergence_eps` (the user-specified convergence criterion), the algorithm stops. `convergence_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, $\gamma \leq 0$ 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 rota-

tions 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 `convergence_eps` (the user-specified convergence criterion), the algorithm stops. `convergence_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 perform 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 $\gamma = \text{diag}(\beta^T \beta)$ 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]/bij`. 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 <imsls.h>

int 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[0][0], 3,
        0);

    /* Print results */
    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS, a,
        0);

    imsls_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 <imsls.h>
#include <stdio.h>

int 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[0][0], 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);

imsls_free(evals);
imsls_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
	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
	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 <imsls.h>

int 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 $\text{TRAN}(A)^{\text{target}} = \#$. This must be greater than or equal to $n_{\text{factors}} = \#$.

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, *or*
IMSLS_ROWS_DELETE,
IMSLS_PRIOR_EQUAL, *or*
IMSLS_PRIOR_PROPORTIONAL, *or*
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.

<code>ido</code>	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 <code>n_rows</code> observations of <code>x</code> will be performed.
2	This is an intermediate invocation; updating for the <code>n_rows</code> observations of <code>x</code> will be performed.
3	All statistics are updated for the <code>n_rows</code> observations. The discriminant functions and other statistics are computed.
4	The discriminant functions are used to classify each of the <code>n_rows</code> observations of <code>x</code> .
5	The covariance matrices are computed, and workspace is released. No further call to <code>discriminant_analysis</code> with <code>ido</code> greater than 1 should be made without first calling <code>discriminant_analysis</code> with <code>ido = 1</code> .
6	Workspace is released. No further calls to <code>discriminant_analysis</code> with <code>ido</code> greater than 1 should be made without first calling <code>discriminant_analysis</code> with <code>ido = 1</code> . Invo-cation with this option is not required if a call has already been made with <code>ido = 5</code> .

Default: `ido = 0`

`IMSLS_ROWS_ADD` (Input)

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` (Input)

or

`IMSLS_PRIOR_PROPORTIONAL` (Input)

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 $\sum w_i f_i x_i$ 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** \times **n_variables** elements comprise the group 1 covariance matrix, the next **n_variables** \times **n_variables** elements comprise the group 2 covariance, ..., and the last **n_variables** \times **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 $i = 1, 2, \dots, \mathbf{n_groups}$. Array **coefficients** is always computed as the linear discriminant function coefficients even when quadratic discrimination is specified. Specifically, given the linear discriminant function

$$z_i = \ln(p_i) - 0.5\bar{x}_i^T S_p^{-1} \bar{x}_i + x^T S_p^{-1} \bar{x}_i$$

the intercept $\ln(p_i) - 0.5\bar{x}_i^T S_p^{-1} \bar{x}_i$ is assigned to **coefficients**[$i \times (\mathbf{n_variables} + 1)$] and the j -th element of $S_p^{-1} \bar{x}_i$ is assigned to **coefficients**[$i \times (\mathbf{n_variables} + 1) + j$], where $i = 0, \dots, \mathbf{n_groups} - 1$ and $j = 1, \dots, \mathbf{n_variables} + 1$. 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[]` (Output)

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 $4 + 2 \times (\mathbf{n_groups} + 1)$ containing various statistics of interest. Array `stats` is updated when `ido` is equal to 0, 2, 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 $5 + \mathbf{n_groups}$ 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 $4 + \mathbf{n_groups}$). Finally, the last $\mathbf{n_groups} + 1$ 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

- Common choices for the Bayesian prior probabilities are given by:
 - `prior_input[i] = 1.0/n_groups` (equal priors)
 - `prior_input[i] = gcounts[i]/n_rows` (proportional priors)
 - `prior_input[i] = Past history or subjective judgment.`
 In all cases, the priors should sum to 1.0.
- 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.
- Here are a few rules and guidelines for the correct value of `ido` in a series of calls:

- a. 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.
- b. 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.
- c. `ido` may not be 4 or 5 before a call with `ido = 3` has been made.
- d. `ido` may not be 2, 3, 4, 5, or 6
 - i) Immediately after a call with `ido = 0`.
 - ii) Before a call with `ido = 1` has been made.
 - iii) Immediately after a call with `ido` equal to 5 or 6 has been made.

The following is a valid sequence of `ido`'s:

<code>ido</code>	Explanation
0	Data Set A: Perform a complete analysis. All data to be used in the analysis must be present in <code>x</code> . Since cleanup of workspace is automatic for <code>ido = 0</code> , no further calls are necessary.
1	Data Set B: Begin analysis. The <code>n_rows</code> observations in <code>x</code> are used for initialization.
2	Data Set B: Continue analysis. New observations placed in <code>x</code> are added to (or deleted from, see <code>IMSL_ROWS_DELETE</code>) the analysis.
2	Data Set B: Continue analysis. <code>n_rows</code> new observations placed in <code>x</code> are added to (or deleted from, see <code>IMSL_ROWS_DELETE</code>) the analysis.
3	Data Set B: Continue analysis. <code>n_rows</code> new observations are added (or deleted) and discriminant functions and other statistics are computed.
4	Data Set B: Classification of each of the <code>n_rows</code> observations in the current <code>x</code> matrix.
5	Data Set B: End analysis. Covariance matrices are computed and workspace is released. This analysis could also have been ended by choosing <code>ido = 6</code>
1	Data Set C: Begin analysis. Note that for this call to be valid the previous call must have been made with <code>ido</code> 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.

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,

$$S_i = \frac{1}{N_i - 1} \sum_{j=1}^{M_i} w_j f_j (x_j - \bar{x})(x_j - \bar{x})^T$$

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 \bar{x} denotes the mean vector of the observations in group i . The mean vectors are computed as

$$\bar{x} = \left(\frac{1}{W_i} \right) \sum_{j=1}^{M_i} w_j f_j x_j \quad \text{where } W_i = \sum_{j=1}^{M_i} w_j f_j$$

Given the means and the covariance matrices, the linear discriminant function for group i is computed as:

$$z_i = \ln(p_i) - 0.5\bar{x}_i^T S_p^{-1} \bar{x}_i + x^T S_p^{-1} \bar{x}_i$$

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:

$$D_{ij}^2 = (\bar{x}_i - \bar{x}_j)^T S^{-1} (\bar{x}_i - \bar{x}_j)$$

Finally, the asymptotic chi-squared test for the equality of covariance matrices is computed as follows (Morrison 1976, p. 252):

$$\gamma = C^{-1} \sum_{i=1}^k n_i \{ \ln(|S_p|) - \ln(|S_i|) \}$$

where n_i is the number of degrees of freedom in the i -th sample covariance matrix, k is the number of groups, and

$$C^{-1} = 1 - \frac{2p^2 + 3p - 1}{6(p + 1)(k - 1)} \left(\sum_{i=1}^k \frac{1}{n_i} - \frac{1}{\sum_j n_j} \right)$$

where p is the number of variables.

When `ido = 4`, the estimated posterior probability of each observation x belonging to group i 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

$$\hat{q}_i(x) = \frac{\exp(-0.5D_i^2(x))}{\sum_{j=1}^k \exp(-0.5D_j^2(x))}$$

where

$$D_i^2(x) = \begin{cases} (x - \bar{x}_i)^T S_i^{-1} (x - \bar{x}_i) + \ln|S_i| - 2\ln(p_i) & \text{method} = 1 \text{ or } 2 \\ (x - \bar{x}_i)^T S_p^{-1} (x - \bar{x}_i) - 2\ln(p_i) & \text{method} = 3 \end{cases}$$

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][igrp]` (see optional argument `IMSLS_X_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).

Examples

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

int 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);

    imsls_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      -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>

int 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;
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);

imsls_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*(n_groups+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 #.

Survival and Reliability Analysis

Functions

Survival Analysis

Computes Kaplan-Meier estimates of survival probabilities	kaplan_meier_estimates	1000
Analyzes survival and reliability data using Cox's proportional hazards model	prop_hazards_gen_lin	1007
Analyzes survival data using the generalized linear model	survival_glm	1022
Estimates using various parametric modes	survival_estimates	1046

Reliability Analysis

Estimates a reliability hazard function using a nonparametric approach	nonparam_hazard_rate	1053
--	--------------------------------------	------

Actuarial Tables

Produces population and cohort life tables	life_tables	1063
--	-----------------------------	------

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 described 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_PRINT,  
    IMSLS_X_RESPONSE_COL, int irt,
```

```
IMSLS_CENSOR_CODES_COL, int icen,  
IMSLS_FREQ_RESPONSE_COL, int ifrq,  
IMSLS_STRATUM_NUMBER_COL, int igrp,  
IMSLS_SORTED,  
IMSLS_N_MISSING, int *nrmiss,  
IMSLS_RETURN_USER, float table[ ],  
0)
```

Optional Arguments

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, *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.

IMSL_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.

IMSL_N_MISSING, *int *nrmiss* (Output)

Number of rows of data in `x` containing missing values.

IMSL_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.

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 $i = 1, \dots, n$ denote the failure censoring times and the censoring codes for the n observations in a single sample. Here, $t_i = \mathbf{x}_{i-1, irt}$ is a failure time if δ_i is 0, where $\delta_i = \mathbf{x}_{i-1, icen}$. 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 $s_{(j)}$ be denoted by $n_{(j)}$, where $s_{(j)}$ 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 $s_{(j)}$ to $s_{(j+1)}$ (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 $n_{(j)}$ 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 \mathbf{x} during a single call. This is accomplished through the `igrp` column of \mathbf{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 \mathbf{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 \mathbf{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 \mathbf{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>

int 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, ifrq,
        IMSLS_CENSOR_CODES_COL, icen,
        IMSLS_STRATUM_NUMBER_COL, igrp,
        0);
}
```

Output

```
Kaplan Meier Survival Probabilities
For Group Value = 5
```

Number at risk	Number Failing	Time	Survival Probability	Estimated 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 at risk	Number Failing	Time	Survival Probability	Estimated 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` \times `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 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** \times 4, **coef**, containing the parameter estimates and associated statistics.

Column	
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_DUMMY, 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.

<code>iprint</code>	Action
0	No printing is performed.
1	Printing is performed, but observational statistics are not printed.
2	All output statistics are printed.

Default: `iprint` = 0.

`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 `alg1` 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. Let

$$r_k = \exp\left(z_k \hat{\beta} + w_k\right)$$

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 `xk,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 $r_{\min} > \text{ratio} r_{\max}$, 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.

Default: `ratio` = 1000.0.

`IMSLS_X_RESPONSE_COL`, *int* `irt` (Input)

Column index in `x` containing the response variable. For point observations, `xi,irt` contains the time of the i -th event. For right-censored observations, `xi,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.

<code>x_{i,icen}</code>	Censoring
0	Exact censoring time <code>x_{i,irt}</code> .
1	Right censored. The exact censoring time is greater than <code>x_{i,irt}</code> .

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.

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.

Default: **itie** = 0.

IMSLS_MAXIMUM_LIKELIHOOD, *float *alg1* (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_DUMMY`, *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 [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 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 $m = \text{istrat}$ 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 $100 * \mathbf{eps}$, 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

$$r_{ki} = \exp(w_{ki} + z_{ki}\hat{\beta}) \sum_{j \in R(t_{ki})} \frac{d_{kj}}{\sum_{\ell \in R(t_{kj})} \exp(w_{k\ell} + z_{k\ell}\hat{\beta})}$$

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

$$e_j = \sum_{l \leq j} \frac{1}{h-l+1}$$

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

$$\hat{H}_{k0}(t_{ik}) = \sum_{t_{kj} \leq t_{ki}} \frac{d_{kj}}{\sum_{l \in R(t_{kj})} \exp(w_{kl} + z_{kl}\hat{\beta})}$$

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 \mathbf{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 $z_{ki}\hat{\beta}$. 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 \mathbf{X} from largest to smallest with respect to the failure/censoring times $\mathbf{x}_{i, \text{irt}}$ 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 $t_i = \mathbf{x}_{i, \text{irt}}$ 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. If `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 $\rho = \text{ratio}$ is specified by the user. Thus, for example, if $\rho = 10000$, 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 $\rho > 0$. 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:

$$\ln(\lambda) = \beta_1x_1 + \beta_2x_2 + \beta_3x_3 + \alpha_i + \gamma_j$$

where α_i and γ_j correspond to dummy variables generated from column indices 5 and 6 of \mathbf{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 \mathbf{x} .

```
#include <imsls.h>

#define NOBS 40
#define NCOL 7
#define NCLVAR 2
#define NEF 5

int 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 error	Asymptotic z-statistic	Asymptotic 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 Probability	Influence	Residual	Cumulative hazard	Prop. 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	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	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
5	Logistic
6	Log least extreme value
7	Least extreme value
8	Log extreme value

model	PDF of the Response Variable
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, or
```

```
    IMSLS_NO_INTERCEPT,
```

```
    IMSLS_INFINITY_CHECK, int lp_max, or
```

```
IMSLS_NO_INFINITY_CHECK,
IMSLS_EFFECTS, int n_effects, int n_var_effects[ ], int indices_effects,
IMSLS_INITIAL_EST_INTERNAL, or
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[ ],
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.

<code>x</code>	<code>icen</code>	Censoring type
0		Exact failure at <code>x [i] [irt]</code>
1		Right Censored. The response is greater than <code>x [i] [irt]</code> .
2		Left Censored. The response is less than or equal to <code>x [i] [irt]</code> .
3		Interval Censored. The response is greater than <code>x [i] [irt]</code> , but less than or equal to <code>x [i] [ilt]</code> .

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 right-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 right-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_X_COL_FREQUENCIES`, *int* `ifrq` (Input)

Column number of `x` containing the frequency of response for each observation.

`IMSLS_X_COL_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, (Input)

Indicates the intercept is automatically included in the model.

Default: **IMSLS_INTERCEPT**

or

IMSLS_NO_INTERCEPT, (Input)

Indicates there is no intercept in the model (unless otherwise provided for by the user).

Default: **IMSLS_INTERCEPT**

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: `IMSLS_NO_INFINITY_CHECK`; `lp_max = 0`

or

`IMSLS_NO_INFINITY_CHECK`

Iterates without checking for infinite estimates.

Default: `IMSLS_NO_INFINITY_CHECK`

`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`, (Input)

Indicates unweighted linear regression is used to obtain initial estimates.

Default: `IMSLS_INITIAL_EST_INTERNAL`

or

`IMSLS_INITIAL_EST_INPUT`, *int* `n_coef_input`, *float* `estimates []` (Input)

Indicates 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`.

Default: `IMSLS_INITIAL_EST_INTERNAL`

`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

$$\sum_{i=0}^{n_class-1} n_class_values[i]$$

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:

	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:

	Description
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[]* (Output)

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:

	Statistic
0	Estimated predicted value.
1	Estimated influence or leverage.
2	Estimated residual.

	Statistic
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.

	Status of Observation
0	Observation <i>i</i> is in the likelihood
1	Observation <i>i</i> cannot be in the likelihood because it contains at least one missing value in <i>x</i> .
2	Observation <i>i</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.

	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 function `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 $\eta = \mathbf{x}^T \boldsymbol{\beta}$ be the linear parameterization, where \mathbf{x} is a design vector obtained by `imsls_f_survival_glm` via function `imsls_f_regressors_for_glm` from a row of \mathbf{x} , and $\boldsymbol{\beta}$ 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 \mathbf{x}). Use of this parameter is discussed below. There also may be nuisance parameters $\theta > 0$, or $\sigma > 0$ to be estimated (along with $\boldsymbol{\beta}$) in the various models. Let Φ denote the cumulative normal distribution. The survival models available in `imsls_f_survival_glm` are:

	Name	
0	Exponential	$\exp \left[-t \exp \left(w_i + \eta \right) \right]$
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	$\left\{ 1 + \exp \left(\frac{\ln(t) - \eta - w_i}{\sigma} \right) \right\}^{-1}$
5	Logistic	$\left\{ 1 + \exp \left(\frac{t - \eta - w_i}{\sigma} \right) \right\}^{-1}$

	Name	
6	Log least extreme value	$\exp \left\{ -\exp \left(\frac{\ln(t) - \eta - w_i}{\sigma} \right) \right\}$
7	Least extreme value	$\exp \left\{ -\exp \left(\frac{t - \eta - w_i}{\sigma} \right) \right\}$
8	Log extreme value	$1 - \exp \left\{ -\exp \left(\frac{\ln(t) - \eta - w_i}{\sigma} \right) \right\}$
9	Extreme value	$1 - \exp \left\{ -\exp \left(\frac{t - \eta - w_i}{\sigma} \right) \right\}$
10	Weibull	$\exp \left\{ - \left[\frac{t}{\exp(w_i + \eta)} \right]^\theta \right\}$

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 \mathbf{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 $\lambda = \exp(\mathbf{x}\boldsymbol{\beta})$ may be known a priori. 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 $\lambda = \exp(x_1\boldsymbol{\beta}_1 + x_2\boldsymbol{\beta}_2)$, where $\boldsymbol{\beta}_2$ is known. Letting $W_i = x_2\boldsymbol{\beta}_2$, estimates for $\boldsymbol{\beta}_1$ can be obtained via `imsls_f_survival_glm` with the known fixed values for $\boldsymbol{\beta}_2$. Standard methods can then be used to test hypothesis about $\boldsymbol{\beta}_1$ 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 `IMSL_S_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

$$S^{-1}(\hat{S}(t)) - w_i = \alpha + \phi t'$$

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

$$S^{-1}(\hat{S}(t)) - \hat{\phi} t' - w_i = x^T \beta$$

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 α_j denotes one of the parameters 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

$$\eta_j = x_j^T \beta$$

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_j . If some (or all) of the removed observations should not have been removed (because their estimated η_j 's must be finite), then the iterations are restarted with a log-likelihood based upon the finite η_j 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 $l_i(\theta_i)$ denote the log-likelihood of the i -th observation evaluated at θ_i , let I_i denote the vector of derivatives of l_i with respect to all parameters, $I'_{h,i}$ denote the derivative of l_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

$$(I'_i)^T H^{-1} I'_i$$

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 $I'_{h,i}$.

- 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>

int 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>
#include <stdio.h>

int 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

$$\chi^2 = \mathbf{g}^T \hat{\Sigma}^{-1} \mathbf{g}$$

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).

$$\chi_{LR}^2 = 2(206.6835 - 204.1392) = 5.0886$$

Neither statistic is significant at the $\alpha = 0.05$ level.

```

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

int 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 Errors

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 + ixdelta` for $i = 0, 1, \dots, npt - 1$.

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 $(2 \times \text{n_observations} + 1)$ 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 $i = 1, 2, \dots, \text{npt}$.

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 $(2 \times \text{n_observations} + 1)$ 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 $i = 1, 2, \dots, \text{npt}$.

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 `xp`t (`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 `xp`t). 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:

	Name	
0	Exponential	$\exp[-t \exp(w_i + \eta)]$
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	$\left\{1 + \exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\right\}^{-1}$
5	Logistic	$\left\{1 + \exp\left(\frac{t - \eta - w_i}{\sigma}\right)\right\}^{-1}$
6	Log least extreme value	$\exp\left\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\right\}$
7	Least extreme value	$\exp\left\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\right\}$
8	Log extreme value	$1 - \exp\left\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\right\}$
9	Extreme value	$1 - \exp\left\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\right\}$
10	Weibull	$\exp\left\{-\left[\frac{t}{\exp(w_i + \eta)}\right]^\theta\right\}$

Let $\lambda(t)$ denote the hazard rate at time t . Then $\lambda(t)$ and $S(t)$ are related at

$$S(t) = \exp\left(\int_{-\infty}^t \lambda(s) ds\right)$$

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>

int 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);

imsls_free (survival_info);
imsls_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 Errors

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 `IMSL_S_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 $j = 0, \dots, n_hazard - 1$.

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_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, int k_min, int 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[],
    IMSLS_RETURN_USER, float haz[],
    0)
```

Optional Arguments

IMSLS_PRINT_LEVEL, *int* `iprint` (Input)

Printing option.

<code>iprint</code>	Action
0	No printing is performed.
1	The grid estimates and the optimized estimates are printed for each value of k .

Default: `iprint` = 0.

IMSLS_CENSOR_CODES, *int* `ensor_codes` [] (Input)

`ensor_codes` is an array of length `n_observations` containing the censoring codes for each time in `t`. If `ensor_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 $\text{weight} = \ln(1 + 1 / (n_{\text{observations}} - i))$ is used for the i -th smallest observation. Otherwise, $\text{weight} = 1 / (n_{\text{observations}} - i)$ 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`, *int* `k_min`, *int* `k_increment` (Input)

Finds the optimal value of k over the range given by: $k_{\text{min}} + (j - 1) \times k_{\text{increment}}$, for $j = 1, \dots, n_k$. 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 $\beta^* = -8, -4, -2, -1, -0.5, 0.5, 1, \text{ and } 2$, 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**.

IMSL_RETURN_USER, float haz [] (Output)

If specified, haz is a user supplied array of length n_hazard containing the estimated hazard rates.

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,

$$K(x) = \begin{cases} \frac{15}{16}(1-x^2)^2 & \text{for } |x| < 1 \\ 0 & \text{elsewhere} \end{cases}$$

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

$$h(x) = \sum_{i=1}^N \left\{ (1 - \delta_i) w_i K_S(x - x_{(i)}) \right\}$$

where $N = n_{\text{observations}}$, 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

$$L = \prod_{i=1}^N \left\{ h(x_i)^{(1-\delta_i)} \exp(-H(x_i)) \right\}$$

where \prod 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>
#include <stdio.h>

int 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
    };

    int 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,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1
    };

    haz = imsls_f_nonparam_hazard_rate (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

IMSL5_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 $|\text{age}[0]|$ and that the initial age interval is from 0.0 to $|\text{age}[0]|$. 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.

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 a user-specified array of length `n_classes`×12 containing the life table.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)

Printing option.

<code>iprint</code>	Action
0	No printing is performed.
1	The life table is printed.

Default: `iprint` = 0.

`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 user-specified 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 $i = 0, 1, \dots, t_n$ 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 $u_i \neq h_i$, 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, $S_0 = \text{initial_pop}$ for population table, and $S_0 = \text{n_cohort} [0]$ 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)

$$\text{var}(e_i) = \frac{\sum_{j=i}^{n-1} P_j^2 \sigma_j^2 [e_{j+1} + h_{j+1} (1 - a_j)]^2}{P_i^2}$$

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

int 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		

Probability Distribution Functions and Inverses

Functions

Discrete Random Variables: Distribution Functions and Probability Functions

Distribution Functions

Binomial distribution function	binomial_cdf	1074
Binomial probability function	binomial_pdf	1076
Discrete geometric cumulative distribution function.	geometric_cdf	1078
Inverse of the discrete geometric cumulative distribution function.	geometric_inverse_cdf	1080
Discrete geometric probability density function	geometric_pdf	1082
Hypergeometric distribution function	hypergeometric_cdf	1084
Hypergeometric probability function	hypergeometric_pdf	1087
Poisson distribution function	poisson_cdf	1089
Poisson probability function	poisson_pdf	1092
Discrete uniform cumulative distribution function	discrete_uniform_cdf	1094
Inverse of the discrete uniform cumulative distribution function.	discrete_uniform_inverse_cdf	1096
Discrete uniform probability density function	discrete_uniform_pdf	1098

Continuous Random Variables

Distribution Functions and their Inverses

Beta distribution function	beta_cdf	1100
Inverse beta distribution function.	beta_inverse_cdf	1102
Noncentral beta cumulative distribution function (CDF).	non_central_beta_cdf	1104
Inverse of the noncentral beta CDF.	non_central_beta_inverse_cdf	1107
Noncentral beta probability density function (PDF)	non_central_beta_pdf	1110
Bivariate normal distribution function.	bivariate_normal_cdf	1113
Chi-squared distribution function	chi_squared_cdf	1115
Inverse chi-squared distribution function.	chi_squared_inverse_cdf	1118
Calculates the complement of the chi-squared distribution.	complementary_chi_squared_cdf	1120
Noncentral chi-squared distribution function	non_central_chi_sq	1123
Inverse of the noncentral chi-squared distribution function	non_central_chi_sq_inv	1127
Noncentral chi-squared probability density function (PDF)	non_central_chi_sq_pdf	1129

Exponential cumulative distribution function	exponential_cdf	1132
Inverse of the exponential cumulative distribution function	exponential_inverse_cdf	1134
Exponential probability density function.	exponential_pdf	1136
<i>F</i> distribution function.	F_cdf	1138
Inverse <i>F</i> distribution function	F_inverse_cdf	1141
Calculates the complement of the <i>F</i> distribution function	complementary_F_cdf	1143
Noncentral <i>F</i> probability density function (PDF).	non_central_F_pdf	1146
Noncentral <i>F</i> cumulative distribution function (CDF).	non_central_F_cdf	1150
Calculates the complement of the noncentral <i>F</i> CDF	complementary_non_central_F_cdf	1153
Inverse of the noncentral <i>F</i> distribution function	non_central_F_inverse_cdf	1157
Gamma distribution function	gamma_cdf	1160
Inverse gamma distribution function	gamma_inverse_cdf	1163
Lognormal cumulative distribution function	lognormal_cdf	1165
Inverse of the lognormal cumulative distribution function	lognormal_inverse_cdf	1167
Lognormal probability density function	lognormal_pdf	1169
Normal (Gaussian) distribution function.	normal_cdf	1171
Multivariate normal distribution function	multivariate_normal_cdf	1174
Inverse normal distribution function.	normal_inverse_cdf	1182
Student's <i>t</i> distribution function	t_cdf	1184
Inverse Student's <i>t</i> distribution function.	t_inverse_cdf	1187
Calculates the complement of the Student's <i>t</i> distribution.	complementary_t_cdf	1189
Noncentral Student's <i>t</i> distribution function	non_central_t_cdf	1193
Inverse of the noncentral Student's <i>t</i> distribution function	non_central_t_inv_cdf	1196
Pareto cumulative probability distribution function.	pareto_cdf	1201
Pareto probability density function.	pareto_pdf	1203
 Parameter Estimation		
Maximum likelihood estimation for univariate probability distributions	max_likelihood_estimates	1205

Usage Notes

The *distribution function* for the (real, single-valued) random variable X is the function F defined for all real x by

$$F(x) = \Pr(X \leq x)$$

where $\Pr(\cdot)$ denotes the probability of an event. The distribution function is often called the *cumulative distribution function* (CDF).

In general, the CDF does not have an inverse because it is not one-to-one. Nevertheless, a *quantile function*, also called an *inverse CDF*, is well-defined as

$$F^{-1}(p) = \min \{x \in \mathfrak{X} : F(x) \geq p\}, p \in (0,1)$$

Here p represents a probability on the open interval, $(0, 1)$.

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.

Discrete Distributions

For discrete distributions, the function giving the probability that a random variable takes on specific values is called the *probability mass function*, or just *probability function*, defined by

$$f(x) = \Pr(X = x)$$

The CDF for a discrete random variable is

$$F(x) = \Pr[X \leq x] = \sum_A f(k)$$

where A is the set such that $k \leq x$.

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

$$\Pr(a < X \leq b) = \int_a^b f(x)dx$$

The relationship between the CDF and the PDF is

$$F(x) = \int_{-\infty}^x f(t)dt$$

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. Thus, the inverse-CDF is a proper inverse function on the interval $(0,1)$ and

$$F^{-1}(F(x)) = x .$$

The “`_inverse_cdf`” functions described in this chapter compute the inverses of the distribution functions. The inverses are defined only over the open interval $(0,1)$.

Parameter Notation and Estimation

To emphasize the dependence of a PDF or CDF on one or more parameters, we use the following notation:

$$f(x|\boldsymbol{\theta}) \text{ or } F(x|\boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ represents one or more distributional parameters. The vertical bar “|” is read as “given”. Some authors prefer to use the semi-colon instead of the vertical bar, as in $f(x; \boldsymbol{\theta})$, to emphasize this dependency. However, the vertical bar is more consistent with the notation used for conditional probability distributions. For instance, in the Bayesian framework, there is a distribution on the parameters, so in that sense the probability distributions treated in this chapter are conditional distributions.

A related task to evaluating a probability density or distribution function is to estimate the values of its parameters. For many of the distributions covered in this chapter, function `imsls_f_max_likelihood_estimates` provides maximum likelihood estimates of the unknown parameter values given a sample of observations.

Additional Comments

1. In order to keep the calling sequences simple, whenever possible the functions 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”.

2. 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 to standardize the variables by subtracting the mean and dividing by the square root of the variance.
3. 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.)
4. Many of the algorithms used by functions 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.
5. 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 functions 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).
6. 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 functions 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.

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:

$$\Pr(X = j) = \frac{(n + 1 - j)p}{j(1 - p)} \Pr(X = j - 1)$$

such that:

$$F(k|n,p) = \sum_{i=0}^k \Pr(X = j)$$

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 $(1 - p)^n \epsilon$ if forward computation is performed and by $p^n \epsilon$ 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>
#include <stdio.h>

int main()
{
    int    k = 3, n = 5;
    float  p = 0.95, pr;

    pr = imsls_f_binomial_cdf(k,n,p);
    printf("Pr(x <= %d) = %6.4f\n", k, 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)
```

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**. Specifically,

$$f(k|n,p) = \Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

where $k = \{0,1,2,\dots,n\}$, $n \geq 1$, $0 \leq p \leq 1$, and

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

These probabilities are computed by the recursive relationship:

$$\Pr(X = k) = \frac{(n+1-k)p}{k(1-p)} \Pr(X = k-1)$$

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 summing the probabilities, which are rescaled by $(1-p)^n \epsilon$ if forward computation is performed and by $p^n \epsilon$ 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.

Examples

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>

int main()
{
    int k = 3, n = 5;
    float p = 0.95, prob;

    prob = imsls_f_binomial_pdf(k, n, p);
    printf("The probability that X is equal to "
           "%d is %f\n", k, prob);
}
```

Output

```
The probability that X is equal to 3 is 0.021434
```

geometric_cdf

Evaluates the discrete geometric cumulative distribution function (CDF).

Synopsis

```
#include<imsls.h>

floatimsls_f_geometric_cdf(intix, floatpin)
```

The type *double* function is `imsls_d_geometric_cdf`.

Required Arguments

intix (Input)

Argument for which the discrete geometric CDF is to be evaluated. *ix* must be non-negative.

floatpin (Input)

Probability parameter of the discrete geometric CDF (the probability of success for each independent trial). *pin* must be in the open interval (0, 1).

Return Value

The probability that a discrete geometric random variable takes a value less than or equal to *ix*. A value of NaN is returned if an input value is in error.

Description

The function `geometric_cdf` evaluates the discrete geometric cumulative distribution function (CDF), defined

$$F(I|p) = \sum_{i=0}^I p(1-p)^i = 1 - (1-p)^{I+1}, \quad 0 < p < 1$$

where the return value $F(I|p)$ is the probability that up to $I = \mathbf{ix}$ trials would be observed before observing a success, and input parameter $p = \mathbf{pin}$ is the probability of success for each independent trial.

Example

In this example, we evaluate the discrete geometric CDF at $ix = 3$, $pin = 0.25$.

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

int main()
{
    int    ix = 3;
    float  pin = 0.25;
    float  p;

    p = imsls_f_geometric_cdf(ix, pin);
    printf("The probability that a discrete geometric ");
    printf("random variable\nwith probability ");
    printf("parameter pin = %4.2f is less than ", pin);
    printf("or equal\nto %li is %8.6f\n\n", ix, p);
}
```

Output

```
The probability that a discrete geometric random variable
with probability parameter pin = 0.25 is less than or equal
to 3 is 0.683594
```

geometric_inverse_cdf

Evaluates the inverse of the discrete geometric cumulative distribution function (CDF).

Synopsis

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

```
intimsls_f_geometric_inverse_cdf(floatp, floatpin)
```

The type *double* function is `imsls_d_geometric_inverse_cdf`.

Required Arguments

floatp (Input)

Probability for which the inverse of the discrete geometric CDF is to be evaluated. *p* must be in the open interval (0, 1).

floatpin (Input)

Probability parameter of the discrete geometric CDF (the probability of success for each independent trial). *pin* must be in the open interval (0, 1).

Return Value

The probability that a discrete geometric random variable takes a value less than or equal to the returned value is the input probability, *p*. A value of -1 is returned if an input value is in error.

Description

The function `geometric_inverse_cdf` evaluates the inverse CDF of a discrete geometric random variable with parameter *pin*. The discrete geometric CDF is defined:

$$p = F(I|P) = \sum_{i=0}^I P(1-P)^i = 1 - (1-P)^{I+1}$$

where the return value $p = F(I|P)$ is the probability that up to I trials would be observed before observing a success, and input parameter $P = \mathbf{pin}$ is the probability of success for each independent trial. The discrete geometric inverse CDF is defined:

$$I = F^{-1}(p|P) = \left\lceil \frac{\log(1-p)}{\log(1-P)} - 1 \right\rceil$$

which is the smallest integer I such that the discrete geometric CDF is greater than or equal to input argument $p = \mathbf{p}$, where $0 < p < 1$, and input parameter $P = \mathbf{pin}$.

Example

In this example, we evaluate the inverse probability function at $\mathbf{pin} = 0.25$, $\mathbf{p} = 0.6835$.

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

int main()
{
    int    ix;
    float pin = 0.25;
    float p  = 0.6835;

    ix = imsls_f_geometric_inverse_cdf(p, pin);
    printf("The probability that a discrete geometric ");
    printf("random variable\nwith probability ");
    printf("parameter pin = %4.2f is less than ", pin);
    printf("or equal\nto %2i is %6.4f\n\n", ix, p);
}
```

Output

```
The probability that a discrete geometric random variable
with probability parameter pin = 0.25 is less than or equal
to 3 is 0.6835
```

geometric_pdf

Evaluates the discrete geometric probability density function (PDF).

Synopsis

```
#include<imsls.h>

floatimsls_f_geometric_pdf(intix, floatpin)
```

The type *double* function is `imsls_d_geometric_pdf`.

Required Arguments

intix (Input)

Argument for which the discrete geometric PDF is to be evaluated. `ix` must be non-negative.

floatpin (Input)

Probability parameter of the discrete geometric PDF (the probability of success for each independent trial). `pin` must be in the open interval (0, 1).

Return Value

The probability that a discrete geometric random variable having parameter `pin` will be equal to `ix`. A value of NaN is returned if an input value is in error.

Description

The function `geometric_pdf` evaluates the discrete geometric probability density function (PDF), defined

$$f(I | p) = p(1 - p)^I, \quad 0 < p < 1$$

where the return value $f(I|p)$ is the probability that $I = \text{ix}$ trials would be observed before observing a success, and input parameter $p = \text{pin}$ is the probability of success for each independent trial.

Example

In this example, we evaluate the discrete geometric PDF at $ix = 3$, $pin = 0.25$.

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

int main()
{
    int    ix = 3;
    float  pin = 0.25;
    float  p;

    p = imsls_f_geometric_pdf(ix, pin);
    printf("The probability density of a discrete ");
    printf("geometric\nrandom variable with ");
    printf("probability parameter pin = %4.2f\n", pin);
    printf("and value ix = %li is %8.6f\n\n", ix, p);
}
```

Output

```
The probability density of a discrete geometric
random variable with probability parameter pin = 0.25
and value ix = 3 is 0.105469
```

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

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

where $i = \max(0, n - l + m)$ and

$$F(k|n,m,l) = \sum_{i=0}^k \Pr(X = j)$$

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>
#include <stdio.h>

int main()
{
    int    k = 7, l = 1000, m = 70, n = 100;
    float  p;

    p = imsls_f_hypergeometric_cdf(k,n,m,l);
    printf("Pr (x <= %d) = %6.4f\n", k, p);
}
```

Output

```
Pr (x <= 7) = 0.5995
```

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>

floatimsls_f_hypergeometric_pdf (intk, intn, intm, intl)
```

The type *double* function is `imsls_d_hypergeometric_pdf`.

Required Arguments

- intk* (Input)
Argument for which the hypergeometric probability function is to be evaluated.
- intn* (Input)
Sample size. *n* must be greater than zero and greater than or equal to *k*.
- intm* (Input)
Number of defectives in the lot.
- intl* (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_hypergeometric_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

$$f(k|n,m,l) = \Pr(X = k) = \frac{\binom{m}{k} \binom{l-m}{n-k}}{\binom{l}{n}} \text{ for } k = i, i+1, i+2, \dots, \min(n,m)$$

where

$$\binom{m}{k} = \frac{m!}{k!(m-k)!}$$

and

$$i = \max(0, n - l + m).$$

`imsls_f_hypergeometric_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>
#include <stdio.h>

int main()
{
    int    k=7, n = 100, l = 1000, m = 70;
    float  pr;
    pr = imsls_f_hypergeometric_pdf(k, n, m, l);
    printf("The probability that X is equal to "
           "%d is %6.4f\n", k, 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_c (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 $\theta = \text{theta}$) is as follows:

$$f(x | \theta) = e^{-\theta} \theta^x / x!, \quad \text{for } x = 0, 1, 2, \dots$$

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

$$f(x+1 | \theta) = f(x | \theta) \left(\frac{\theta}{x+1} \right) \text{ for } x = 0, 1, 2, \dots, k-1$$

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

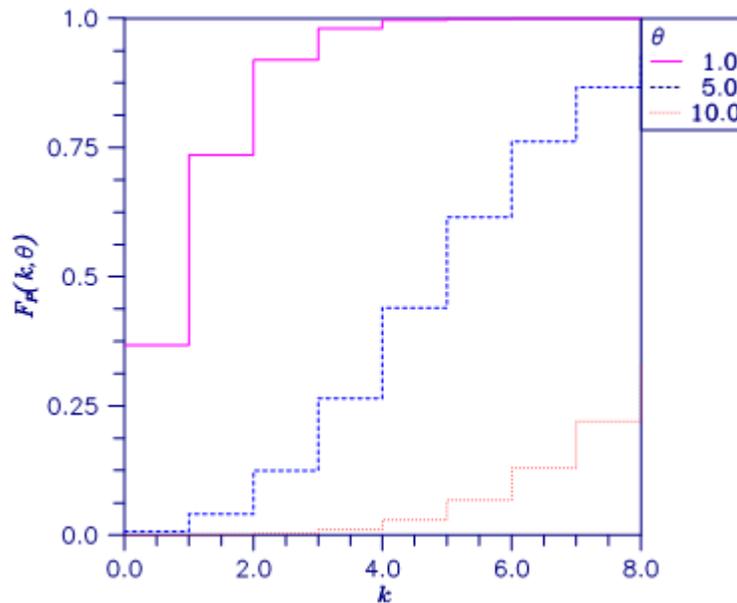


Figure 11, Plot of $F_p(k, \theta)$

Example

Suppose X is a Poisson random variable with $\theta = 10$. In this example, we evaluate the probability that X is less than or equal to 7.

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

int main()
{
    int    k = 7;
    float  theta = 10.0, p;

    p = imsls_f_poisson_cdf(k, theta);
    printf("Pr(x <= %d) = %6.4f\n", k, 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 $\theta = \text{theta}$) is

$$f(x | \theta) = e^{-\theta} \theta^k / k!, \quad \text{for } k = 0, 1, 2, \dots$$

`imsls_f_poisson_pdf` evaluates this function directly, taking logarithms and using the log gamma function.

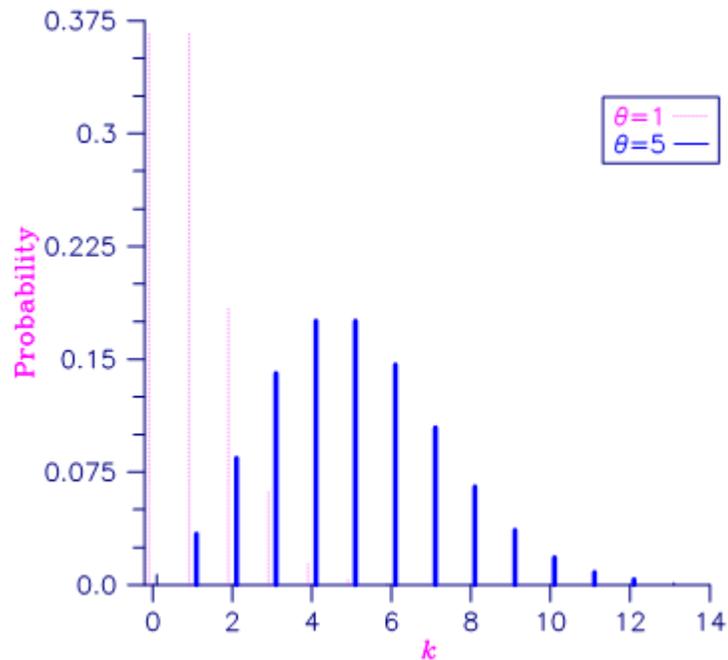


Figure 12, Poisson Probability Function

Example

Suppose X is a Poisson random variable with $\theta = 10$. In this example, we evaluate the probability function at 7.

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

int main () {
    int    k = 7;
    float theta = 10.0;

    printf ("The probability that X is equal to %d is %g.\n",
           k, imsls_f_poisson_pdf (k, theta));
}
```

Output

```
The probability that X is equal to 7 is 0.0900792.
```

discrete_uniform_cdf

Evaluates the discrete uniform cumulative distribution function (CDF).

Synopsis

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

```
floatimsls_f_discrete_uniform_cdf(int ix, int n)
```

The type *double* function is `imsls_d_discrete_uniform_cdf`.

Required Arguments

*int***ix** (Input)

Argument for which the discrete uniform CDF is to be evaluated. **ix** must be positive.

*int***n** (Input)

Scale parameter. **n** must be positive.

Return Value

The probability that a discrete uniform random variable takes a value less than or equal to **ix**. A value of NaN is returned if an input value is in error.

Description

The function `discrete_uniform_cdf` evaluates the discrete uniform cumulative distribution function (CDF) with scale parameter **n**, defined

$$F(I|N) = \frac{I}{N}, \quad 1 \leq I \leq N$$

where $I = \mathbf{ix}$ and $N = \mathbf{n}$.

Example

In this example, we evaluate the discrete uniform CDF at $ix = 3$, $n = 5$.

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

int main()
{
    int    ix = 3;
    int    n = 5;
    float  p;

    p = imsls_f_discrete_uniform_cdf(ix, n);
    printf("The probability that a discrete uniform ");
    printf("random variable\nwith scale ");
    printf("parameter n = %li is less than ", n);
    printf("or equal to %li\nis %6.4f\n\n", ix, p);
}
```

Output

```
The probability that a discrete uniform random variable
with scale parameter n = 5 is less than or equal to 3
is 0.6000
```

discrete_uniform_inverse_cdf

Evaluates the inverse of the discrete uniform cumulative distribution function (CDF).

Synopsis

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

```
int imsls_f_discrete_uniform_inverse_cdf(float p, int n)
```

The type *double* function is `imsls_d_discrete_uniform_inverse_cdf`.

Required Arguments

float **p** (Input)

Probability for which the inverse of the discrete uniform cumulative distribution function is to be evaluated. **p** must lie in the closed interval [0, 1].

int **n** (Input)

Scale parameter. **n** must be positive.

Return Value

The probability that a discrete uniform random variable takes a value less than or equal to the returned value is the input probability **p**. A value of -1 is returned if an input value is in error.

Description

The function `discrete_uniform_inverse_cdf` evaluates the integer value I of the discrete uniform inverse cumulative distribution function (CDF) with probability argument p and scale parameter N , i.e. the smallest integer $I \leq N$ with discrete uniform CDF value $\geq p$, defined

$$I = F^{-1}(p|N) = \lceil pN \rceil, \quad 0 \leq p \leq 1$$

where $p = \mathbf{p}$, $N = \mathbf{n}$, and $\lceil x \rceil$ is defined as the smallest integer \geq real value x .

Example

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

int main()
{
    float p = 0.60;
    int    n = 5;
    int    ix;

    ix = imsls_f_discrete_uniform_inverse_cdf(p, n);
    printf("The probability that a discrete uniform ");
    printf("random variable\nwith scale ");
    printf("parameter n = %li is less than ", n);
    printf("or equal to %2i\nis %4.2f\n\n", ix, p);
}
```

Output

```
The probability that a discrete uniform random variable
with scale parameter n = 5 is less than or equal to 3
is 0.60
```

discrete_uniform_pdf

Evaluates the discrete uniform probability density function (*PDF*).

Synopsis

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

```
floatimsls_f_discrete_uniform_pdf(int ix, int n)
```

The type *double* function is `imsls_d_discrete_uniform_pdf`.

Required Arguments

*int***ix** (Input)

Argument for which the discrete uniform PDF is to be evaluated. **ix** must be positive.

*int***n** (Input)

Scale parameter. **n** must be positive.

Return Value

The probability that a random variable from a discrete uniform distribution with scale parameter **n** will be equal to **ix**. A value of NaN is returned if an input value is in error.

Description

The function `discrete_uniform_pdf` evaluates the discrete uniform probability density function (PDF) with scale parameter **n**, defined

$$p = f(I|N) = \frac{1}{N}, \quad 1 \leq I \leq N$$

where $I = \mathbf{ix}$ and $N = \mathbf{n}$. As a convenience to the user, `discrete_uniform_pdf` accepts values of $I > N$, returning $p = 0$. `discrete_uniform_pdf` returns an error message for values of $I \leq 0$.

Example

In this example, we evaluate the discrete uniform PDF at $ix = 3$, $n = 5$.

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

int main()
{
    int    ix = 3;
    int    n = 5;
    float  p;

    p = imsls_f_discrete_uniform_pdf(ix, n);
    printf("The probability density of a discrete ");
    printf("uniform\nrandom variable with ");
    printf("scale parameter n = %1i\n", n);
    printf("and value ix = %1d is %6.4f\n\n", ix, p);
}
```

Output

```
The probability density of a discrete uniform
random variable with scale parameter n = 5
and value ix = 3 is 0.2000
```

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`. It is given by

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

where $\Gamma(\cdot)$ is the gamma function. This function is sometimes called the incomplete beta ratio and, with $p = \text{pin}$ and $q = \text{qin}$, is denoted by $I_x(p, q)$.

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>
#include <stdio.h>

int main()
{
    float pin = 12.0, qin = 12.0, x = 0.6, p;

    p = imsls_f_beta_cdf(x, pin, qin);
    printf("The probability that X is less than "
           "%3.1f is %6.4f\n", x, p);
    x = 0.5;
    p -= imsls_f_beta_cdf(x, pin, qin);
    printf("The probability that X is between "
           "%3.1f and", x);
    printf(" 0.6 is %6.4f\n", p);
}
```

Output

```
The probability that X is less than 0.6 is 0.8364
The probability that X is between 0.5 and 0.6 is 0.3364
```

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 $P = \mathbf{p}$, $p = \mathbf{pin}$, and $q = \mathbf{qin}$, 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. In other words:

$$F^{-1}(P|pin,qin) = x$$

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>
#include <stdio.h>

int main()
{
    float pin = 12.0, qin = 12.0, p = 0.9, x;

    x = imsls_f_beta_inverse_cdf(p, pin, qin);
    printf(" X is less than %6.4f with "
           "probability %3.1f.\n", x, p);
}
```

Output

```
X is less than 0.6299 with probability 0.9.
```

non_central_beta_cdf

Evaluates the noncentral beta cumulative distribution function (*CDF*).

Synopsis

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

```
float imsls_f_non_central_beta_cdf (float x, float shape1, float shape2, float lambda)
```

The type *double* function is `imsls_d_non_central_beta_cdf`.

Required Arguments

float x (Input)

Argument for which the noncentral beta cumulative distribution function is to be evaluated. **x** must be non-negative and less than or equal to 1.

float shape1 (Input)

First shape parameter of the noncentral beta distribution. **shape1** must be positive.

float shape2 (Input)

Second shape parameter of the noncentral beta distribution. **shape2** must be positive.

float lambda (Input)

Noncentrality parameter. **lambda** must be non-negative.

Return Value

The probability that a noncentral beta random variable takes a value less than or equal to **x**.

Description

The noncentral beta distribution is a generalization of the beta distribution. If Z is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and Y is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of Z , then

$$X = \frac{Z}{Z+Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1 (1-X)}$$

is a noncentral F -distributed random variable. The CDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F CDF

$$F_x(x|\alpha_1, \alpha_2, \lambda) = F_F(f|2\alpha_1, 2\alpha_2, \lambda)$$

where $F_x(x|\alpha_1, \alpha_2, \lambda)$ is a noncentral beta CDF with $x = \mathbf{x}$, $\alpha_1 = \mathbf{shape1}$, $\alpha_2 = \mathbf{shape2}$, and noncentrality parameter $\lambda = \mathbf{lambda}$; $F_F(f|2\alpha_1, 2\alpha_2, \lambda)$ is a noncentral F CDF with argument f , numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ ; and

$$f = \frac{\alpha_2}{\alpha_1} \frac{x}{1-x}; \quad x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$$

(See documentation for function [imsls_f_non_central_F_cdf](#) for a discussion of how the noncentral F CDF is defined and calculated.)

With a noncentrality parameter of zero, the noncentral beta distribution is the same as the beta distribution.

Example

This example traces out a portion of a noncentral beta distribution with parameters `shape1 = 50`, `shape2 = 5`, and `lambda = 10`.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int i;
    float f[] = {0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.8, 4.0};
    float x, shape1 = 50., shape2 = 5., lambda = 10.;
    float bcdfv, fcdfv, bcdfvexpect;

    printf ("shape1: %4.0f\n", shape1);
    printf ("shape2: %4.0f\n", shape2);
    printf ("lambda: %4.0f\n\n", lambda);
    printf (" x ncbetcdf(x) ncbetcdf(x)\n");
    printf (" expected\n");

    for (i=0; i<8; i++) {
        x = (shape1*f[i]) / (shape1*f[i] + shape2);
```

```
        fcdfv = imsls_f_non_central_F_cdf
            (f[i], 2.*shapel, 2.*shape2, lambda);
        bcdfvexpect = fcdfv;
        bcdfv = imsls_f_non_central_beta_cdf
            (x, shapel, shape2, lambda);
        printf (" %8.4f %12.4e %12.4e\n",
            x, bcdfvexpect, bcdfv);
    }
}
```

Output

```
shapel: 50
shape2: 5
lambda: 10
```

x	ncbetcdf(x)	ncbetcdf(x)
	expected	
0.0000	0.0000e+000	0.0000e+000
0.8000	4.8879e-003	4.8879e-003
0.8889	2.0263e-001	2.0263e-001
0.9231	5.2114e-001	5.2114e-001
0.9412	7.3385e-001	7.3385e-001
0.9524	8.5041e-001	8.5041e-001
0.9655	9.4713e-001	9.4713e-001
0.9756	9.8536e-001	9.8536e-001

non_central_beta_inverse_cdf

Evaluates the inverse of the noncentral beta cumulative distribution function (*CDF*).

Synopsis

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

```
float imsls_f_non_central_beta_inverse_cdf (float p, float shape1, float shape2,  
float lambda)
```

The type *double* function is `imsls_d_non_central_beta_inverse_cdf`.

Required Arguments

float p (Input)

Probability for which the inverse of the noncentral beta cumulative distribution function is to be evaluated. *p* must be non-negative and less than or equal to 1.

float shape1 (Input)

First shape parameter of the noncentral beta distribution. *shape1* must be positive.

float shape2 (Input)

Second shape parameter of the noncentral beta distribution. *shape2* must be positive.

float lambda (Input)

Noncentrality parameter. *lambda* must be non-negative.

Return Value

If the probability that a noncentral beta random variable takes a value less than or equal to *x* is *p*, then *x* is the return value of the noncentral beta inverse CDF evaluated at *p*.

Description

The noncentral beta distribution is a generalization of the beta distribution. If *Z* is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and *Y* is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of *Z*, then

$$X = \frac{Z}{Z+Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1 (1-X)}$$

is a noncentral F -distributed random variable. The CDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F CDF:

$$p = F_x(x|\alpha_1, \alpha_2, \lambda) = F_F(f|2\alpha_1, 2\alpha_2, \lambda)$$

where $F_x(x|\alpha_1, \alpha_2, \lambda)$ is a noncentral beta CDF with $x = x$, $\alpha_1 = \text{shape1}$, $\alpha_2 = \text{shape2}$, and noncentrality parameter $\lambda = \text{lambda}$; $F_F(f|2\alpha_1, 2\alpha_2, \lambda)$ is a noncentral F CDF with argument f , numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ ; $p =$ the probability that $F < f =$ the probability that $X < x$; and

$$f = \frac{\alpha_2 x}{\alpha_1 (1-x)}; \quad x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$$

(See documentation for function `imsls_f_non_central_F_cdf` for a discussion of how the noncentral F CDF is defined and calculated.) The correspondence between the arguments of function `imsls_f_non_central_beta_inverse_cdf` and the variables in the above equations is as follows: $\alpha_1 = \text{shape1}$, $\alpha_2 = \text{shape2}$, $\lambda = \text{lambda}$, and $p = p$.

Function `imsls_f_non_central_beta_inverse_cdf` evaluates

$$x = F_x^{-1}(p|\alpha_1, \alpha_2, \lambda)$$

by first evaluating

$$f = F_F^{-1}(p|2\alpha_1, 2\alpha_2, \lambda)$$

and then solving for x using

$$x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$$

(See documentation for function `imsls_f_non_central_F_inverse_cdf` for a discussion of how the inverse noncentral F CDF is calculated.)

Example

This example traces out a portion of an inverse noncentral beta distribution with parameters `shape1 = 50`, `shape2 = 5`, and `lambda = 10`.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int i;
    float f[] = {0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.8, 4.0};
    float shape1 = 50., shape2 = 5., lambda =10.;
    float x, p, bcdfinv;
    printf ("shape1: %4.0f\n", shape1);
    printf ("shape2: %4.0f\n", shape2);
    printf ("lambda: %4.0f\n\n", lambda);
    printf ("      x          p = cdf(x)      cdfinv(p)\n");
    for (i=0; i<8; i++) {
        x = (shape1*f[i]) / (shape1*f[i] + shape2);
        p = imsls_f_non_central_beta_cdf
            (x, shape1, shape2, lambda);
        bcdfinv = imsls_f_non_central_beta_inverse_cdf
            (p, shape1, shape2, lambda);
        printf (" %12.4e %12.4e %12.4e\n", x, p, bcdfinv);
    }
}
```

Output

```
shape1: 50
shape2: 5
lambda: 10
      x          p = cdf(x)      cdfinv(p)
0.0000e+000  0.0000e+000  0.0000e+000
8.0000e-001  4.8879e-003  8.0000e-001
8.8889e-001  2.0263e-001  8.8889e-001
9.2308e-001  5.2114e-001  9.2308e-001
9.4118e-001  7.3385e-001  9.4118e-001
9.5238e-001  8.5041e-001  9.5238e-001
9.6552e-001  9.4713e-001  9.6552e-001
9.7561e-001  9.8536e-001  9.7561e-001
```

non_central_beta_pdf

Evaluates the noncentral beta probability density function (*PDF*).

Synopsis

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

```
float imsls_f_non_central_beta_pdf (float x, float shape1, float shape2, float lambda)
```

The type *double* function is `imsls_d_non_central_beta_pdf`.

Required Arguments

float x (Input)

Argument for which the noncentral beta probability density function is to be evaluated. *x* must be non-negative and less than or equal to 1.

float shape1 (Input)

First shape parameter of the noncentral beta distribution. *shape1* must be positive.

float shape2 (Input)

Second shape parameter of the noncentral beta distribution. *shape2* must be positive.

float lambda (Input)

Noncentrality parameter. *lambda* must be non-negative.

Return Value

The probability density associated with a noncentral beta random variable with value *x*.

Description

The noncentral beta distribution is a generalization of the beta distribution. If Z is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and Y is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of Z , then

$$X = \frac{Z}{Z+Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1 (1-X)}$$

is a noncentral F -distributed random variable. The PDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F PDF:

$$f_x(x|\alpha_1, \alpha_2, \lambda) = f_F(f|2\alpha_1, 2\alpha_2, \lambda) \frac{df}{dx}$$

where $f_x(x|\alpha_1, \alpha_2, \lambda)$ is a noncentral beta PDF with $x = \mathbf{x}$, $\alpha_1 = \mathbf{shape1}$, $\alpha_2 = \mathbf{shape2}$, and noncentrality parameter $\lambda = \mathbf{lambda}$; $f_F(f|2\alpha_1, 2\alpha_2, \lambda)$ is a noncentral F PDF with argument f , numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ ; and:

$$f = \frac{\alpha_2 x}{\alpha_1 (1-x)}; \quad x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2};$$

$$\frac{df}{dx} = \frac{(\alpha_2 + \alpha_1 f)^2}{\alpha_1 \alpha_2} = \frac{\alpha_2}{\alpha_1 (1-x)^2}$$

(See documentation for function [imsls_f_non_central_F_pdf](#) for a discussion of how the noncentral F PDF is defined and calculated.)

With a noncentrality parameter of zero, the noncentral beta distribution is the same as the beta distribution.

Example

This example traces out a portion of a noncentral beta distribution with parameters $\mathbf{shape1} = 50$, $\mathbf{shape2} = 5$, and $\mathbf{lambda} = 10$.

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

int main()
{
    int i;
    float f[] = {0., .4, .8, 3.2, 5.6, 8.8, 14., 18.};
    float x, shape1 = 50., shape2 = 5., lambda = 10.;
    float bpdfv, fpdfv, bpdfvexpect, dfdx;

    printf ("shape1: %4.0f\n", shape1);
    printf ("shape2: %4.0f\n", shape2);
```

```

printf ("lambda: %4.0f\n\n", lambda);
printf ("      x      ncbetpdf(x)  ncbetpdf(x)\n");
printf ("          expected\n");

for (i=0; i<8; i++) {
  x = (shapel*f[i]) / (shapel*f[i] + shape2);
  dfdx = (shape2/shapel) / ((1. - x) * (1. - x));
  fpdfv = imsls_f_non_central_F_pdf
    (f[i], 2.*shapel, 2.*shape2, lambda);
  bpdfvexpect = dfdx * fpdfv;
  bpdfv = imsls_f_non_central_beta_pdf
    (x, shapel, shape2, lambda);
  printf (" %8.4f %12.4e %12.4e\n",
    x, bpdfvexpect, bpdfv);
}
}

```

Output

```

shapel: 50
shape2: 5
lambda: 10

```

x	ncbetpdf(x) expected	ncbetpdf(x)
0.0000	0.0000e+000	0.0000e+000
0.8000	2.4372e-001	2.4372e-001
0.8889	6.5862e+000	6.5862e+000
0.9697	4.0237e+000	4.0237e+000
0.9825	9.1954e-001	9.1954e-001
0.9888	2.1910e-001	2.1910e-001
0.9929	4.3665e-002	4.3665e-002
0.9945	1.7522e-002	1.7522e-002

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 $\rho = \text{rho}$, and $|\rho| < 1$,

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

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

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

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>
#include <stdio.h>

int main()
{
    float rho = 0.9, x = -2.0, y = 0.0, p;

    p = imsls_f_bivariate_normal_cdf(x, y, rho);
    printf(" The probability that X is less than %4.1f\n"
           " and Y is less than %3.1f is %6.4f\n", x, y, 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 cumulative distribution function (CDF).

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 0.

Return Value

The probability p 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, $F(x|v)$, of a chi-squared random variable $x = \text{chi_squared}$ with $v = \text{df}$ degrees of freedom, where:

$$F(x|v) = \frac{1}{2^{v/2} \Gamma(v/2)} \int_0^x e^{-t/2} t^{v/2-1} dt$$

and $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x .

For $v > v_{\max} = 1. \text{e}7$, `imsls_f_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun [A&S] 1964, Equation 26.4.17) for p in terms of the normal CDF, which is evaluated using function `imsls_f_normal_cdf`.

For $\nu \leq \nu_{\max}$, `imsls_f_chi_squared_cdf` uses series expansions to evaluate p : for $x < \nu$, `imsls_f_chi_squared_cdf` calculates p using A&S series 6.5.29, and for $x > \nu$, `imsls_f_chi_squared_cdf` calculates p using the continued fraction expansion of the incomplete gamma function given in A&S equation 6.5.31.

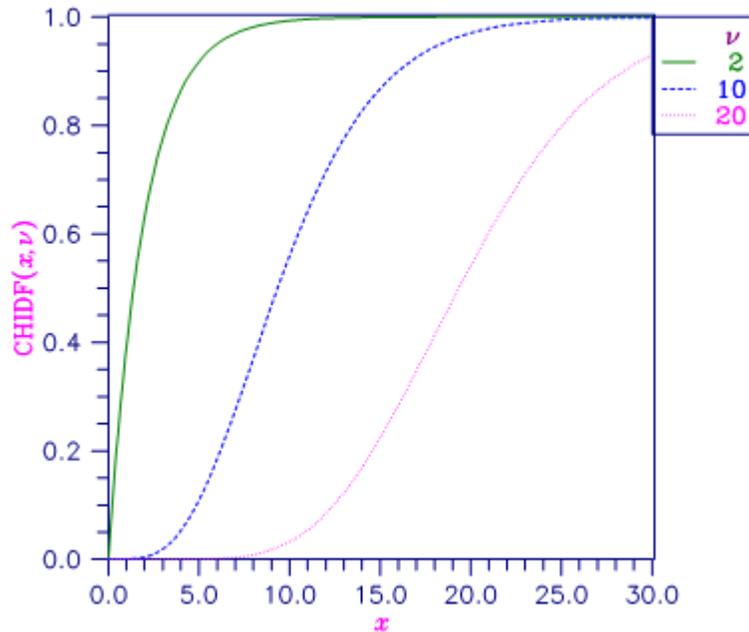


Figure 13, Plot of $F_x(x, df)$

Example

Suppose X is a chi-squared random variable with two degrees of freedom. In this example, we find the probability that X is less than 0.15 and the probability that X is greater than 3.0.

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

int main()
{
    float  chi_squared = 0.15, df = 2.0, p;

    p      = imsls_f_chi_squared_cdf(chi_squared, df);
    printf("The probability that chi-squared"
           " with %1.0f df is less than %4.2f is %5.4f\n",
           df, chi_squared, p);

    chi_squared = 3.0;
```

```
p = 1.0 - imsls_f_chi_squared_cdf(chi_squared, df);  
printf("The probability that chi-squared"  
      " with %1.0f df is greater than %3.1f is %5.4f\n",  
      df, chi_squared, p);  
}
```

Output

```
The probability that chi-squared with 2 df is less than 0.15 is 0.0723  
The probability that chi-squared with 2 df is greater than 3.0 is 0.2231
```

Informational Errors

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 0.

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 $v = df$ and with probability *p*. That is, it determines

$x = \text{imsls_f_chi_squared_inverse_cdf}(p, df)$, such that

$$p = F(x | v) = \frac{1}{2^{v/2} \Gamma(v/2)} \int_0^x e^{-t/2} t^{v/2-1} dt$$

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 $v < 40$, `imsls_f_chi_squared_inverse_cdf` uses bisection (if $v \leq 2$ or $p > 0.98$) or regula falsi to find the point at which the chi-squared distribution function is equal to p . The distribution function is evaluated using IMSL function `imsls_f_chi_squared_cdf`.

For $40 \leq v < 100$, 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 $v \geq 100$, 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>
#include <stdio.h>

int main ()
{
    float  p = 0.99, df = 2.0, x;

    x = imsls_f_chi_squared_inverse_cdf(p, df);
    printf("For p = %3.2f with %1.0f df, x = %7.3f.\n",
           p, df, x);

    df = 64.0;
    x = imsls_f_chi_squared_inverse_cdf(p,df);
    printf("For p = %3.2f with %2.0f df, x = %7.3f.\n",
           p, df, x);
}
```

Output

```
For p = .99 with 2 df, x = 9.210.
For p = .99 with 64 df, x = 93.217.
```

Warning Errors

<code>IMSLS_UNABLE_TO_BRACKET_VALUE</code>	The bounds that enclose “p” could not be found. An approximation for <code>imsls_f_chi_squared_inverse_cdf</code> is returned.
<code>IMSLS_CHI_2_INV_CDF_CONVERGENCE</code>	The value of the inverse chi-squared could not be found within a specified number of iterations. An approximation for <code>imsls_f_chi_squared_inverse_cdf</code> is returned.

complementary_chi_squared_cdf

Evaluates the complement of the chi-squared cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_complementary_chi_squared_cdf (float chi_squared, float df)
```

The type *double* function is `imsls_d_complementary_chi_squared_cdf`.

Required Arguments

float `chi_squared` (Input)

Argument for which the complementary chi-squared distribution function is to be evaluated.

float `df` (Input)

Number of degrees of freedom of the complementary chi-squared distribution. `df` must be greater than 0.

Return Value

The probability p that a chi-squared random variable takes a value greater than `chi_squared`.

Description

Function `imsls_f_complementary_chi_squared_cdf` evaluates the complement of the CDF, $1 - F(x|v)$, of a chi-squared random variable $x = \text{chi_squared}$ with $v = \text{df}$ degrees of freedom, where,

$$F(x|v) = \frac{1}{2^{v/2} \Gamma(v/2)} \int_0^x e^{-t/2} t^{v/2-1} dt$$

is the chi-squared CDF and $\Gamma(\cdot)$ is the gamma function. The value of the complementary chi-squared CDF at the point x is the probability that the random variable takes a value greater than x .

For $v > v_{\max} = 1.e7$, `imsls_f_complementary_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun [A&S] 1964, Equation 26.4.17) for p in terms of the normal CDF, which is evaluated using function `imsls_f_normal_cdf`.

For $v \leq v_{\max}$, `imsls_f_complementary_chi_squared_cdf` uses series expansions to evaluate p : for $x < v$, `imsls_f_complementary_chi_squared_cdf` calculates p using A&S series 6.5.29, and for $x \geq v$, `imsls_f_complementary_chi_squared_cdf` calculates p using the continued fraction expansion of the incomplete gamma function given in A&S equation 6.5.31.

Function `imsls_f_complementary_chi_squared_cdf` provides higher right tail accuracy for the complementary chi-squared distribution than does function `1 - imsls_f_chi_squared_cdf`.

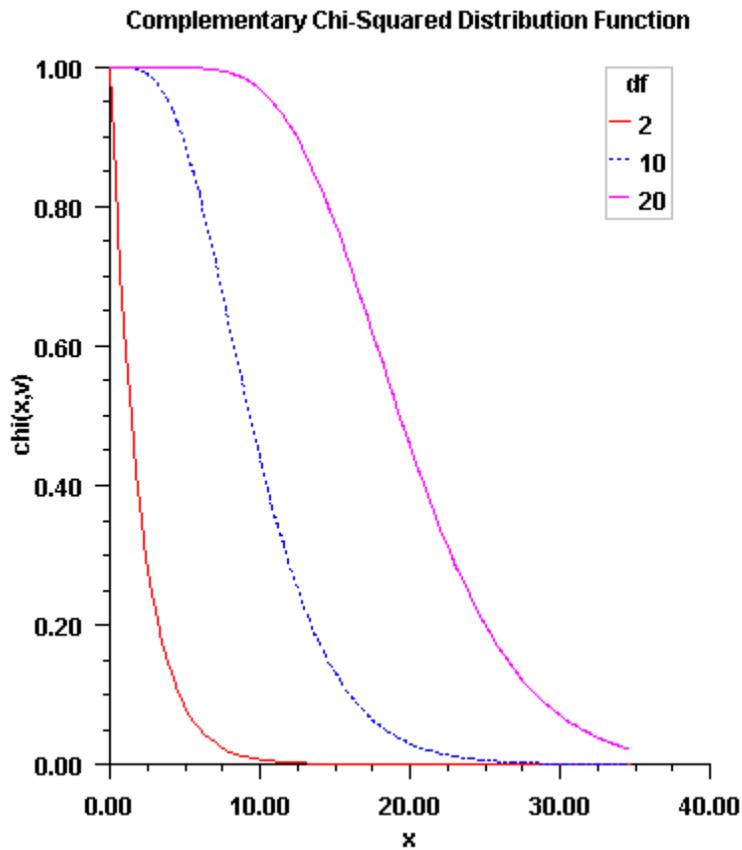


Figure 14, Plot of $F_x(x, df)$

Example

In this example, we find the probability that X , a chi-squared random variable, is less than 0.15 and the probability that X is greater than 3.0.

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

int main()
{
    float chi_squared = 0.15, df = 2.0, p;

    p = imsls_f_chi_squared_cdf(chi_squared, df);
    printf("The probability that chi-squared\n");
    printf(" with df = %1.0f is less than %4.2f is %6.4f\n",
           df, chi_squared, p);

    chi_squared = 3.0;
    p = imsls_f_complementary_chi_squared_cdf(chi_squared, df);
    printf("The probability that chi-squared\n");
    printf(" with df = %1.0f is greater than %4.2f is %6.4f\n",
           df, chi_squared, p);
}
```

Output

```
The probability that chi-squared
with df = 2 is less than 0.15 is 0.0723
The probability that chi-squared
with df = 2 is greater than 3.00 is 0.2231
```

Informational Errors

IMSLS_COMP_CHISQ_ZERO

Since "chi_squared" = # is less than zero, the distribution function is one at "chi_squared".

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 0.

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 $\nu = df$, $\lambda = \text{alam}$, and $x = \text{chi_squared}$,

$$F(x|v,\lambda) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} \int_0^x \frac{t^{(v+2i)/2-1} e^{-t/2}}{2^{(v+2i)/2} \Gamma\left(\frac{v+2i}{2}\right)} dt$$

where $\Gamma(\cdot)$ 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.

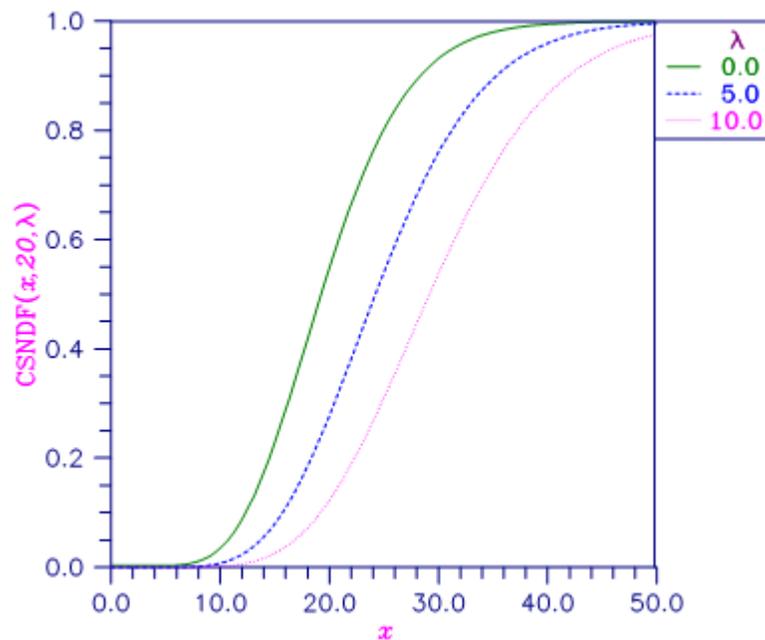


Figure 15, Noncentral Chi-squared Distribution Function

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>

int main()
{
    float chsq = 8.642, df = 2.0, alam = 1.0, p;

    p = imsls_f_non_central_chi_sq(chsq, df, alam);
    printf("The probability that a noncentral chi-squared "
           "random variable with %2.0f df and noncentrality "
           "parameter %3.1f is less 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
```

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 0.

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 $P = p$, $v = df$, and $\lambda = delta$, it determines $c_0 (= imsls_f_non_central_chi_sq_inv (p, df, delta))$, such that

$$P = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} \int_0^{c_0} \frac{x^{(v+2i)/2-1} e^{-x/2}}{2^{(v+2i)/2} \Gamma(\frac{v+2i}{2})} dx$$

where $\Gamma(\cdot)$ is the gamma function. In other words:

$$F^{-1}(P|df, \delta) = x$$

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 function `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>

int main()
{
    int df = 2;
    float p = .95, delta = 1.0, chi_squared;

    chi_squared = imsls_f_non_central_chi_sq_inv(p, df, delta);
    printf("The %4.2f noncentral chi-squared critical value is "
           "%6.4f.\n", 1.0-p, chi_squared);
}
```

Output

```
The 0.05 noncentral chi-squared critical value is 8.6422.
```

non_central_chi_sq_pdf

Evaluates the noncentral chi-squared probability density function.

Synopsis

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

```
float imsls_f_non_central_chi_sq_pdf (float x, float df, float lambda)
```

The type *double* function is `imsls_d_non_central_chi_sq_pdf`.

Required Arguments

float **x** (Input)

Argument for which the noncentral chi-squared probability density function is to be evaluated. **x** must be greater than or equal to 0.

float **df** (Input)

Number of degrees of freedom of the noncentral chi-squared distribution. **df** must be greater than 0.

float **lambda** (Input)

Noncentrality parameter. **lambda** must be greater than or equal to 0.

Return Value

The probability density associated with a noncentral chi-squared random variable with value **x**.

Description

The noncentral chi-squared distribution is a generalization of the chi-squared distribution. If $\{X_i\}$ are k independent, normally distributed random variables with means μ_i and variances σ_i^2 , then the random variable:

$$X = \sum_{i=1}^k \left(\frac{X_i}{\sigma_i} \right)^2$$

is distributed according to the noncentral chi-squared distribution. The noncentral chi-squared distribution has two parameters: k which specifies the number of degrees of freedom (i.e. the number of X_i), and λ which is related to the mean of the random variables X_i by:

$$\lambda = \sum_{i=1}^k \left(\frac{\mu_i}{\sigma_i} \right)^2$$

The noncentral chi-squared distribution is equivalent to a (central) chi-squared distribution with $k + 2i$ degrees of freedom, where i is the value of a Poisson distributed random variable with parameter $\lambda / 2$. Thus, the probability density function is given by:

$$F(x|k, \lambda) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} f(x, k + 2i)$$

where the (central) chi-squared PDF $f(x|k)$ is given by:

$$f(x|k) = \frac{(x/2)^{k/2} e^{-x/2}}{x \Gamma(k/2)} \text{ for } x > 0, \text{ else } 0$$

where $\Gamma(\cdot)$ is the gamma function. The above representation of $F(x|k, \lambda)$ can be shown to be equivalent to the representation:

$$F(x|k, \lambda) = \frac{e^{-(\lambda+x)/2} (x/2)^{k/2}}{x} \sum_{i=0}^{\infty} \phi_i$$

$$\phi_i = \frac{(\lambda x / 4)^i}{i! \Gamma(k/2 + i)}$$

Function `imsls_f_non_central_chi_sq_pdf` evaluates the probability density function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `lambda`, corresponding to $k = df$, $\lambda = lambda$, and $x = x$.

Function `imsls_f_non_central_chi_sq` evaluates the cumulative distribution function incorporating the above probability density function.

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the central chi-squared distribution.

Example

This example calculates the noncentral chi-squared distribution for a distribution with 100 degrees of freedom and noncentrality parameter $\lambda = 40$.

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

int main()
{
    int i;
    float x[] = {0, 8, 40, 136, 280, 400};
    float df = 100, lambda = 40.0, pdfv;

    printf ("\n\n df: %4.0f;  lambda: %4.0f\n\n",
            df, lambda);
    printf ("      x          pdf(x)\n");

    for (i=0; i<6; i++) {
        pdfv = imsls_f_non_central_chi_sq_pdf(x[i], df, lambda);
        printf (" %5.0f  %12.4e\n",x[i], pdfv);
    }
}
```

Output

```
df: 100; lambda:  40

      x          pdf(x)
    0  0.0000e+000
     8  4.7644e-044
    40  3.4621e-014
   136  2.1092e-002
   280  4.0027e-010
   400  1.1250e-022
```

exponential_cdf

Evaluates the exponential cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_exponential_cdf(float x, float b)
```

The type *double* function is `imsls_d_exponential_cdf`.

Required Arguments

float **x** (Input)

Argument for which the exponential CDF is to be evaluated. **x** must be non-negative.

float **b** (Input)

Scale parameter of the exponential CDF. **b** must be positive.

Return Value

The probability that an exponential random variable takes a value less than or equal to **x**. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_exponential_cdf` evaluates the exponential cumulative distribution function (CDF). This function is a special case of the gamma CDF

$$G(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-\frac{t}{b}} t^{a-1} dt$$

Setting $a=1$ and applying the scale parameter $b = \mathbf{b}$ yields the exponential CDF

$$F(x) = \int_0^x e^{-\frac{t}{b}} dt = 1 - e^{-\frac{x}{b}}$$

This relationship between the gamma and exponential CDFs is used by `imsls_f_exponential_cdf`.

Example

In this example, we evaluate the exponential CDF at $x = 2.0$, $b = 1.0$.

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

int main()
{
    float x = 2.0;
    float b = 1.0;
    float p;

    p = imsls_f_exponential_cdf(x,b);
    printf("The probability that exponential random ");
    printf("variable X with\nscale parameter b = ");
    printf("%3.1f is less than or equal to %3.1f", b, x);
    printf("\nis %6.4f\n\n", p);
}
```

Output

```
The probability that exponential random variable X with
scale parameter b = 1.0 is less than or equal to 2.0
is 0.8647
```

exponential_inverse_cdf

Evaluates the inverse of the exponential cumulative distribution function (CDF).

Synopsis

```
#include<imsls.h>

float imsls_f_exponential_inverse_cdf(float p, float b)
```

The type *double* function is `imsls_d_exponential_inverse_cdf`.

Required Arguments

- float***p** (Input)
Probability for which the inverse of the exponential CDF is to be evaluated. **p** must lie in the closed interval [0, 1].
- float***b** (Input)
Scale parameter of the exponential CDF. **b** must be positive.

Return Value

Function value, the value of the inverse of the exponential CDF. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_exponential_inverse_cdf(p, b)` evaluates $F^{-1}(p|b)$, the inverse CDF of an exponential random variable with probability argument $p = \mathbf{p}$ and scale parameter $b = \mathbf{b}$:

$$F^{-1}(p|b) = -b \log(1-p)$$

The probability that an exponential random variable takes a value less than or equal to the returned value is **p**.

Example

In this example, we evaluate the exponential inverse CDF at **p** = 0.8647, **b** = 1.0.:

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

int main()
{
    float p = 0.8647;
    float b = 1.0;
    float x;

    x = imsls_f_exponential_inverse_cdf(p, b);
    printf("The probability that exponential random ");
    printf("variable X with\nscale parameter b = ");
    printf("%3.1f is less than or equal to %6.4f", b, x);
    printf("\nis %6.4f\n\n", p);
}
```

Output

```
The probability that exponential random variable X with
scale parameter b = 1.0 is less than or equal to 2.0003
is 0.8647
```

exponential_pdf

Evaluates the exponential probability density function (PDF).

Synopsis

```
#include<imsls.h>

floatimsls_f_exponential_pdf(floatx, floatb)
```

The type *double* function is `imsls_d_exponential_pdf`.

Required Arguments

- floatx* (Input)
Argument for which the exponential PDF is to be evaluated. **x** must be non-negative.
- floatb* (Input)
Scale parameter of the exponential PDF. **b** must be positive.

Return Value

The value of the exponential probability density function with argument **x** and scale parameter **b**. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_exponential_pdf` evaluates the exponential probability density function. The exponential distribution is a special case of the gamma distribution and is defined as

$$f(x|b) = \Gamma(x|1,b) = \frac{1}{b}e^{-\frac{x}{b}}$$

Example

In this example, we evaluate the exponential PDF at **x** = 2.0, **b** = 1.0.

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

```
#include <stdio.h>

int main()
{
    float x = 2.0;
    float b = 1.0;
    float p;

    p = imsls_f_exponential_pdf(x,b);
    printf("The probability density of exponential ");
    printf("random variable X\nwith scale parameter b = ");
    printf("%3.1f and value x = %3.1f is %6.4f\n\n", b, x, p);
}
```

Output

```
The probability density of exponential random variable X
with scale parameter b = 1.0 and value x = 2.0 is 0.1353
```

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

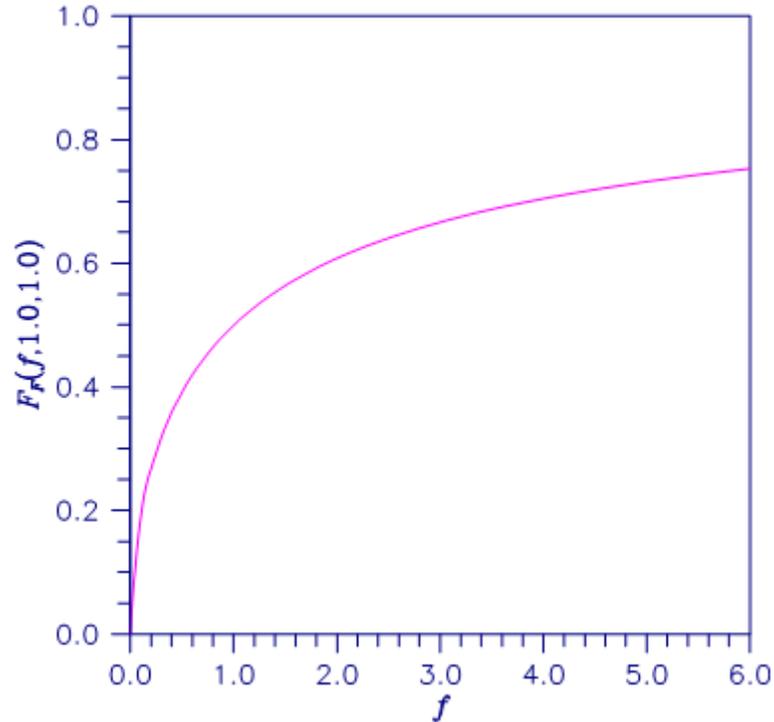
The probability that an F random variable takes a value less than or equal to the input point, `f`.

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 16, 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>
#include <stdio.h>

int main()
{
    float F = 648.0, df_numerator = 1.0;
    float df_denominator = 1.0, p;

    p = 1.0 - imsls_f_F_cdf(F, df_numerator, df_denominator);
    printf("The probability that an F(%1.0f, %1.0f) variate"
        "is greater than %3.0f is %6.4f.\n", df_numerator,
        df_denominator, F, 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 $\nu_1 = \text{df_numerator}$ numerator degrees of freedom and $\nu_2 = \text{df_denominator}$ 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

$Y = (\mathbf{v}_1 X) / (\mathbf{v}_2 + \mathbf{v}_1 X)$, then Y is a beta variate with parameters $p = \mathbf{v}_1 / 2$ and $q = \mathbf{v}_2 / 2$. 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:

$$F_F(f | v_1, v_2) = 1 - F_F(1/f | v_2, v_1)$$

Example

This example finds the 99-th percentage point for an F random variable with 7 and 1 degrees of freedom.

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

int main()
{
    float  df_denominator = 1.0, df_numerator = 7.0, p = 0.99, f;

    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

IMSLF_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.

complementary_F_cdf

Evaluates the complement of the F distribution function.

Synopsis

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

```
float imsls_f_complementary_F_cdf (float f, float df_numerator, float df_denominator)
```

The type *double* function is `imsls_d_complementary_F_cdf`.

Required Arguments

float f (Input)

Argument for which $\Pr(x > f)$ 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 greater than f .

Description

Function `imsls_f_complementary_F_cdf` evaluates one minus 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 \mathbf{v}_1 and \mathbf{v}_2 degrees of freedom and $Y = (\mathbf{v}_1 X) / (\mathbf{v}_2 + \mathbf{v}_1 X)$, then Y is a beta variate with parameters $p = \mathbf{v}_1 / 2$ and $q = \mathbf{v}_2 / 2$. Function `imsls_f_complementary_F_cdf` also uses a relationship between F random variables that can be expressed as

$$F_F(f|v_1, v_2) = F_F(1/f | v_2, v_1)$$

where F_F is the distribution function for an F random variable.

This function provides higher right tail accuracy for the F distribution.

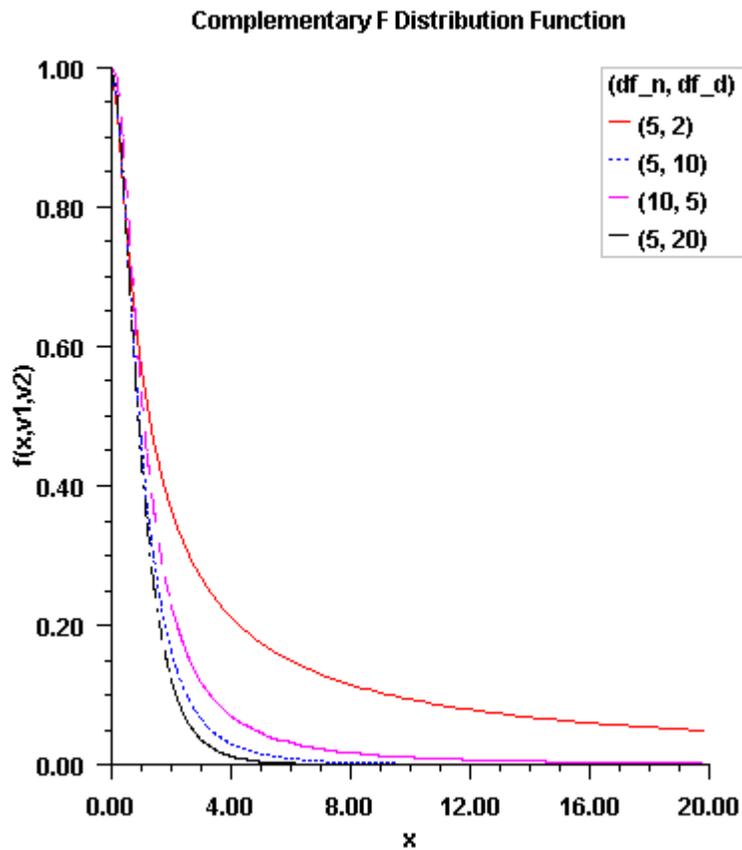


Figure 17, Plot of $F_F(f/df_n, df_d)$

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>
#include <stdio.h>
```

```
int main()
{
    float F = 648.0, df_numerator = 1.0, df_denominator = 1.0, p;

    p = imsls_f_complementary_F_cdf(F,df_numerator, df_denominator);
    printf("The probability that an F(%2.1f,%2.1f) variate is greater",
           df_numerator, df_denominator);
    printf(" than %5.1f is %6.4f.\n", F, p);
}
```

Output

```
The probability that an F(1.0,1.0) variate is greater than 648.0 is 0.0250.
```

non_central_F_pdf

Evaluates the noncentral F probability density function (PDF).

Synopsis

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

```
float imsls_f_non_central_F_pdf (float f, float df_numerator, float df_denominator, float  
    lambda)
```

The type *double* function is `imsls_d_non_central_F_pdf`.

Required Arguments

float f (Input)

Argument for which the noncentral F probability density function is to be evaluated. f must be non-negative.

float df_numerator (Input)

Numerator degrees of freedom of the noncentral F distribution. `df_numerator` must be positive.

float df_denominator (Input)

Denominator degrees of freedom of the noncentral F distribution. `df_denominator` must be positive.

float lambda (Input)

Noncentrality parameter. `lambda` must be non-negative.

Return Value

The probability density associated with a noncentral F random variable with value f .

Description

If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X , then

$$F = (X / v_1) / (Y / v_2)$$

is a noncentral F -distributed random variable whose PDF is given by

$$PDF(f | v_1, v_2, \lambda) = \Psi \sum_{k=0}^{\infty} \phi_k$$

where

$$\Psi = \frac{e^{-\lambda/2} (v_1 f)^{v_1/2} (v_2)^{v_2/2}}{f (v_1 f + v_2)^{(v_1+v_2)/2} \Gamma(v_2/2)}$$

$$\phi_k = \frac{R^k \Gamma\left(\frac{v_1+v_2}{2} + k\right)}{k! \Gamma\left(\frac{v_1}{2} + k\right)}$$

$$R = \frac{\lambda v_1 f}{2(v_1 f + v_2)}$$

and $\Gamma(\cdot)$ is the gamma function, $v_1 = \text{df_numerator}$, $v_2 = \text{df_denominator}$, $\lambda = \text{lambda}$, and $f = \text{f}$.

With a noncentrality parameter of zero, the noncentral F distribution is the same as the F distribution.

The efficiency of the calculation of the above series is enhanced by:

1. calculating each term ϕ_k in the series recursively in terms of either the term ϕ_{k-1} preceding it or the term ϕ_{k+1} following it, and.
2. initializing the sum with the largest series term and adding the subsequent terms in order of decreasing magnitude

Special cases:

For $R = \lambda f = 0$:

$$PDF(f|v_1, v_2, \lambda) = \Psi \Phi_0 = \Psi \frac{\Gamma([v_1 + v_2]/2)}{\Gamma(v_1/2)}$$

For $\lambda = 0$:

$$PDF(f|v_1, v_2, \lambda) = \frac{(v_1 f)^{v_1/2} (v_2)^{v_2/2} \Gamma([v_1 + v_2]/2)}{f^{(v_1 f + v_2)} (v_1 + v_2)^{v_1/2} \Gamma(v_1/2) \Gamma(v_2/2)}$$

For $f = 0$:

$$PDF(f|v_1, v_2, \lambda) = \frac{e^{-\lambda/2} f^{v_1/2-1} (v_1/v_2)^{v_1/2} \Gamma([v_1 + v_2]/2)}{\Gamma(v_1/2) \Gamma(v_2/2)} = \begin{cases} 0 & \text{if } v_1 > 2; \\ e^{-\lambda/2} & \text{if } v_1 = 2 \\ \infty & \text{if } v_1 < 2 \end{cases}$$

Example

This example traces out a portion of a noncentral F distribution with parameters `df_numerator = 100`, `df_denominator = 10`, and `lambda = 10`.

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

int main()
{
    int i;
    float f[] = {0., .4, .8, 3.2, 5.6, 8.8, 14., 18.};
    float df_numerator = 100., df_denominator = 10., lambda = 10., pdfv;

    printf ("df_numerator:  %4.0f\n", df_numerator);
    printf ("df_denominator: %4.0f\n", df_denominator);
    printf ("lambda:       %4.0f\n\n", lambda);
    printf ("      f          pdf(f)\n");

    for (i=0; i<8; i++) {
        pdfv = imsls_f_non_central_F_pdf
            (f[i], df_numerator, df_denominator, lambda);
        printf (" %5.1f %12.4e\n", f[i], pdfv);
    }
}
```

Output

```
df_numerator:  100
df_denominator:  10
lambda:       10

      f          pdf(f)
0.0  0.0000e+000
0.4  9.7488e-002
```

0.8	8.1312e-001
3.2	3.6948e-002
5.6	2.8302e-003
8.8	2.7661e-004
14.0	2.1963e-005
18.0	5.3483e-006

non_central_F_cdf

Evaluates the noncentral F cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_non_central_F_cdf (float f, float df_numerator, float df_denominator,  
                                float lambda)
```

The type *double* function is `imsls_d_non_central_F_cdf`.

Required Arguments

float f (Input)

Argument for which the noncentral F cumulative distribution function is to be evaluated. f must be non-negative.

float df_numerator (Input)

Numerator degrees of freedom of the noncentral F distribution. `df_numerator` must be positive.

float df_denominator (Input)

Denominator degrees of freedom of the noncentral F distribution. `df_denominator` must be positive.

float lambda (Input)

Noncentrality parameter. `lambda` must be non-negative.

Return Value

The probability that a noncentral F random variable takes a value less than or equal to f .

Description

If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X , then

$$F = (X/v_1) / (Y/v_2)$$

is a noncentral F -distributed random variable whose CDF is given by:

$$CDF(f|v_1, v_2, \lambda) = \sum_{j=0}^{\infty} c_j$$

where:

$$c_j = \omega_j I_x\left(\frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

$$\omega_j = e^{-\lambda/2} (\lambda/2)^j / j! = \frac{\lambda}{2j} \omega_{j-1}$$

$$I_x(a, b) \equiv \text{incomplete beta function ratio} \equiv \frac{B_x(a, b)}{B(a, b)}$$

$$B_x(a, b) \equiv \text{incomplete beta function} \equiv \int_0^x t^{a-1} (1-t)^{b-1} dt$$

$$= x^a \sum_{j=0}^{\infty} \frac{\Gamma(j+1-b)}{(a+j)\Gamma(1-b)j!} x^j$$

$$x = \frac{v_1 f}{(v_2 + v_1 f)} \quad < = = > \quad f = \frac{v_2 x}{v_1(1-x)}$$

$$B(a, b) = B_1(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

$$I_x(a+1, b) = I_x(a, b) - T_x(a, b)$$

$$T_x(a, b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\Gamma(b)} x^a (1-x)^b = T_x(a-1, b) \frac{a-1+b}{a} x$$

and $\Gamma(\cdot)$ is the gamma function. The above series expansion for the noncentral F CDF was taken from Butler and Paolella (1999) (see Paolella.pdf), with the correction for the recursion relation given below:

$$I_x(a+1, b) = I_x(a, b) - T_x(a, b)$$

extracted from the AS 63 algorithm for calculating the incomplete beta function as described by Majumder and Bhattacharjee (1973).

The correspondence between the arguments of function `imsls_f_non_central_F_cdf` and the variables in the above equations is as follows: $v_1 = \text{df_numerator}$, $v_2 = \text{df_denominator}$, $\lambda = \text{lambda}$, and $f = f$.

For $\lambda = 0$, the noncentral F distribution is the same as the F distribution.

Example

This example traces out a portion of a noncentral F cumulative distribution function with parameters `df_numerator = 100`, `df_denominator = 10`, and `lambda = 10`.

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

int main()
{
    int i;
    float f[] = {0., .4, .8, 1.2, 1.6, 2.0, 2.8, 4.0};
    float df_numerator = 100., df_denominator = 10., lambda = 10., cdfv;

    printf ("\n df_numerator:   %4.0f\n", df_numerator);
    printf (" df_denominator: %4.0f\n", df_denominator);
    printf (" lambda:             %4.0f\n\n", lambda);
    printf ("      f           CDF(f)\n\n");
    for (i=0; i<8; i++) {
        cdfv = imsls_f_non_central_F_cdf
            (f[i], df_numerator, df_denominator, lambda);
        printf (" %5.1f  %12.4e \n", f[i], cdfv);
    }
}
```

Output

```
df_numerator:   100
df_denominator:  10
lambda:          10

      f           cdf(f)
0.0  0.0000e+000
0.4  4.8879e-003
0.8  2.0263e-001
1.2  5.2114e-001
1.6  7.3385e-001
2.0  8.5041e-001
2.8  9.4713e-001
4.0  9.8536e-001
```

complementary_non_central_F_cdf

Evaluates the complementary noncentral F cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_complementary_non_central_F_cdf (float f, float df_numerator,  
float df_denominator, float lambda)
```

The type *double* function is `imsls_d_complementary_non_central_F_cdf`.

Required Arguments

float `f` (Input)

Argument for which the complementary noncentral F cumulative distribution function is to be evaluated. `f` must be non-negative.

float `df_numerator` (Input)

Numerator degrees of freedom of the complementary noncentral F distribution. `df_numerator` must be positive.

float `df_denominator` (Input)

Denominator degrees of freedom of the complementary noncentral F distribution. `df_denominator` must be positive.

float `lambda` (Input)

Noncentrality parameter. `lambda` must be non-negative.

Return Value

The probability that a noncentral F random variable takes a value greater than `f`.

Description

If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X , then

$$F = (X/v_1) / (Y/v_2)$$

is a noncentral F -distributed random variable whose CDF is given by:

$$F(f|v_1, v_2, \lambda) = \sum_{j=0}^{\infty} c_j$$

where:

$$c_j = \omega_j I_x\left(\frac{v_1}{2} + j, \frac{v_2}{2}\right)$$

$$\omega_j = e^{-\lambda/2} (\lambda/2)^j / j! = \frac{\lambda}{2j} \omega_{j-1}$$

$$I_x(a, b) \equiv \text{incomplete beta function ratio} \equiv \frac{B_x(a, b)}{B(a, b)}$$

$$B_x(a, b) \equiv \text{incomplete beta function} \equiv \int_0^x t^{a-1} (1-t)^{b-1} dt$$

$$= x^a \sum_{j=0}^{\infty} \frac{\Gamma(j+1-b)}{(a+j)\Gamma(1-b)j!} x^j$$

$$x = \frac{v_1 f}{(v_2 + v_1 f)} \quad < = = > \quad f = \frac{v_2 x}{v_1(1-x)}$$

$$B(a, b) = B_1(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

$$I_x(a+1, b) = I_x(a, b) - T_x(a, b)$$

$$T_x(a, b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\Gamma(b)} x^a (1-x)^b = T_x(a-1, b) \frac{a-1+b}{a} x$$

and $\Gamma(\cdot)$ is the gamma function. The above series expansion for the noncentral F CDF, denoted by $F(\cdot)$, was taken from Butler and Paoletta (1999) (see [Paoletta.pdf](#)), with the correction for the recursion relation given below:

$$I_x(a+1, b) = I_x(a, b) - T_x(a, b)$$

extracted from the AS 63 algorithm for calculating the incomplete beta function as described by Majumder and Bhattacharjee (1973).

The series approximation of the complementary (cmp) noncentral F CDF, denoted by $F(\cdot)$, is obtainable by using the following identities:

$$\sum_{j=0}^{\infty} \omega_j = 1$$

$$I_{1-x}(b, a) = 1 - I_x(a, b)$$

$$I_{1-x}(b, a+1) = 1 - I_x(a+1, b) = 1 - I_x(a, b) + T_x(a, b) = I_{1-x}(b, a) + T_x(a, b)$$

Thus:

$$\begin{aligned} \bar{F}(f|v_1, v_2, \lambda) &= 1 - \sum_{j=0}^{\infty} c_j = \sum_{j=0}^{\infty} \omega_j [1 - I_x(\frac{v_1}{2} + j, \frac{v_2}{2})] \\ &= \sum_{j=0}^{\infty} \omega_j I_{1-x}(\frac{v_2}{2}, \frac{v_1}{2} + j) \end{aligned}$$

The correspondence between the arguments of function `imsls_f_complementary_non_central_F_cdf` and the variables in the above equations is as follows: $v_1 = \text{df_numerator}$, $v_2 = \text{df_denominator}$, $\lambda = \text{lambda}$, and $f = f$.

Also, we can use the above expansion of $\bar{F}(f|v_1, v_2, \lambda)$ and the identities:

$$I_{1-x}(b, a+1) = I_{1-x}(b, a) + T_x(a, b)$$

$$T_x(a, b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\Gamma(b)} x^a (1-x)^b = T_x(a-1, b) \frac{a-1+b}{a} x$$

to recursively calculate $\bar{F}(f|v_1, v_2, \lambda)$.

For $\lambda = 0$, the noncentral F distribution is the same as the F distribution.

Example

This example traces out a portion of a complementary noncentral F cumulative distribution function with parameters `df_numerator = 100`, `df_denominator = 10`, and `lambda = 10`.

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

int main()
{
    int i;
    float f[] = {0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.8, 4.0};
    float df_numerator = 100.0, df_denominator = 10.0;
    float lambda = 10.0, cmpcdfv;

    printf ("\n df_numerator:   %4.0f\n", df_numerator);
    printf (" df_denominator: %4.0f\n", df_denominator);
    printf (" lambda:             %4.0f\n\n", lambda);
    printf ("      f      cmpCDF(f)\n\n");
    for (i=0; i<8; i++) {
        cmpcdfv = imsls_f_complementary_non_central_F_cdf
            (f[i], df_numerator, df_denominator, lambda);
        printf (" %5.1f %12.4e \n", f[i], cmpcdfv);
    }
}
```

Output

```
df_numerator:   100
df_denominator:  10
lambda:          10
```

f	cmpCDF(f)
0.0	1.0000e+000
0.4	9.9511e-001
0.8	7.9737e-001
1.2	4.7886e-001
1.6	2.6615e-001
2.0	1.4959e-001
2.8	5.2875e-002
4.0	1.4642e-002

non_central_F_inverse_cdf

Evaluates the inverse of the noncentral F cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_non_central_F_inverse_cdf (float p, float df_numerator,  
    float df_denominator float lambda)
```

The type *double* function is `imsls_d_non_central_F_inverse_cdf`.

Required Arguments

float `p` (Input)

Probability for which the inverse of the noncentral F cumulative distribution function is to be evaluated. `p` must be non-negative and less than one.

float `df_numerator` (Input)

Numerator degrees of freedom of the noncentral F distribution. `df_numerator` must be positive.

float `df_denominator` (Input)

Denominator degrees of freedom of the noncentral F distribution. `df_denominator` must be positive.

float `lambda` (Input)

Noncentrality parameter. `lambda` must be non-negative.

Return Value

The inverse of the noncentral F distribution function evaluated at `p`. The probability that a noncentral F random variable takes a value less than or equal to `imsls_f_non_central_F_inverse_cdf` is `p`.

Description

If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom that is statistically independent of X , then

$$F = (X/v_1) / (Y/v_2)$$

is a noncentral F -distributed random variable whose cumulative distribution function $\rho = CDF(f, v_1, v_2, \lambda)$ is defined as the probability ρ that $F \leq f$ and is evaluated using function `imsls_f_non_central_F_cdf` (`f`, `df_numerator`, `df_denominator`, `lambda`), where $v_1 = \text{df_numerator}$, $v_2 = \text{df_denominator}$, $\lambda = \text{lambda}$, and $\rho = p$.

Function `imsls_f_non_central_F_inverse_cdf` evaluates

$$f = CDF^{-1}(p|v_1, v_2, \lambda)$$

Function `imsls_f_non_central_F_inverse_cdf` uses bisection and modified regula falsi search algorithms to invert the distribution function $CDF(f|v_1, v_2, \lambda)$. For sufficiently small p , an accurate approximation of $CDF^{-1}(p|v_1, v_2, \lambda)$ can be used which requires no such inverse search algorithms.

Example

This example traces out a portion of a noncentral F cumulative distribution function with parameters `df_numerator = 100`, `df_denominator = 10`, and `lambda = 10` and for each value of `f` prints `f`, `p=CDF(f)`, and `CDF-1(p)`.

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

int main()
{
    int i;
    float f[] = {0., .4, .8, 1.2, 1.6, 2.0, 2.8, 4.0};
    float df_numerator = 100.0, df_denominator = 10.0;
    float lambda = 10.0, cdfv, cdfiv;

    printf ("\n df_numerator:   %4.0f\n", df_numerator);
    printf (" df_denominator: %4.0f\n", df_denominator);
    printf (" lambda:                %4.0f\n\n", lambda);
    printf ("      f      p = cdf(f)   cdfinv(p)\n\n");

    for (i=0; i<8; i++) {
        cdfv = imsls_f_non_central_F_cdf
            (f[i], df_numerator, df_denominator, lambda);
        cdfiv = imsls_f_non_central_F_inverse_cdf
            (cdfv, df_numerator, df_denominator, lambda);
        printf (" %5.1f %12.4e %7.3f\n", f[i], cdfv, cdfiv);
    }
}
```

Output

```
df_numerator: 100
df_denominator: 10
lambda: 10
```

f	p = cdf(f)	cdfinv(p)
0.0	0.0000e+000	0.000
0.4	4.8879e-003	0.400
0.8	2.0263e-001	0.800
1.2	5.2114e-001	1.200
1.6	7.3385e-001	1.600
2.0	8.5041e-001	2.000
2.8	9.4713e-001	2.800
4.0	9.8536e-001	4.000

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 $\Gamma(\cdot)$ 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:

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (t-c)^{a-1}$$

If T is a random variable with parameters a , b , and c , the probability that $T \leq t_0$ 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>
#include <stdio.h>

int main()
{
    float x = 0.5, a = 4.0, p;

    p = imsls_f_gamma_cdf(x,a);
    printf("The probability that X is less than "
           "%3.1f is %6.4f\n", x, p);

    x = 1.0;
    p = imsls_f_gamma_cdf(x,a) - p;
    printf("The probability that X is between 0.5 and "
           "%3.1f is %6.4f\n", x, p);
}
```

Output

```
The probability that X is less than 0.5 is 0.0018
The probability that X is between 0.5 and 1.0 is 0.0172
```

Informational Errors

IMSL_ARG_LESS_THAN_ZERO

Since $x = \#$ is less than zero, the distribution function is zero at x .

Fatal Errors

IMSL_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. In other words:

$$F^{-1}(P|a) = x$$

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>
#include <stdio.h>

int main()
{
    float p = .95, a = 4.0, x;

    x = imsls_f_gamma_inverse_cdf(p,a);
    printf("The %4.2f gamma(%1.0f) critical value is %6.4f\n",
        1.0 - p, a, x);
}
```

Output

```
The 0.05 gamma(4) critical value is 7.7537
```

lognormal_cdf

Evaluates the lognormal cumulative distribution function (CDF).

Synopsis

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

```
floatimsls_f_lognormal_cdf(float x, float amu, float sigma)
```

The type *double* function is `imsls_d_lognormal_cdf`.

Required Arguments

*float***x** (Input)

Argument for which the lognormal CDF is to be evaluated. **x** must be non-negative.

*float***amu** (Input)

Location parameter of the lognormal CDF.

*float***sigma** (Input)

Shape parameter of the lognormal CDF. **sigma** must be positive.

Return Value

The probability that a lognormal random variable takes a value less than or equal to **x**. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_lognormal_cdf` evaluates the lognormal cumulative distribution function (CDF), defined as

$$F(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{1}{t} e^{-\frac{1}{2}\left(\frac{\log(t)-\mu}{\sigma}\right)^2} dt = \Phi\left(\frac{\log(x)-\mu}{\sigma}\right)$$

where

$$\phi(y) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^y e^{-\frac{1}{2}u^2} du$$

is the standard normal CDF.

Example

In this example, we evaluate the *CDF* at $x = 0.7137$, $\text{amu} = 0.0$, $\text{sigma} = 0.5$.

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

int main()
{
    float x = 0.7137;
    float amu = 0.0;
    float sigma = 0.5;
    float p;

    p = imsls_f_lognormal_cdf(x,amu,sigma);
    printf("The probability that lognormal random ");
    printf("variable X\n");
    printf("with location parameter amu = %3.1f ", amu);
    printf("and shape parameter\nsigma = %3.1f ", sigma);
    printf("is less than or equal to ");
    printf("%6.4f is %6.4f\n\n", x, p);
}
```

Output

```
The probability that lognormal random varisable X
with location parameter amu = 0.0 and shape parameter
sigma = 0.5 is less than or equal to 0.7137 is 0.2500
```

lognormal_inverse_cdf

Evaluates the inverse of the lognormal cumulative distribution function (CDF).

Synopsis

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

```
float imsls_f_lognormal_inverse_cdf(float p, float amu, float sigma)
```

The type *double* function is `imsls_d_lognormal_inverse_cdf`.

Required Arguments

*float***p** (Input)

Probability for which the inverse of the lognormal CDF is to be evaluated. **p** must lie in the closed interval [0, 1].

*float***amu** (Input)

Location parameter of the lognormal CDF.

*float***sigma** (Input)

Shape parameter of the lognormal CDF. **sigma** must be positive.

Return Value

Function value, the probability that a lognormal random variable takes a value less than or equal to the returned value is the input probability **p**. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_lognormal_inverse_cdf` evaluates the inverse CDF of a lognormal random variable with location parameter **amu** and scale parameter **sigma**. The probability that a standard lognormal random variable takes a value less than or equal to the returned value is **p** ($p=P$).

$$P = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x \frac{e^{-\frac{1}{2}\left(\frac{\log(t)-\mu}{\sigma}\right)^2}}{t} dt = \varphi\left(\frac{\log(x)-\mu}{\sigma}\right)$$

where

$$\varphi(y) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^y e^{-\frac{1}{2}u^2} du$$

In other words

$$F^{-1} = (P|\mu,\sigma) = x$$

Example

In this example, we evaluate the inverse CDF at $p = 0.25$, $\text{amu} = 0.0$, $\text{sigma} = 0.5$.

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

int main()
{
    float p = 0.25;
    float amu = 0.0;
    float sigma = 0.5;
    float x;

    x = imsls_f_lognormal_inverse_cdf(p, amu, sigma);
    printf("The probability that lognormal random ");
    printf("variable X\n");
    printf("with location parameter amu = %3.1f ", amu);
    printf("and shape parameter\nsigma = %3.1f ", sigma);
    printf("is less than or equal to ");
    printf("%6.4f is %4.2f\n\n", x, p);
}
```

Output

```
The probability that lognormal random variable X
with location parameter amu = 0.0 and shape parameter
sigma = 0.5 is less than or equal to 0.7137 is 0.25
```

lognormal_pdf

Evaluates the lognormal probability density function (*PDF*).

Synopsis

```
#include<imsls.h>

float imsls_f_lognormal_pdf(float x, float amu, float sigma)
```

The type *double* function is `imsls_d_lognormal_pdf`.

Required Arguments

- float***x** (Input)
Argument for which the lognormal PDF is to be evaluated. **x** must be non-negative.
- float***amu** (Input)
Location parameter of the lognormal PDF.
- float***sigma** (Input)
Shape parameter of the lognormal PDF. **sigma** must be positive.

Return Value

The probability density of a lognormally distributed random variable with value **x**, location parameter **amu**, and shape parameter **sigma**. A value of NaN is returned if an input value is in error.

Description

The function `imsls_f_lognormal_pdf` evaluates the lognormal probability density function (PDF), defined as

$$f(x|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{[\log(x)-\mu]^2}{2\sigma^2}}$$

Example

In this example, we evaluate the PDF at $x = 1.0$, $\text{amu} = 0.0$, $\text{sigma} = 0.5$.

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

int main()
{
    float x = 1.0;
    float amu = 0.0;
    float sigma = 0.5;
    float pdfv;

    pdfv = imsls_f_lognormal_pdf(x, amu, sigma);
    printf("The probability density of lognormal random ");
    printf("variable X\n");
    printf("with location parameter amu = %3.1f, ", amu);
    printf("shape parameter\nsigma = %3.1f, ", sigma);
    printf("and value x = %3.1f is %6.4f\n\n", x, pdfv);
}
```

Output

```
The probability density of lognormal random variable X
with location parameter amu = 0.0, shape parameter
sigma = 0.5, and value x = 1.0 is 0.7979
```

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**.

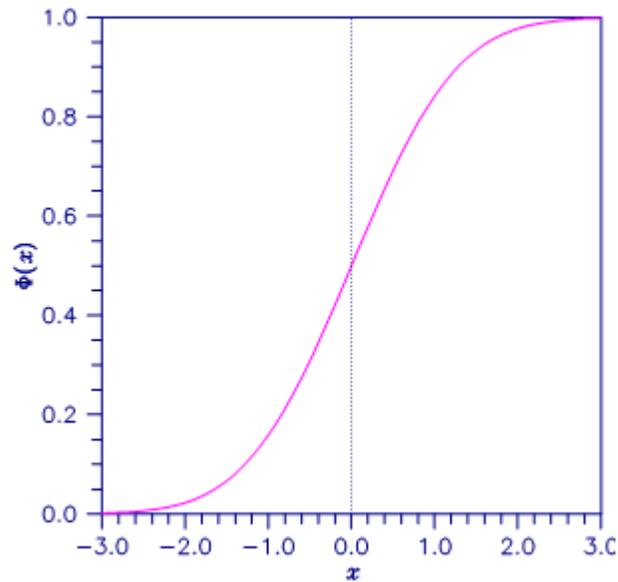
Description

Function `imsls_f_normal_cdf` evaluates the distribution function, $F(x)$, of a standard normal (Gaussian) random variable as follows:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt$$

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 18, Plot of $F(x)$

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>
#include <stdio.h>

int 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
```

multivariate_normal_cdf



[more...](#)

Evaluates the cumulative distribution function for the multivariate normal distribution.

Synopsis

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

```
float imsls_f_multivariate_normal_cdf (int k, float h[ ], float mean[ ], float sigma[ ], ...0)
```

The type *double* function is `imsls_d_multivariate_normal_cdf`.

Required Arguments

int **k** (Input)

The number of variates in the multivariate normal distribution. The number of variates must be greater than or equal to 1 and less than or equal to 1100.

float **h[]** (Input)

Array of length **k** containing the upper bounds for calculating the cumulative distribution function, $F(X_1 < h_1, X_2 < h_2, \dots, X_k < h_k)$.

float **mean[]** (Input)

Array of length **k** containing the mean of the multivariate normal distribution, i.e., $E(x_1, x_2, \dots, x_k) = \mu^T = (\mu_1, \mu_2, \dots, \mu_k)$.

float **sigma[]** (Input)

Array of length **k** by **k** containing the positive definite symmetric variance-covariance matrix for the multivariate normal distribution, i.e., $Var(x_1, x_2, \dots, x_k) = \Sigma$.

Return Value

The value of the cumulative distribution function for a multivariate normal random variable,

$$F(X_1 < h_1, X_2 < h_2, \dots, X_k < h_k).$$

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_multivariate_normal_cdf (int k, float h[ ], float mean[ ], float sigma[ ],
    IMSLS_PRINT,
    IMSLS_ERR_ABS, float err_abs,
    IMSLS_ERR_REL, float err_rel,
    IMSLS_TOLERANCE, float tolerance,
    IMSLS_MAX_EVALS, int max_evals,
    IMSLS_RANDOM_SEED, int random_seed,
    IMSLS_ERR_EST, float *err_est,
    0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

Print intermediate computations.

Default: No printing.

`IMSLS_ERR_ABS`, `float err_abs` (Input)

The absolute accuracy requested for the calculated cumulative probability.

Default: `err_abs` = $1.0e^{-3}$.

`IMSLS_ERR_REL`, `float err_rel` (Input)

The relative accuracy desired.

Default: `err_rel` = $1.0e^{-5}$.

`IMSLS_TOLERANCE`, `float tolerance` (Input)

The minimum value for the smallest eigenvalue of `sigma`. If the smallest eigenvalue is less than `tolerance`, then the terminal error `IMSLS_SIGMA_SINGULAR` is issued. Default: `tolerance` = ϵ , where ϵ is the machine precision.

`IMSLS_MAX_EVALS`, `int max_evals` (Input)

The maximum number of function evaluations allowed. If this limit is exceeded, the `IMSLS_MAX_EVALS_EXCEEDED` warning message is issued and the optimization terminates.

Default: `max_evals = 1000×k`.

`IMSL_RANDOM_SEED`, *int* `random_seed` (Input)

The value of the random seed used for generating quadrature points. By using different seeds on different calls to this function, slightly different answers will be obtained in the digits beyond the estimated error. If `random_seed = 0`, then the seed is set to the value of the system clock which will automatically generate slightly different answers for each call.

Default: `random_seed = 7919`.

`IMSL_ERR_EST`, *float* `*err_est` (Output)

The estimated error.

Description

Function `imsls_f_multivariate_normal_cdf` evaluates the cumulative distribution function F of a multivariate normal distribution with $E(X_1, X_2, \dots, X_k) = \boldsymbol{\mu}$ and $Var(X_1, X_2, \dots, X_k) = \Sigma$. The input arrays `mean` and `sigma` are used as the values for $\boldsymbol{\mu}$ and Σ , respectively. The formula for the CDF calculation is given by the multiple integral described in Johnson and Kotz (1972):

$$F(h_1, h_2, \dots, h_k) = (2\pi)^{-k/2} |\Sigma|^{-1/2} \int_{-\infty}^{h_1} \int_{-\infty}^{h_2} \dots \int_{-\infty}^{h_k} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} dx_k dx_{k-1} \dots dx_1$$

Σ must be positive definite, i.e. $|\Sigma| > 0$.

In the special case of non-negative equal correlations (i.e. $Cov(X_m, X_n) = \rho \geq 0, m \neq n$), the above integral is transformed into a univariate integral using the transformation developed by Dunnett and Sobel (1955). This produces very accurate and fast calculations even for a large number of variates.

If $k > 2$ and the correlations are not equal or both equal and negative, the Cholesky decomposition transformation described by Genz (1992) is used (with permission from the author). This transforms the problem into a definite integral involving $k-1$ variables which is solved numerically using randomized Korobov rules if $k \leq 100$, see Cranley and Patterson (1976) and Keast (1973); otherwise, the integral is solved using quasi-random Richtmeyer points described in Davis and Rabinowitz (1984).

Setting $\sigma_i = Var(X_i)$ and denoting the correlation matrix related to Σ by $W = (\rho_{mn})$, where

$$\rho_{mn} = Cov(X_m, X_n) / (\sqrt{\sigma_m} \sqrt{\sigma_n}),$$

an integral transformation transforms $F(h_1, h_2, \dots, h_k)$ into the standardized k -variate normal distribution with correlation matrix W :

$$\Phi\left(\frac{h_1 - \mu_1}{\sqrt{\sigma_1}}, \dots, \frac{h_k - \mu_k}{\sqrt{\sigma_k}}\right) = (2\pi)^{-k/2} |W|^{-1/2} \int_{-\infty}^{(h_1 - \mu_1)/\sqrt{\sigma_1}} \dots \int_{-\infty}^{(h_k - \mu_k)/\sqrt{\sigma_k}} e^{-\frac{1}{2}u^T W^{-1}u} du_k \dots du_1.$$

Therefore, it's also possible to compute the integral with the correlation matrix W defined in argument `sigma` if in addition the variances $\sigma_1, \dots, \sigma_k$ are known: If, with respect to the variance-covariance matrix, the bounds of the integral are h_1, \dots, h_k and the means are μ_1, \dots, μ_k , set required argument `h` to `h[i] = (h_{i+1} - \mu_{i+1})/\sqrt{\sigma_{i+1}}` and required argument `mean` to `mean[i] = 0` for $i = 0, \dots, k-1$.

Examples

Example 1

This example evaluates the cumulative distribution function for a trivariate normal random variable. There are three calculations. The first calculation is of $F(1.96, 1.96, 1.96)$ for a trivariate normal variable with $\mu = \{0, 0, 0\}$, and

$$\Sigma = \begin{bmatrix} 1 & 0.9 & 0.9 \\ 0.9 & 1 & 0.9 \\ 0.9 & 0.9 & 1 \end{bmatrix}$$

In this case, `imsls_f_multivariate_normal_cdf` calculates $F(1.96, 1.96, 1.96) = 0.958179$.

The second calculation involves a trivariate variable with the same correlation matrix as the first calculation but with a mean of $\mu = \{0, 1, -1\}$. This is the same distribution as the first example shifted by the mean. The calculation of $F(1.96, 2.96, 0.96)$ verifies that this probability is equal to the same value as reported for the first case.

The last calculation is the same calculation reported in Genz (1992) for a trivariate normal random variable with $\mu = \{0, 0, 0\}$ and

$$\Sigma = \begin{bmatrix} 1 & 3/5 & 1/3 \\ 3/5 & 1 & 11/15 \\ 1/3 & 11/15 & 1 \end{bmatrix}$$

In this example the calculation of $F(1, 4, 2) = 0.827985$.

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

int main()
{
    float bounds1[3] = {1.96, 1.96, 1.96};
    float bounds2[3] = {1.96, 2.96, 0.96};
    float bounds3[3] = {1.0, 4.0, 2.0};
    float mean1[3]   = {0.0, 0.0, 0.0};
    float mean2[3]   = {0.0, 1.0, -1.0};
    float stdev1[9]  = {1.0, 0.9, 0.9,
                       0.9, 1.0, 0.9,
                       0.9, 0.9, 1.0};
```

```

float stdev2[9] = {1.0, 0.6, 1.0/3.0,
                  0.6, 1.0, 11.0/15.0,
                  1.0/3.0, 11.0/15.0, 1.0};
float f;
char *fmt = {"%5.3W"};
imsls_f_write_matrix("Mean Vector", 1, 3, mean1,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
imsls_f_write_matrix("Correlation Matrix", 3, 3,
                    stdev1, IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
f = imsls_f_multivariate_normal_cdf(3, bounds1, mean1,
stdev1, 0);
printf("\nF(X1<%f, X2<%f, X3<%f) = %f\n\n",
      bounds1[0], bounds1[1], bounds1[2], f);

imsls_f_write_matrix("Mean Vector\n", 1, 3, mean2,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
imsls_f_write_matrix("Correlation Matrix", 3, 3,
                    stdev1, IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
f = imsls_f_multivariate_normal_cdf(3, bounds2, mean2,
stdev1, 0);
printf("\nF(X1<%f, X2<%f, X3<%f) = %f\n",
      bounds2[0], bounds2[1], bounds2[2], f);

imsls_f_write_matrix("Mean Vector", 1, 3, mean1,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
imsls_f_write_matrix("Correlation Matrix", 3, 3, stdev2,
                    IMSLS_NO_ROW_LABELS,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
f = imsls_f_multivariate_normal_cdf(3, bounds3, mean1,
stdev2, 0);
printf("\nF(X1<%f, X2<%f, X3<%f) = %f\n",
      bounds3[0], bounds3[1], bounds3[2], f);
}

```

Output

```

Mean Vector
0    0    0

Correlation Matrix
1.0  0.9  0.9

```

0.9	1.0	0.9
0.9	0.9	1.0
F(X1<1.960000, X2<1.960000, X3<1.960000) = 0.958179		
Mean Vector		
0	1	-1
Correlation Matrix		
1.0	0.9	0.9
0.9	1.0	0.9
0.9	0.9	1.0
F(X1<1.960000, X2<2.960000, X3<0.960000) = 0.958179		
Mean Vector		
0	0	0
Correlation Matrix		
1.00	0.60	0.33
0.60	1.00	0.73
0.33	0.73	1.00
F(X1<1.000000, X2<4.000000, X3<2.000000) = 0.827985		

Example 2

This example illustrates the calculation of the cdf for a multivariate normal distribution with a mean of $\boldsymbol{\mu} = \{1, 0, -1, 0, 1, -1\}$, and a correlation matrix of

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0.1 & 0.2 & 0.3 & 0.4 & 0 \\ 0.1 & 1 & 0.6 & 0.1 & 0.2 & 0.5 \\ 0.2 & 0.6 & 1 & 0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 0 & 1 & 0 & 0.5 \\ 0.4 & 0.2 & 0.1 & 0 & 1 & 0.3 \\ 0 & 0.5 & 0.2 & 0.5 & 0.3 & 1 \end{bmatrix}$$

The optional argument `IMSL_PRINT` is used to illustrate the type of intermediate output available from this function. This function sorts the variables by the limits for the cdf calculation specified in `x`. This improves the accuracy of the calculations, see Genz (1992). In this case, $F(X_1 < 1, X_2 < 2.5, X_3 < 2, X_4 < 0.5, X_5 < 0, X_6 < 0.8) = 0.087237$ with an estimated error of $8.7e-05$.

By increasing the correlation of X_2 and X_3 from 0.1 to 0.7, the correlation matrix becomes singular. This function checks for this condition and issues an error when `sigma` is not symmetric or positive definite.

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

```

int main()
{
    float bounds[6] = {1.0, 2.5, 2.0, 0.5, 0.0, 0.8};
    float mean[6]   = {1.0, 0.0, -1.0, 0.0, 1.0, -1.0};
    float s1[6*6]  = {1.0, 0.1, 0.2, 0.3, 0.4, 0.0,
                     0.1, 1.0, 0.6, 0.1, 0.2, 0.5,
                     0.2, 0.6, 1.0, 0.0, 0.1, 0.2,
                     0.3, 0.1, 0.0, 1.0, 0.0, 0.5,
                     0.4, 0.2, 0.1, 0.0, 1.0, 0.3,
                     0.0, 0.5, 0.2, 0.5, 0.3, 1.0};
    /* The following matrix is not positive definite */
    float s2[6*6] = {1.0, 0.1, 0.2, 0.3, 0.4, 0.0,
                    0.1, 1.0, 0.6, 0.7, 0.2, 0.5,
                    0.2, 0.6, 1.0, 0.0, 0.1, 0.2,
                    0.3, 0.7, 0.0, 1.0, 0.0, 0.5,
                    0.4, 0.2, 0.1, 0.0, 1.0, 0.3,
                    0.0, 0.5, 0.2, 0.5, 0.3, 1.0};

    float f, err;
    int i, k=6;
    f = imsls_f_multivariate_normal_cdf(k, bounds, mean, s1,
    IMSLS_PRINT,
    IMSLS_ERR_EST, &err,
    0);
    printf("F(X1<%2.1f, X2<%2.1f, X3<%2.1f, ",
           bounds[0], bounds[1], bounds[2]);
    printf("X4<%2.1f, X5<%2.1f, X6<%2.1f) = %f\n",
           bounds[3], bounds[4], bounds[5], f);
    printf("Estimated Error = %g\n", err);
    /* example of error message when sigma is not positive definite */
    f = imsls_f_multivariate_normal_cdf(k, bounds, mean, s2,
    IMSLS_ERR_EST, &err,
    0);
}

```

Output

```

                Original H Limits
    1.0    2.5    2.0    0.5    0.0    0.8

                Original Sigma Matrix
    1.0    0.1    0.2    0.3    0.4    0.0
    0.1    1.0    0.6    0.1    0.2    0.5
    0.2    0.6    1.0    0.0    0.1    0.2
    0.3    0.1    0.0    1.0    0.0    0.5
    0.4    0.2    0.1    0.0    1.0    0.3
    0.0    0.5    0.2    0.5    0.3    1.0

                Sorted Sigma Matrix
    1.0    0.3    0.4    0.0    0.1    0.2
    0.3    1.0    0.0    0.5    0.1    0.0
    0.4    0.0    1.0    0.3    0.2    0.1
    0.0    0.5    0.3    1.0    0.5    0.2
    0.1    0.1    0.2    0.5    1.0    0.6
    0.2    0.0    0.1    0.2    0.6    1.0

Eigenvalues of Sigma
eigenvalue[0] = 2.215651
eigenvalue[1] = 1.256233

```

```

eigenvalue[2] = 1.165661
eigenvalue[3] = 0.843083
eigenvalue[4] = 0.324266
eigenvalue[5] = 0.195106

Condition Number of Sigma = 7.327064

Cholesky Decomposition of Sorted Sigma Matrix
1.000 0.300 0.400 0.000 0.100 0.200
0.300 0.954 -0.126 0.524 0.073 -0.063
0.400 -0.126 0.908 0.403 0.186 0.013
0.000 0.524 0.403 0.750 0.515 0.303
0.100 0.073 0.186 0.515 0.827 0.515
0.200 -0.063 0.013 0.303 0.515 0.774

Prob. = 0.0872375 Error = 3.10012e-005
F(X1<1.0, X2<2.5, X3<2.0, X4<0.5, X5<0.0, X6<0.8) = 0.087237
Estimated Error = 3.10012e-005
eigenvalue[0] = 2.477894
eigenvalue[1] = 1.250438
eigenvalue[2] = 1.039730
eigenvalue[3] = 0.854005
eigenvalue[4] = 0.382186
eigenvalue[5] = -0.004253

*** FATAL Error IMSLS_SIGMA_SINGULAR from
*** imsls_f_multivariate_normal_cdf.
*** "sigma" is not positive definite. Its smallest eigenvalue is
*** "e[5]"=-4.252925e-003 which is less than
*** "tolerance"=1.192093e-007.

```

Warning Errors

IMSLS_MAX_EVALS_EXCEEDED

The maximum number of iterations for the CDF calculation has exceeded `max_evals`. Required accuracy may not have been achieved.

Fatal Errors

IMSLS_SIGMA_SINGULAR

"sigma" is not positive definite. Its smallest eigenvalue is "e[#]"=#, which is less than "tolerance"=#.

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, $F(x)$, of a standard normal (Gaussian) random variable, $\text{imsls_f_normal_inverse_cdf}(p) = F^{-1}(x)$, where

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt$$

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` 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>
#include <stdio.h>

int main()
{
    float  p = 0.9, x;

    x = imsls_f_normal_inverse_cdf(p);
    printf("The %2.0fth percentile of a standard normal is "
           "%6.4f.\n", p*100.0, x);
}
```

Output

```
The 90th percentile of a standard normal is 1.2816.
```

t_cdf

Evaluates the Student's t cumulative distribution function (CDF).

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 cumulative distribution function is to be evaluated.

float df (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The probability that a Student's t random variable takes a value less than or equal to the input t .

Description

Function `imsls_f_t_cdf` evaluates the cumulative distribution function of a Student's t random variable with $\nu = \text{df}$ degrees of freedom. If $t^2 \geq \nu$, the following identity relating the Student's t cumulative distribution function, $F(t, \nu)$ to the incomplete beta ratio function $I_x(a, b)$ is used:

$$F(t|\nu) = \frac{1}{2}I_x\left(\frac{\nu}{2}, \frac{1}{2}\right), t \leq 0, t^2 \geq \nu$$

where

$$x = \frac{\nu}{t^2 + \nu}$$

and

$$F(t|v) = 1 - F(-t,v), t > 0, t^2 \geq v$$

If $t^2 < v$, the solution space is partitioned into four algorithms as follows: If $v \geq 64$ and $t^2/v \leq 0.1$, a Cornish-Fisher expansion is used to evaluate the distribution function. If $v < 64$ and an integer and $|t| < 2.0$, a trigonometric series is used (see Abramowitz and Stegun 1964, Equations 26.7.3 and 26.7.4 with some rearrangement). If $v < 64$ and an integer and $|t| \geq 2.0$, a series given by Hill (1970) that converges well for large values of t is used. For the remaining $t^2 < v$ cases, $F(t|v)$ is calculated using the identity:

$$F(t|v) = I_x\left(\frac{v}{2}, \frac{v}{2}\right)$$

where

$$x = \frac{t + \sqrt{t^2 + v}}{2\sqrt{t^2 + v}}$$

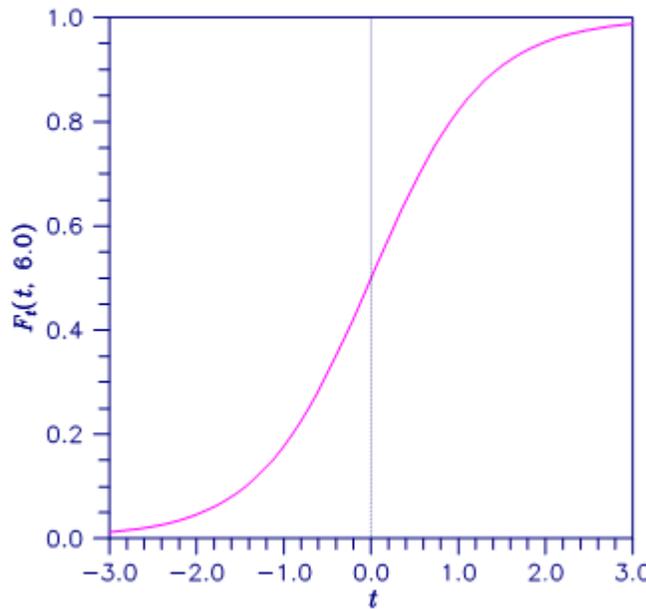


Figure 19, Plot of $F_t(t, 6.0)$

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>
#include <stdio.h>

int main ()
{
    float  t = 2.447, df = 6.0, p;

    p = 2.0*imsls_f_t_cdf(-t,df);
    printf("Pr(|t(%1.0f)| > %5.3f) = %6.4f\n", df, t, p);
}
```

Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

t_inverse_cdf

Evaluates the inverse of the Student's t distribution function.

Synopsis

```
#include <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 $\nu = \text{df}$ 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>
#include <stdio.h>

int main()
{
    float  df = 6.0, p = 0.975, t;

    t = imsls_f_t_inverse_cdf(p,df);
    printf("The two-sided t(%1.0f) %4.2f critical value is "
           "%6.3f\n", df, (1.0-p)*2.0, 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.

complementary_t_cdf

Evaluates the complement of the Student's t distribution.

Synopsis

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

```
float imsls_f_complementary_t_cdf (float t, float df)
```

The type *double* function is `imsls_d_complementary_t_cdf`.

Required Arguments

float **t** (Input)

Argument for which $\Pr(x > t)$ 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 greater than t .

Description

Function `imsls_f_complementary_t_cdf` evaluates one minus the distribution function of a Student's t random variable with $\nu = \text{df}$ degrees of freedom. If $t^2 \geq \nu$, the following identity relating the complementary Student's t cumulative distribution function, denoted by $\bar{F}(t|\nu)$, to the incomplete beta ratio function $I_x(a, b)$ is used:

$$\bar{F}(t|\nu) = \frac{1}{2} I_x\left(\frac{\nu}{2}, \frac{1}{2}\right), \quad t > 0, t^2 \geq \nu$$

where

$$x = \frac{v}{t^2 + v}$$

and

$$\bar{F}(t|v) = 1 - \bar{F}(-t | v), t \leq 0, t^2 \geq v$$

If $t^2 < v$, the solution space is partitioned into four algorithms as follows: If $v \geq 64$ and $t^2/v \leq 0.1$, a Cornish-Fisher expansion is used to evaluate the distribution function. If $v < 64$ and an integer and $|t| < 2.0$, a trigonometric series is used (see Abramowitz and Stegun 1964, Equations 26.7.3 and 26.7.4 with some rearrangement). If $v < 64$ and an integer and $|t| \geq 2.0$, a series given by Hill (1970) that converges well for large values of t is used. For the remaining $t^2 < v$ cases, $\bar{F}(t|v)$ is calculated using the identity:

$$\bar{F}(t|v) = I_{1-x}\left(\frac{v}{2}, \frac{v}{2}\right)$$

where

$$x = \frac{t + \sqrt{t^2 + v}}{2\sqrt{t^2 + v}}$$

This function provides higher right tail accuracy for the Student's t distribution.

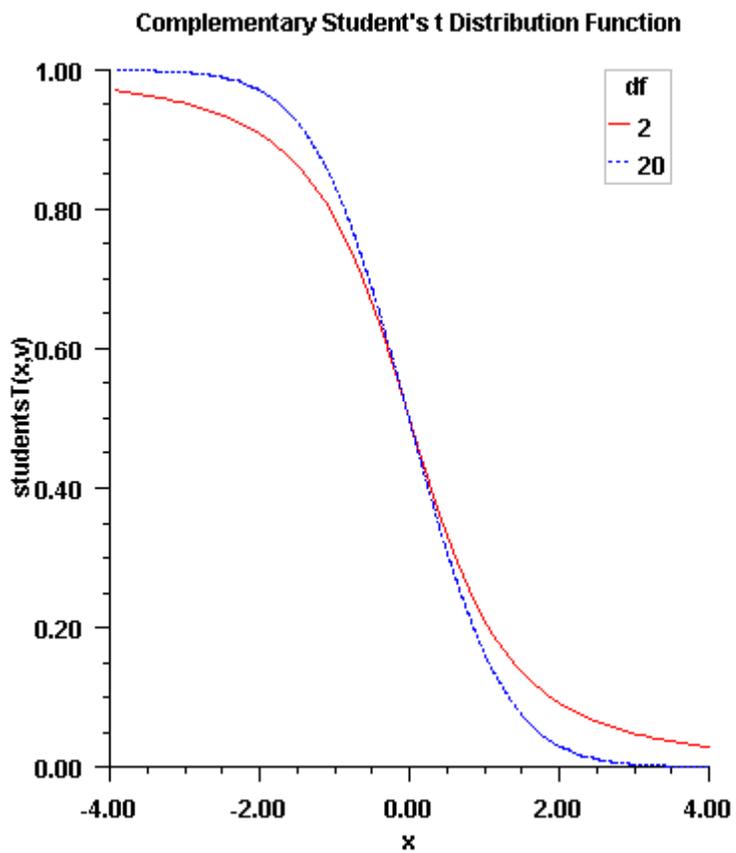


Figure 20, Plot of $F_t(t, df)$

Example

This example finds the 2-tail probability that a Student's t random variable exceeds 2.447.

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

int main ()
{
    float  t = 2.447, df = 6.0, p;

    p = 2.0*imsls_f_complementary_t_cdf(t,df);
    printf("Pr(|t(%1.0f)| > %4.3f) = %6.4f\n", df, t, p);
}
```

Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

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 $\nu = \text{df}$, $\delta = \text{delta}$, and $t_0 = t$,

$$F(t_0) = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi} \Gamma(v/2) (v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \frac{\Gamma((v+i+1)/2) [\delta x]^i (2/(v+x^2))^{i/2}}{i!} dx$$

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.

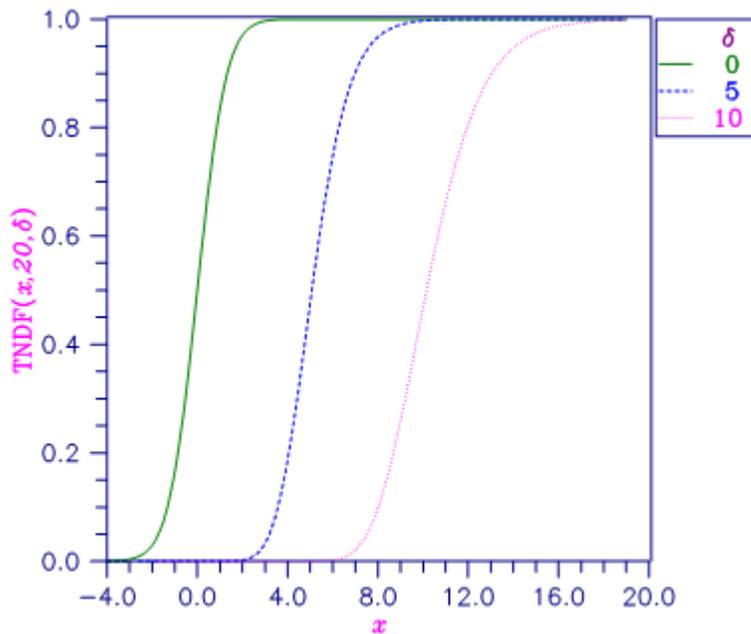


Figure 21, Noncentral Student's t Distribution Function

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>

int main()
{
    int df = 6;
    float t = 12.0, delta = 6.0, p;

    p = imsls_f_non_central_t_cdf(t, df, delta);
    printf("The probability that t is less than %2.0f "
           "is %6.4f.\n", t, 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 $P = p$, $v = df$, and $\delta = delta$, it determines $t_0 (= \text{imsls_f_non_central_t_inv_cdf}(p, df, delta))$, such that

$$P = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi} \Gamma(v/2) (v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2) \frac{\delta^i}{i!} \left(\frac{-2x^2}{v+x^2}\right)^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. In other words:

$$F^{-1} = (P|df, delta) = x$$

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](#) 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 function [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>

int main()
{
    int df = 6;
    float p = 0.95, delta = 6.0, t;

    t = imsls_f_non_central_t_inv_cdf(p, df, delta);
    printf("The %4.2f noncentral t critical value is "
           "%6.4f.\n", 1.0-p, t);
}
```

Output

```
The 0.05 noncentral t critical value is 11.995.
```

non_central_t_pdf

Evaluates the noncentral Student's t probability density function.

Synopsis

```
#include <imspls.h>
```

```
float impls_f_non_central_t_pdf (floatt, floatdf, floatdelta)
```

The type *double* function is `impls_d_non_central_t_pdf`.

Required Arguments

floatt (Input)

Argument for which the noncentral Student's t probability density function is to be evaluated.

floatdf (Input)

Number of degrees of freedom of the noncentral Student's t distribution. *df* must be greater than 0.

floatdelta (Input)

Noncentrality parameter.

Return Value

The probability density associated with a noncentral Student's t random variable with value t .

Description

If w is a normally distributed random variable with unit variance and mean δ and u is a chi-square random variable with ν degrees of freedom that is statistically independent of w , then

$$T = w / \sqrt{u/\nu}$$

is a noncentral t -distributed random variable with ν degrees of freedom and noncentrality parameter δ , that is, with $\nu = df$, and $\delta = delta$. The probability density function for the noncentral t -distribution is:

$$f(t|v,\delta) = \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi} \Gamma(v/2) (v+t^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \phi_i$$

where

$$\phi_i = \frac{\Gamma((v+i+1)/2) [\delta t]^i (2/(v+t^2))^{i/2}}{i!}$$

and $t = t$.

For $\delta = 0$, the PDF reduces to the (central) Student's t PDF:

$$f(t|v,0) = \frac{\Gamma((v+1)/2) (1 + (t^2/v))^{-(v+1)/2}}{\sqrt{v\pi} \Gamma(v/2)}$$

and, for $t = 0$, the PDF becomes:

$$f(0|v,\delta) = \frac{\Gamma((v+1)/2) e^{-\delta^2/2}}{\sqrt{v\pi} \Gamma(v/2)}$$

Example

This example calculates the noncentral Student's t PDF for a distribution with 2 degrees of freedom and noncentrality parameter $\delta = 10$.

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

int main()
{
    int i;
    float t[] = {-.5, 1.5, 3.5, 7.5, 51.5, 99.5};
    float df = 2.0, delta = 10.0, pdfv;

    printf ("\n\n df: %4.0f; delta: %4.0f\n\n", df, delta);
    printf ("    t        pdf(t)\n");

    for (i=0; i<6; i++) {
        pdfv = imsls_f_non_central_t_pdf(t[i], df, delta);
        printf (" %5.1f %12.4e\n", t[i], pdfv);
    }
}
```

Output

df:	2;	delta:	10
t	pdf(t)		
-0.5	1.6399e-024		
1.5	7.4417e-010		
3.5	2.8972e-003		
7.5	7.8853e-002		
51.5	1.4215e-003		
99.5	2.0290e-004		

pareto_cdf

Evaluates the Pareto cumulative probability distribution function.

Synopsis

```
#include <ims1s.h>

float ims1s_f_pareto_cdf (float x, float xm, float k)
```

The type *double* function is `ims1s_d_pareto_cdf`.

Required Arguments

- float x* (Input)
Argument for which the Pareto distribution function is to be evaluated.
- float xm* (Input)
The scale parameter.
- float k* (Input)
The shape parameter.

Return Value

The probability that a Pareto random variable takes a value less than or equal to *x*. NaN is returned on error.

Description

The `ims1s_f_pareto_cdf` function evaluates the distribution function, F , of a Pareto random variable with scale parameter x_m and shape parameter k . It is given by:

$$F(x|x_m, k) = 1 - \left(\frac{x_m}{x}\right)^k$$

where $x_m > 0$ and $k > 0$. The function is only defined for $x \geq x_m$.

Example

Suppose X is a Pareto random variable with $x_m = 0.4$ and $k = 0.7$. The function finds the probability that X is less than or equal to 0.5.

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

int main(){
    float x = 0.5;
    float xm = 0.4;
    float k = 0.7;
    float pr = 0.0;

    pr = imsls_f_pareto_cdf(x, xm, k);
    printf("Pr(x <= %3.1f) = %6.4f\n", x, pr);
}
```

Output

```
Pr(x <= 0.5) = 0.1446
```

pareto_pdf

Evaluates the Pareto probability density function.

Synopsis

```
#include <imsls.h>

float imsls_f_pareto_pdf (float x, float xm, float k)
```

The type *double* function is `imsls_d_pareto_pdf`.

Required Arguments

float **x** (Input)
Argument for which the function is to be evaluated.

float **xm** (Input)
The scale parameter.

float **k** (Input)
The shape parameter.

Return Value

The probability density at **x**. NaN is returned on error.

Description

The probability density function of the Pareto distribution is:

$$f(x|x_m, k) = k \frac{x_m^k}{x^{k+1}}$$

where the scale parameter $x_m > 0$ and the shape parameter $k > 0$. The function is only defined for $x \geq x_m$.

Example

In this example, we evaluate the Pareto PDF at $x = 0.5$, $x_m = 0.4$ and $k = 0.7$.

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

int main(){
    float x = 0.5;
    float xm = 0.4;
    float k = 0.7;
    float pr = 0.0;

    pr = imsls_f_pareto_pdf(x, xm, k);
    printf("The probability density of a Pareto random");
    printf("variable X with\na scale parameter xm = ");
    printf("%3.1f and a shape parameter ", xm);
    printf("k = %3.1f\nand value x = %3.1f is %6.4f.\n",
           k, x, pr);
}
```

Output

```
The probability density of a Pareto random variable X with
a scale parameter xm = 0.4 and a shape parameter k = 0.7
and value x = 0.5 is 1.1975.
```

max_likelihood_estimates

Calculates maximum likelihood estimates (MLE) for the parameters of one of several univariate probability distributions.

Synopsis

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

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

The type *double* function is `imsls_d_max_likelihood_estimates`.

Required Arguments

int `n_observations` (Input)

Number of observations.

float `x[]` (Input)

Array of length `n_observations` containing the data.

int `ipdf` (Input)

Specifies the probability density function.

Distribution	ipdf	n_parameters	i	parameters[i]
Discrete uniform	0	1	0	scale - upper limit
Bernoulli	1	1	0	probability of success (mean)
Binomial ⁽¹⁾	2	1	0	probability of success
Negative binomial ⁽²⁾	3	1	0	probability of success
Poisson	4	1	0	location (mean) - θ
Geometric	5	1	0	probability of success
Continuous uniform	6	2	0 1	scale - lower boundary scale - upper boundary
Beta	7	2	0 1	shape - p shape - q
Exponential	8	1	0	scale - b

Distribution	ipdf	n_parameters	i	parameters[i]
Gamma	9	2	0 1	shape - k scale - θ
Weibull	10	2	0 1	scale - λ shape - k
Rayleigh	11	1	0	scale - α
Extreme value	12	2	0 1	location - μ scale - σ
Generalized extreme value	13	3	0 1 2	location - μ scale - σ shape - β
Pareto	14	2	0 1	scale (lower boundary) x_m shape - k
Generalized Pareto	15	2	0 1	scale - σ shape - α
Normal	16	2	0 1	location(mean) - μ scale(variance) - σ^2
Log-normal	17	2	0 1	location(mean of log(x)) - μ scale(variance of log(x)) - σ^2
Logistic	18	2	0 1	location(mean) - μ scale - s
Log-logistic	19	2	0 1	scale(exp(mean)) - e^μ shape - β
Inverse Gaussian	20	2	0 1	location(mean) - μ shape - λ

Note: 1 - The binomial distribution requires the optional argument `IMSLS_NUMBER_OF_TRIALS`.

Note: 2 - The negative binomial distribution requires the optional argument `IMSLS_NUMBER_OF_FAILURES`.

Return Value

A pointer to an array of length `n_parameters` containing the parameter values (see `ipdf` table above).

Synopsis with Optional Arguments

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

```
float *imsls_f_max_likelihood_estimates (int n_observations, float x[ ], int ipdf,  
    IMSLS_PRINT_LEVEL, int iprint,  
    IMSLS_N_PARAMETERS, int *n_parameters,  
    IMSLS_NUMBER_OF_TRIALS, int n_trials,  
    IMSLS_NUMBER_OF_FAILURES, int n_failures,  
    IMSLS_MLOGLIKE, float *mloglike,  
    IMSLS_STD_ERRORS, float **se,  
    IMSLS_STD_ERRORS_USER, float se[ ],  
    IMSLS_HESSIAN, float **hess,  
    IMSLS_HESSIAN_USER, float hess[ ],  
    IMSLS_RETURN_USER, float param[ ],  
    IMSLS_PARAM_LB, float paramlb[ ],  
    IMSLS_PARAM_UB, float paramub[ ],  
    IMSLS_INITIAL_ESTIMATES, float initial_estimates[ ],  
    IMSLS_XSCALE, float xscale[ ],  
    IMSLS_MAX_ITERATIONS, int maxit,  
    IMSLS_MAX_FCN, int maxfcn,  
    IMSLS_MAX_GRAD, int maxgrad,  
    0)
```

Optional Arguments

IMSL5_PRINT_LEVEL, *int* `iprint` (Input)

Printing option.

iprint	Action
0	No printing
1	Print final results only
2	Print intermediate and final results

Default: `iprint = 0`.

IMSL5_N_PARAMETERS, *int** `n_parameters` (Output)

The number of parameters in the distribution specified by `ipdf`.

IMSL5_NUMBER_OF_TRIALS, *int* `n_trials` (Input)

The number of trials. `n_trials` is required for the binomial distribution, (`ipdf = 2`).

Default: Not used, except for `ipdf = 2`.

IMSL5_NUMBER_OF_FAILURES, *int* `n_failures` (Input)

The number of failures. `n_failures` is required for the negative binomial distribution, (`ipdf = 3`).

Default: Not used, except for `ipdf = 3`.

IMSL5_MLOGLIKE, *float** `mloglike` (Output)

Minus log-likelihood evaluated at the parameter estimates.

IMSL5_STD_ERRORS, *float*** `se` (Output)

Address of a pointer to an internally allocated array of length `n_parameters` containing the standard errors of the parameter estimates.

IMSL5_STD_ERRORS_USER, *float* `se[]` (Output)

Storage for array `se` is provided by the user. See `IMSL5_STD_ERRORS`.

IMSL5_HESSIAN, *float*** `hess` (Output)

Address of a pointer to an internally allocated array of length `n_parameters × n_parameters` containing the Hessian matrix.

IMSL5_HESSIAN_USER, *float* `hess[]` (Output)

Storage for array `hess` is provided by the user. See `IMSL5_HESSIAN`.

IMSLS_RETURN_USER, *float* param[] (Output)

User-allocated array of length `n_parameters` containing the estimated parameters.

Note: The following optional arguments are used in cases in which a quasi-Newton method is used to solve the likelihood problem (`ipdf = 7,9,10,12,13,15,18,19`).

IMSLS_PARAM_LB, *float* paramlb[] (Input)

Array of length `n_parameters` containing the lower bounds of the parameters.

Exceptions	paramlb
Extreme value distribution (<code>ipdf = 12</code>)	<code>paramlb[1] = 0.25</code> , for the scale parameter
Generalized Pareto distribution (<code>ipdf = 15</code>)	<code>paramlb[1] = -5.0</code> , for the shape parameter
Generalized extreme value distribution (<code>ipdf = 13</code>)	<code>paramlb[2] = -10.0</code> , for the shape parameter

Default: The default lower bound depends on the range of the parameter. That is, if the range of the parameter is positive for the desired distribution, `paramlb[i] = 0.01`. If the range of the parameter is non-negative (≥ 0), then `paramlb[i] = 0.0`. If the range of the parameter is unbounded, then `paramlb[i] = -10000.00`.

IMSLS_PARAM_UB, *float* paramub[] (Input)

Array of length `n_parameters` containing the upper bounds of the parameters.

Exceptions	paramub
Generalized Pareto distribution (<code>ipdf = 15</code>)	<code>paramub[1] = -5.0</code> , for the shape parameter
Generalized extreme value distribution (<code>ipdf = 13</code>)	<code>paramub[2] = -10.0</code> , for the shape parameter

Default: `paramub[i] = 10000.0`.

IMSLS_INITIAL_ESTIMATES, *float* initial_estimates[] (Input)

Array of length `n_parameters` containing the initial estimates of the parameters.

Default: Method of moments estimates are used for initial estimates.

IMSLS_XSCALE, *float* xscale[] (Input)

Array of length `n_parameters` containing the scaling factors for the parameters. `xscale` is used in the optimization algorithm in scaling the gradient and the distance between two points.

Default: `xscale[i] = 1.0`.

`IMSLS_MAX_ITERATIONS`, *int* `maxit` (Input)

Maximum number of iterations.

Default: `maxit` = 100.

`IMSLS_MAX_FCN`, *int* `maxfcn` (Input)

Maximum number of function evaluations.

Default: `maxfcn` = 400.

`IMSLS_MAX_GRAD`, *int* `maxgrad` (Input)

Maximum number of gradient evaluations.

Default: `maxgrad` = 400.

Description

Function `imsls_f_max_likelihood_estimates` calculates maximum likelihood estimates for the parameters of a univariate probability distribution, where the distribution is one specified by `ipdf` and where the input data \mathbf{x} is (assumed to be) a random sample from that distribution.

Let $\{x_i, i=1, \dots, N\}$ represent a random sample from a probability distribution with density function $f(x|\theta)$, which depends on a vector $\theta \in \mathcal{R}^p$ containing the values of the parameters of the distribution. The values in θ are fixed but unknown and the problem is to find an estimate for θ given the sample data.

The likelihood function is defined to be the product

$$L(\theta | \{x_i; i=1, \dots, N\}) = \prod_{i=1, \dots, N} f(x_i | \theta)$$

The estimator

$$\begin{aligned} \hat{\theta}_{MLE} &= \arg \max_{\theta} L(\theta | \{x_1, x_2, \dots, x_N\}) \\ &= \arg \max_{\theta} \prod_{i=1, \dots, N} f(x_i | \theta) \\ &= \arg \max_{\theta} \sum_{i=1, \dots, N} \log(f(x_i | \theta)) \end{aligned}$$

That is, the estimator that maximizes L also maximizes $\log L$ and is the maximum likelihood estimate, or *MLE* for θ .

The likelihood problem is in general a constrained non-linear optimization problem, where the constraints are determined by the permissible range of θ . In some situations, the problem has a closed form solution. Otherwise, `imsls_f_max_likelihood_estimates` uses a quasi-Newton method to solve the likelihood problem. If

optional argument `IMSLS_INITIAL_ESTIMATES` is not supplied, method of moments estimates serve as starting values of the parameters. In some cases, method of moments estimators may not exist, such as when certain moments of the true distribution do not exist; thus it is possible that the starting values are not truly method of moments estimates.

Upper and lower bounds, when needed for the optimization, have default values for each selection of `ipdf` (defaults will vary depending on the allowable range of the parameters). It is possible that the optimization will fail. In such cases, the user may try adjusting upper and lower bounds using the optional arguments `IMSLS_PARAM_LB`, `IMSLS_PARAM_UB`, or adjusting up or down the scaling factors using optional argument `IMSLS_XSCALE`, which can sometimes help the optimization converge.

Standard errors and covariances are supplied, in most cases, using the asymptotic properties of *MLE* estimators. Under some general regularity conditions, *MLE* estimates are consistent and asymptotically normally distributed with variance-covariance equal to the inverse *Fisher's Information* matrix evaluated at the true value of the parameter, θ_0 :

$$\text{Var}(\hat{\theta}) = I(\theta_0)^{-1} = -E \left[\frac{\partial^2 \log L}{\partial \theta^2} \right]_{\theta_0}^{-1}$$

`imsls_f_max_likelihood_estimates` approximates the asymptotic variance using the negative inverse Hessian evaluated at the *MLE* estimate:

$$\text{Var}(\hat{\theta}) \approx - \left[\frac{\partial^2 \log L}{\partial \theta^2} \right]_{\theta=\hat{\theta}_{MLE}}^{-1}$$

The Hessian is approximated numerically for all but a few cases where it can be determined in closed form.

In cases when the asymptotic result does not hold, standard errors may be available from the known sampling distribution. For example, the *MLE* estimate of the Pareto distribution location parameter is the minimum of the sample. The variance is estimated using the known sampling distribution of the minimum or first order-statistic for the Pareto distribution.

For further details regarding the properties of the estimators and the theory of the maximum likelihood method, see Kendall and Stuart (1979). The different probability distributions have wide coverage in the statistical literature. See Johnson and Kotz (1970a, 1970b, or later editions).

Parameter estimation (including maximum likelihood) for the generalized Pareto distribution is studied in Hosking and Wallis (1987) and Giles and Feng (2009), and estimation for the generalized extreme value distribution is treated in Hosking, Wallis, and Wood (1985).

Remarks

1. The location parameter is not estimated for the generalized Pareto distribution (`ipdf=15`). Instead, the minimum of the sample is subtracted from each observation before the estimation procedure.
2. Only the probability of success parameter is estimated for the binomial and negative binomial distributions, (`ipdf = 2,3`). The number of trials and the number of failures, respectively, must be provided using optional arguments `IMSLS_NUMBER_OF_TRIALS` or `IMSLS_NUMBER_OF_FAILURES`.
3. `imsls_f_max_likelihood_estimates` issues an error if missing or NaN values are encountered in the input data. Missing or NaN values should be removed before calling `imsls_f_max_likelihood_estimates`.

Examples

Example 1

The data are $N=100$ observations generated from the logistic distribution with location parameter $\mu = 0.85$ and parameter $\sigma = 0.5$.

```
#include <imsls.h>

int main() {
    int ipdf = 18, n_observations = 100;
    float *p_hess, *p_se, *param, mloglike;
    float x[100] = {
        2.020394,2.562315,-0.5453395,1.258546,0.7704533, 0.3662717,
        0.6885536,2.619634,-0.49581,2.972249,0.5356222,0.4262079,
        1.023666,0.8286033,1.319018,2.123659,0.3904647,-0.1196832,
        1.629261,1.069602,0.9438083,1.314796,1.404453,-0.5496156,
        0.8326595,1.570288,1.326737,0.9619384,-0.1795268,1.330161,
        -0.2916453,0.7430826,1.640854,1.582755,1.559261,0.6177695,
        1.739638,1.308973,0.568709,0.2587071,0.745583,1.003815,
        1.475413,1.444586,0.4515438,1.264374,1.788313,1.062330,
        2.126034,0.3626510,1.365612,0.5044735,2.51385,0.7910572,
        0.5932584,1.140248,2.104453,1.345562,-0.9120445,0.0006519341,
        1.049729,-0.8246097,0.8053433,1.493787,-0.5199705,2.285175,
        0.9005916,2.108943,1.40268,1.813626,1.007817,1.925250,1.037391,
        0.6767235,-0.3574937,0.696697,1.104745,-0.7691124,1.554932,
        2.090315,0.60919,0.4949385,-2.449544,0.668952,0.9480486,
        0.9908558,-1.495384,2.179275,0.1858808,-0.3715074,0.1447150,
        0.857202,1.805844,0.405371,1.425935,0.3187476,1.536181,
        -0.6352768,0.5692068,1.706736};

    param = imsls_f_max_likelihood_estimates(n_observations, x, ipdf,
        IMSLS_PRINT_LEVEL, 2,
        IMSLS_HESSIAN, &p_hess,
        IMSLS_STD_ERRORS, &p_se,
        IMSLS_MLOGLIKE, &mloglike,
        0);
```

}

Output

Maximum likelihood estimation for the logistic distribution

Starting Estimates: 0.90677 0.51128

Initial -log-likelihood: 132.75304

-log-likelihood 132.61490

MLE for parameter 1 0.95321

MLE for parameter 2 0.50953

Std error for parameter 1 0.08825

Std error for parameter 2 0.04354

Hessian

	1	2
1	-128.5	-7.6
2	-7.6	-527.9

Example 2

The data are $N = 100$ observations generated from the generalized extreme value distribution with location parameter $\mu = 0$, scale parameter $\sigma = 1.0$, and shape parameter $\xi = -0.25$.

```
#include <imsls.h>

int main() {
    int ipdf = 13, n_observations = 100;
    float *p_hess, *p_se, *param, mloglike;
    float x[100] = {
        0.7688048,0.1944504,-0.2992029,-0.3853738,
        -1.185593,0.3056149,-0.4407711,0.5001115,
        0.3635027,-1.058632,-0.2927695,-0.3205969,
        0.03367599,0.8850839,1.860485,0.4841038,
        0.5421101,1.883694,1.707392,0.2166106,
        1.537204,1.340291,0.4589722,1.616080,
        -0.8389288,0.7057426,1.532988,1.161350,
        0.9475416,0.4995294,-0.2392898,0.8167126,
        0.992479,-0.8357962,-0.3194499,1.233603,
        2.321555,-0.3715629,-0.1735171,0.4624801,
        -0.6249577,0.7040129,-0.3598889,0.7121399,
        -0.5178735,-1.069429,0.7169358,0.4148059,
        1.606248,-0.4640152,1.463425,0.9544342,
        -1.383239,0.1393160,0.622689,0.365793,
        0.7592438,0.810005,0.3483791,2.375727,
        -0.08124195,-0.4726068,0.1496043,0.4961212,
        1.532723,-0.1106993,1.028553,0.856018,
        -0.6634978,0.3573150,0.06391576,0.3760349,
        -0.5998756,0.4158309,-0.2832369,-1.023551,
        1.116887,1.237714,1.900794,0.6010037,
```

```

1.599663,-0.3341879,0.5278575,0.5497694,
0.6392933,0.592865,1.646261,-1.042950,
-1.113611,1.229645,1.655998,0.6913992,
0.4548073,0.4982649,-1.073640,-0.4765107,
-0.8692533,-0.8316462,-0.03609102,0.655814};

```

```

param = imsls_f_max_likelihood_estimates(n_observations,
x, ipdf,
IMSLS_PRINT_LEVEL, 2,
IMSLS_HESSIAN, &p_hess,
IMSLS_STD_ERRORS, &p_se,
IMSLS_MLOGLIKE, &mloglike,
0);
}

```

Output

```

Maximum likelihood estimation for the generalized extreme value distribution
Starting Estimates:  -0.00888    0.67451    0.00000

```

```

Initial -log-likelihood:  135.43817

```

```

-log-likelihood  126.09406

```

```

MLE for parameter      1      0.07541

```

```

MLE for parameter      2      0.85112

```

```

MLE for parameter      3     -0.27974

```

```

Std error for parameter 1      0.09419

```

```

Std error for parameter 2      0.06906

```

```

Std error for parameter 3      0.06603

```

```

Hessian

```

	1	2	3
1	-141.7	-53.9	-112.4
2	-53.9	-340.8	-239.7
3	-112.4	-239.7	-439.7

Warning Errors

IMSLS_HESSIAN_NOT_CALCULATED

The Hessian is not calculated for the requested distribution.

IMSLS_HESSIAN_NOT_USED

The Hessian is not used to calculate the standard errors of the estimates for the # distribution.

IMSLS_HESSIAN_NOT_CALC_2

For the Pareto distribution, the Hessian cannot be calculated because the parameter estimate is 0.

Random Number Generation

Functions

Univariate Discrete Distributions

Generates pseudorandom binomial numbers	random_binomial	1223
Generates pseudorandom geometric numbers	random_geometric	1226
Generates pseudorandom hypergeometric numbers	random_hypergeometric	1228
Generates pseudorandom logarithmic numbers	random_logarithmic	1231
Generates pseudorandom negative binomial numbers	random_neg_binomial	1233
Generates pseudorandom Poisson numbers	random_poisson	1236
Generates pseudorandom discrete uniform numbers	random_uniform_discrete	1238
Generates pseudorandom numbers from a general discrete distribution	random_general_discrete	1240
Sets up a table to generate pseudorandom numbers from a general discrete distribution	discrete_table_setup	1245

Univariate Continuous Distributions

Generates pseudorandom beta numbers	random_beta	1251
Generates pseudorandom Cauchy numbers	random_cauchy	1254
Generates pseudorandom chi_squared numbers	random_chi_squared	1256
Generates pseudorandom exponential numbers	random_exponential	1258
Generates pseudorandom mixed exponential numbers	random_exponential_mix	1260
Generates pseudorandom gamma numbers	random_gamma	1263
Generates pseudorandom lognormal numbers	random_lognormal	1266
Generates pseudorandom normal numbers	random_normal	1269
Generates pseudorandom numbers from a stable distribution	random_stable	1272
Generates pseudorandom Student's t	random_student_t	1275
Generates pseudorandom triangular numbers	random_triangular	1277
Generates pseudorandom uniform numbers	random_uniform	1279
Generates pseudorandom Von Mises numbers	random_von_mises	1282
Generates pseudorandom Weibull numbers	random_weibull	1284
Generates pseudorandom numbers from a general continuous distribution	random_general_continuous	1287
Sets up table to generate pseudorandom numbers		

from a general continuous distribution	continuous_table_setup	1290
Multivariate Continuous Distributions		
Generates multivariate normal vectors	random_normal_multivariate	1294
Generates a pseudorandom orthogonal matrix or a correlation matrix	random_orthogonal_matrix	1298
Generates pseudorandom numbers from a multivariate distribution determined from a given sample.	random_mvar_from_data	1301
Generates pseudorandom numbers from a multinomial distribution	random_multinomial	1305
Generates pseudorandom points on a unit circle or K-dimensional sphere	random_sphere	1308
Generates a pseudorandom two-way table	random_table_twoway	1311
Generates multivariate Gaussian Copula vectors	random_mvar_gaussian_copula	1314
Generates multivariate Student's t Copula vectors	random_mvar_t_copula	1319
Generates a canonical correlation matrix	canonical_correlation	1325
Order Statistics		
Generates pseudorandom order statistics from a standard normal distribution.	random_order_normal	1331
Generates pseudorandom order statistics from a uniform (0, 1) distribution	random_order_uniform	1334
Stochastic Processes		
Generates pseudorandom ARMA process numbers	random_arma	1337
Generates pseudorandom numbers from a nonhomogeneous Poisson process	random_npp	1342
Samples and Permutations		
Generates a pseudorandom permutation	random_permutation	1346
Generates a simple pseudorandom sample of indices	random_sample_indices	1348
Generates a simple pseudorandom sample from a finite population	random_sample	1350
Utility Functions		
Selects the uniform (0, 1) generator	random_option	1354
Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator	random_option_get	1356
Retrieves the current value of the seed.	random_seed_get	1357
Retrieves a seed for the congruential generators	random_substream_seed_get	1359
Initializes a random seed	random_seed_set	1361
Sets the current table used in the shuffled generato.	random_table_set	1362
Retrieves the current table used in the shuffled generator	random_table_get	1363
Sets the current able used in the GFSR generator	random_GFSR_table_set	1365
Retrieves the current table used in the GFSR generator.	random_GFSR_table_get	1366
Initializes the 32-bit Mersenne Twister generator using an array.	random_MT32_init	1370

Retrieves the current table used in the 32-bit Mersenne Twister generator	random_MT32_table_get	1371
Sets the current table used in the 32-bit Mersenne Twister generator	random_MT32_table_set	1374
Initializes the 64-bit Mersenne Twister generator using an array	random_MT64_init	1375
Retrieves the current table used in the 64-bit Mersenne Twister generato	random_MT64_table_get	1376
Sets the current table used in the 64-bit Mersenne Twister generator	random_MT64_table_set	1379
Low-discrepancy sequence		
Generates a shuffled Faure sequence	faure_next_point	1380

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 function `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 functions 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 $X_t = X_{t-1563} \oplus X_{t-96}$. 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 functions `imsls_f_random_table_get` and `imsls_f_random_table_set`; and for the GFSR generator; the table is retrieved and set by the functions `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 function `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 functions [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 functions in Chapter 7, [Tests of Goodness of Fit](#), 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 functions 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.

Copula Generators and Canonical Correlation

A copula is a multivariate cumulative probability distribution (CDF) whose arguments are random variables uniformly distributed on the interval $[0, 1]$ corresponding to the probabilities (variates) associated with arbitrarily distributed marginal deviates. The copula structure allows the multivariate CDF to be partitioned into the copula, which has associated with it information characterizing the dependence among the marginal variables, and the set of separate marginal deviates, each of which has its own distribution structure.

Two functions, [imsls_f_random_mvar_gaussian_copula](#) and [imsls_f_random_mvar_t_copula](#), allow the user to specify a correlation structure (in the form of a Cholesky matrix) which can be used to imprint correlation information on a sequence of multivariate random vectors. Each call to one of these functions returns a random vector whose elements (variates) are each uniformly distributed on the interval $[0, 1]$ and correlated according to a user-specified Cholesky matrix. These variate vector sequences may then be inverted to marginal deviate sequences whose distributions and imprinted correlations are user-specified.

Function [imsls_f_random_mvar_gaussian_copula](#) generates a random Gaussian copula vector by inverting a vector of uniform $[0, 1]$ random numbers to an $N(0, 1)$ deviate vector, imprinting the $N(0,1)$ vector with the correlation information by multiplying it with the Cholesky matrix, and then using the $N(0,1)$ CDF to map the Cholesky-imprinted deviate vector back to a vector of imprinted uniform $[0, 1]$ variates.

Function `imsls_f_random_mvar_t_copula` inverts a vector of uniform $[0, 1]$ random numbers to an $N(0, 1)$ deviate vector, imprints the vector with correlation information by multiplying it with the Cholesky matrix, transforms the imprinted $N(0, 1)$ vector to an imprinted Student's t vector (where each element is Student's t distributed with ν degrees of freedom) by dividing each element of the imprinted $N(0, 1)$ vector by $\sqrt{s/\nu}$, where s is a random deviate taken from a chi-squared distribution with ν degrees of freedom, and finally maps each element of the resulting imprinted Student's t vector back to a uniform $[0, 1]$ distributed variate using the Student's t CDF.

The third copula function, `imsls_f_canonical_correlation`, extracts a "canonical correlation" matrix from a sequence of multivariate deviate vectors whose component marginals are arbitrarily distributed. This is accomplished by first extracting the empirical CDF from each of the marginal deviate sequences and then using this empirical CDF to map the deviates to uniform $[0, 1]$ variates which are then inverted to $N(0, 1)$ deviates. Each element C_{ij} of the canonical correlation matrix can then be extracted by averaging the products $z_{it} z_{jt}$ of $N(0, 1)$ deviates i and j over the t -indexed sequence. The utility of function `imsls_f_canonical_correlation` is that because the canonical correlation matrix is derived from $N(0, 1)$ deviates, the correlation is unbiased, i.e. undistorted by the arbitrary marginal distribution structures of the original deviate vector sequences. This is important in such financial applications as portfolio optimization, where correlation is used to estimate and minimize risk.

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

IMSL_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 be less than 1. The probability function (with $n = \mathbf{n}$ and $p = \mathbf{p}$) is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, 1, 2, \dots, n$.

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 <imsls.h>

int 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 success 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 $x = 1, 2, \dots$ 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 $0, 1, 2, \dots$, 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 <imsls.h>

int 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 $x = \max(0, N - L + M), 1, 2, \dots, \min(N, M)$

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 function 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 $i = 0, 1, \dots$, 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 occurrence 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>

int 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 $x = 1, 2, 3, \dots$, 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>

int 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

IMSL_S_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 $x = 0, 1, 2, \dots$

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 $r, r+1, r+2, \dots$ 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>

int 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



[more...](#)

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 $\theta = \text{theta}$) is

$$f(x) = (e^{-\theta} \theta^x) / x! \quad \text{for } x = 0, 1, 2, \dots$$

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

int 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 <imsls.h>

int 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 `IMSL_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 function `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 `imsls_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 iwkc[], float wk[],
    IMSLS_SET_INDEX_VECTORS, int iwkc[], 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 iwkc[], float wk[]` (Output)

User-supplied arrays of length `nmass` used to 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` with 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

Function `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 $i = i_{\min}, i_{\min} + 1, \dots, i_{\min} + n_m - 1$ where $j = i - i_{\min} + 1$, $p_j = \text{probs}[j-1]$, $i_{\min} = \text{imin}$, and $n_m = \text{nmass}$.

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 function `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.

Examples

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 <imsls.h>

int 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);

    imsls_free(ir);

    ir = imsls_f_random_general_discrete(nr, imin, nmass, probs,
        IMSLS_SET_INDEX_VECTORS, iw, 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 <imsls.h>
#include <stdlib.h>

float prf(int ix);

int 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, ndx, &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_pdf (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 $1.0 - \text{del}$.

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 $1.0 - \text{del}$.

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 `IMSLI_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 $\text{prf}(\text{imin}) \geq \text{del}$. `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 `iminin + j` for which `prf(iminin + j) < del`, for $j = 0, 1, \dots, imin_{out} - imin_{in}$, where `iminin` denotes the input value of `imin` and `iminout` 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 `imsls_free`.

Synopsis with Optional Arguments

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

```
float *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`, (Input)

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` is filled.

IMSL5_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 [Passing Data to User-Supplied Functions](#) section at the beginning of this manual for more details.

Description

Function `imsls_f_discrete_table_setup` sets up a table that function `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 function `imsls_f_random_general_discrete`.

Examples

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

float prf(int ix);

int 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

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 function `imsls_f_binomial_pdf` (Chapter 11, [Probability Distribution Functions and Inverses](#)) is used to compute the probabilities.

```

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

float prf(int ix);
int 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_pdf(ix, n, p);
}
```

Output

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

Fatal Errors

IMSL_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

random_beta



[more...](#)

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 `imsls_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

IMSL_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 $p = \text{pin}$ and $q = \text{qin}$, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1} (1-x)^{q-1} \text{ for } 0 \leq x \leq 1$$

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 $1.0 - \epsilon$ 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>

int 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 $\tan[\pi(u - 0.5)]$. 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>

int 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 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_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>

int 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

`IMLS_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.

Deviate 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>
#include <stdio.h>

#define N_RANDOM 5

int 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%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



[more...](#)

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 $p = p$, $\theta_1 = \text{theta1}$, and $\theta_2 = \text{theta2}$.

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, respectively, and with mixing parameter 0.5.

```
#include <imsls.h>

int 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



[more...](#)

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

IMSL_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

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \text{ for } x \geq 0$$

Various computational algorithms are used depending on the value of the shape parameter a . For the special case of $a = 0.5$, squared and halved normal deviates are used; 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>

int 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

[more...](#)

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 $\mu = \text{mean}$ and $\sigma = \text{std}$, 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 <imsls.h>

int 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



[more...](#)

Generates pseudorandom numbers from a normal, $N(\mu, \sigma^2)$, 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_ZIGGURAT_METHOD,  
    IMSLS_RETURN_USER, float r[ ],  
    0)
```

Optional Arguments

`IMSLS_MEAN`, *float mean* (Input)

Parameter `mean` contains the mean, μ , of the $N(\mu, \sigma^2)$ 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 $N(\mu, \sigma^2)$ from which random normal deviates are to be generated.

Default: `variance` = 1.0

`IMSLS_ZIGGURAT_METHOD`, (Input)

By default, random numbers are generated using an inverse CDF technique. When optional argument `IMSLS_ZIGGURAT_METHOD` is specified, the Ziggurat 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, [Probability Distribution Functions and Inverses](#)).

If optional argument `IMSLS_ZIGGURAT_METHOD` is specified, function `imsls_f_random_normal` generates pseudorandom numbers using the Ziggurat method. This method cuts the density into many small pieces. For each random number generated, an interval is chosen at random and a random normal is generated from the chosen interval. In this implementation, the density is cut into 256 pieces, but symmetry is used so that only 128 pieces are needed by the computation. Following Doornik (2005), different uniform random deviates are used to determine which slice to use and to determine the normal deviate from the slice. This method is faster than the default 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>
#include <stdio.h>
#define N_RANDOM 5

int 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:
-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 `imsls_free`.

Synopsis with Optional Arguments

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

```
float *imsls_f_random_stable (int n_random, float alpha, float bprime,  
    IMSLS_RETURN_USER, float r[ ],  
    0)
```

Optional Arguments

IMSLI_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 $0 < \alpha \leq 2$ and $-1 \leq \beta \leq 1$, the characteristic function of the distribution is

$$\phi(t) = \exp[-|t|^\alpha \exp(\pi i \beta (1 - |1 - \alpha|) \text{sign}(t)/2)] \quad \text{for } \alpha \neq 1$$

and

$$\phi(t) = \exp[-|t| (1 + 2i\beta \ln |t|) \text{sign}(t)/\pi] \quad \text{for } \alpha = 1$$

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

$$\beta' = -\tan(\pi(1 - \alpha)/2) \tan(-\pi\beta(1 - |1 - \alpha|)/2) \quad \text{for } \alpha \neq 1$$

and

$$\beta' = \beta \quad \text{for } \alpha = 1$$

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 <imsls.h>

int 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,  
IMSL_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_MEAN, float mean,
    IMSLS_VARIANCE, float variance,
    IMSLS_RETURN_USER, float r[ ],
    0)
```

Optional Arguments

IMSL_S_MEAN, *float mean* (Input)

Mean of the Student's t distribution.

Default: `mean = 0.0`

IMSL_S_VARIANCE, *float variance* (Input)

Variance of the Student's t distribution.

Default: `variance = 1.0`

IMSL_S_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.

Example

In this example, `imsls_f_random_student_t` generates five pseudorandom deviates from a Student's t distribution with 12 degrees of freedom.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int    seed = 123457, n_random = 5;
    float  df = 12.0, *r;
    imsls_random_seed_set (seed);
    r = imsls_f_random_student_t (n_random, df, 0);
    printf("Student's t deviates with %8.4f degrees "
           "of freedom:", df);
    printf("\n%8.4f %8.4f %8.4f %8.4f %8.4f\n",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Student's t deviates with 12.0000 degrees of freedom:
 0.6152  1.1468  0.0877  1.3318 -0.9933
```

random_triangular

[more...](#)

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 $0 \leq x \leq 0.5$, and $f(x) = 4(1 - x)$, for $0.5 < x \leq 1$. 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 <imsls.h>

int 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

`IMLS_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:

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0, 1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by the function `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

int 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

$$f(x) = \frac{1}{2\pi I_0(c)} \exp[c \cos(x)]$$

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 <imsls.h>

int 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: $b = 1.0$

IMSL5_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

$$f(x) = abx^{a-1} \exp(-bx^a)$$

for $x \geq 0$, $a > 0$, 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

$$r(x) = \frac{1}{\alpha^2} x e^{-\left(\frac{x^2}{2\alpha^2}\right)}$$

for $x \geq 0$ 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 Weibull distribution with shape parameter equal to 3.0.

$$\alpha = 3\sqrt{2}$$

```
#include <imsls.h>

int 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

IMSL5_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

[more...](#)

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/Output)

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 function [imsls_f_continuous_table_setup](#).

Return Value

An array of length **n_random** containing the random discrete deviates. To release this space, use `imsls_free`.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *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 (Input)

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

Function `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 function `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

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 function `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 <imsls.h>

float cdf(float);

int 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,
    0);

imsls_random_seed_set(123457);

r = imsls_f_random_general_continuous (5, ndata, &table[0][0],
    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



[more...](#)

Sets up a 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`.

<code>iopt</code>	Action
0	<code>imsls_f_continuous_table_setup</code> fills the last four columns of <code>table</code> . The user inputs the points at which the CDF is to be evaluated in the first column of <code>table</code> . These must be in ascending order.
1	<code>imsls_f_continuous_table_setup</code> fills the last three columns of <code>table</code> . The user supplied function <code>cdf</code> is not used and may be a dummy function; instead, the cumulative distribution function is specified in the first two columns of <code>table</code> . 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/Output)

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** (Input)

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

Function `imsls_f_continuous_table_setup` sets up a table that function `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 function `imsls_f_random_general_continuous` for a description of the method.

On some platforms, `imsls_f_continuous_table_setup` can evaluate the user-supplied function `cdf` in parallel. This is done only if the function `imsls_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

Example

In this example, `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 function `imsls_f_random_general_continuous` to generate the random variates.

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

float cdf(float);

int main()
{
    int i, iopt=0, ndata= 100;
    float table[100][5], x = 0.0;

    imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    for (i=0;i<ndata;i++) {
        table[i][0] = x;
        x += .01;
    }

    imsls_f_continuous_table_setup(cdf, iopt, ndata, &table[0][0], 0);

    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

Fatal Errors

IMSL_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

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

`IMSLI_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 `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 1

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>

int 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	1.595
2	0.058	0.641
3	-0.867	-0.492
4	-0.933	-1.413
5	-0.325	-0.527

Example 2

Using the same variance-covariance matrix as above, `imsls_f_random_normal_multivariate` generates 10 pseudorandom normal vectors of length 2 in 2 blocks of 5. After resetting the random number generator, in this case the Mersenne Twister, `imsls_f_random_normal_multivariate` then generates all 10 random vectors at once. Because the generator is reset, the values in the third call match the combined values of the first two.

```
#include <imsls.h>

int main(){
    int seed = 123457, j;
    int n_vectors, l_vectors;
    float *r1 = NULL, *r2 = NULL, *r3 = NULL;
    float covariances[] = {1.0, 0.5, 0.5, 1.0};
    unsigned long long *itable;

    imsls_random_option(9);
    imsls_random_seed_set (seed);

    n_vectors = 5;
    l_vectors = 2;

    /* Generate the first matrix. */
    r1 = imsls_f_random_normal_multivariate (n_vectors, l_vectors,
        covariances, 0);

    printf("multivariate random normal deviates");
    imsls_f_write_matrix ("\nmatrix 1: ", n_vectors, l_vectors,
        r1, 0);

    /* Generate the second matrix.*/
    r2 = imsls_f_random_normal_multivariate (n_vectors, l_vectors,
        covariances, 0);

    imsls_f_write_matrix ("\nmatrix 2: ", n_vectors, l_vectors, r2, 0);

    /* Reset the generator. Setting itable[0] to a value > 625 resets
       the generator to its original state.*/
    imsls_random_MT64_table_get (&itable,0);
    itable[0] = 1000;
    imsls_random_MT64_table_set (itable);

    /* Generate all rows after resetting the generator. */
    n_vectors = 10;
```

```

r3 = imsls_f_random_normal_multivariate (n_vectors, l_vectors,
covariances, 0);
imsls_f_write_matrix ("\nmatrix 3: ", n_vectors, l_vectors, r3, 0);

imsls_free(r1);
imsls_free(r2);
imsls_free(r3);
imsls_free(itable);
}

```

Output

multivariate random normal deviates

matrix 1:

	1	2
1	1.321	0.598
2	0.055	1.050
3	-0.546	-1.876
4	0.724	1.548
5	-0.591	0.764

matrix 2:

	1	2
1	-0.0288	-0.7060
2	-0.9761	-0.0418
3	0.8074	-0.8965
4	-0.6163	-0.7335
5	-0.4368	0.0183

matrix 3:

	1	2
1	1.321	0.598
2	0.055	1.050
3	-0.546	-1.876
4	0.724	1.548
5	-0.591	0.764
6	-0.029	-0.706
7	-0.976	-0.042
8	0.807	-0.896
9	-0.616	-0.734
10	-0.437	0.018

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 `imsls_free`.

Synopsis with Optional Arguments

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

```
float *imsls_f_random_orthogonal_matrix(int n,  
    IMSLS_EIGENVALUES, float *eigenvalues[],  
    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

Function `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 $\mathbf{A}^T \mathbf{D} \mathbf{A}$, where $\mathbf{D} = \text{diag}(\text{eigenvalues}[0], \dots, \text{eigenvalues}[\mathbf{n}-1])$, 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 <imsls.h>

int 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 `imsls_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 $n_random \times ndim$ 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.) Function `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,

$$x_{j_1}, x_{j_2}, \dots, x_{j_m}$$

are determined; and the mean

$$\bar{x}_j$$

of those nearest neighbors is calculated. Next, a random sample u_1, u_2, \dots, 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

$$\sum_{l=1}^m u_l (x_{jl} - \bar{x}_j) + \bar{x}_j$$

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 function [imsls_f_data_sets](#), Chapter 15, [Utilities](#).)

```
#include <imsls.h>

int main()
{
    int i, nrow, ncol, 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, &nrow,
        IMSLS_N_VARIABLES, &ncol,
        IMSLS_RETURN_USER, rdata,
        0);

    for (i=0;i<nrow;i++) {
        x[i][0] = rdata[i][6];
        x[i][1] = rdata[i][9];
        x[i][2] = rdata[i][20];
        x[i][3] = rdata[i][23];
    }

    r = imsls_f_random_mvar_from_data(nr, k, nsamp, &x[0][0], 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 `imsls_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

Function `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

$$f(x_1, x_2, \dots, x_k) = \frac{n!}{x_1! x_2! \dots x_k!} p_1^{x_1} p_2^{x_2} \dots p_k^{x_k}$$

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, \dots, x_{j-2} with parameters $n - x_0 - x_1 - \dots - x_{j-2}$ and $p_j / (1 - p_0 - p_1 - \dots - p_{j-2})$.

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 <imsls.h>

int 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 `imsls_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

Function `imsls_f_random_sphere` generates pseudorandom coordinates of points that lie on a unit circle or a unit sphere in \mathcal{K} -dimensional space. For points on a circle ($\mathcal{K} = 2$), 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, \mathcal{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 <imsls.h>

int 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. 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 **imsls_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

`IMSL_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

Function `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 <imsls.h>

int 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_mvar_gaussian_copula

Given a Cholesky factorization of a correlation matrix, generates pseudorandom numbers from a Gaussian Copula distribution.

Synopsis

```
#include <imsls.h>

float *imsls_f_random_mvar_gaussian_copula(int n, float chol[], ..., 0)
```

The type *double* function is `imsls_d_random_mvar_gaussian_copula`.

Required Arguments

int n (Input)
Number of random numbers to generate.

float chol[] (Input)
Array of size $n \times n$ containing the upper-triangular Cholesky factorization of the correlation matrix of order n .

Return Value

An array of length n containing the pseudorandom numbers from a multivariate Gaussian Copula distribution.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_random_mvar_gaussian_copula (int n, float chol[],
      IMSLS_RETURN_USER, r[],
      0)
```

Optional Arguments

`IMSLS_RETURN_USER, r[]` (Output)

User-supplied array of length `n` containing the pseudorandom numbers from a multivariate Gaussian Copula distribution.

Description

Function `imsls_f_random_mvar_gaussian_copula` generates pseudorandom numbers from a multivariate Gaussian Copula distribution which are uniformly distributed on the interval (0,1) representing the probabilities associated with standard normal $N(0,1)$ deviates imprinted with correlation information from input upper-triangular Cholesky matrix `chol`. Cholesky matrix `chol` is defined as the “square root” of a user-defined correlation matrix, that is `chol` is an upper triangular matrix such that the transpose of `chol` \times `chol` is the correlation matrix. First, a length `n` array of independent random normal deviates with mean 0 and variance 1 is generated, and then this deviate array is post-multiplied by Cholesky matrix `chol`. Finally, the Cholesky-imprinted random $N(0,1)$ deviates are mapped to output probabilities using the $N(0,1)$ cumulative distribution function (CDF).

Random deviates from arbitrary marginal distributions which are imprinted with the correlation information contained in Cholesky matrix `chol` can then be generated by inverting the output probabilities using user-specified inverse CDF functions.

Example: Using Gaussian Copulas to Imprint and Extract Correlation Information

This example uses function `imsls_f_random_mvar_gaussian_copula` to generate a multivariate sequence `gcdevt` whose marginal distributions are user-defined and imprinted with a user-specified input correlation matrix `corrin` and then uses function `imsls_f_canonical_correlation` to extract an output canonical correlation matrix `corrout` from this multivariate random sequence.

This example illustrates two useful copula related procedures. The first procedure generates a random multivariate sequence with arbitrary user-defined marginal deviates whose dependence is specified by a user-defined correlation matrix. The second procedure is the inverse of the first: an arbitrary multivariate deviate input sequence is first mapped to a corresponding sequence of empirically derived variates, i.e. cumulative distribution function values representing the probability that each random variable has a value less than or equal to the input deviate. The variates are then inverted, using the inverse standard normal CDF function, to $N(0,1)$ deviates; and finally, a canonical covariance matrix is extracted from the multivariate $N(0,1)$ sequence using the standard sum of products.

This example demonstrates that function `imsls_f_random_mvar_gaussian_copula` correctly embeds the user-defined correlation information into an arbitrary marginal distribution sequence by extracting the canonical correlation from these sequences and showing that they differ from the original correlation matrix by a small relative error, which generally decreases as the number of multivariate sequence vectors increases.

```

#include <imsls.h>
#include <imsl.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NVAR 3

int main()
{
    int lmax=15000, i, j, k, kmax, kk;
    float chol[NVAR*NVAR], gcvart[NVAR], *gcdevt, corROUT[NVAR*NVAR],
        relerr, arg1=10.0, arg2=15.0, rs, rs00;
    float corrin[] = {
        1.0, -0.9486832, 0.8164965,
        -0.9486832, 1.0, -0.6454972,
        0.8164965, -0.6454972, 1.0
    };

    printf("Off-diagonal elements of Input Correlation Matrix:\n\n");

    for (i = 1; i < NVAR; i++) {
        for (j = 0; j < i; j++) {
            printf(" CorrIn(%d,%d) = %10.6f\n",
                i, j, corrin[i*NVAR + j]);
        }
    }

    printf("\nOff-diagonal elements of Output Correlation Matrices\n");
    printf("calculated from Gaussian Copula imprinted multivariate\n");
    printf("sequence:\n");
    /*
    * Compute the Cholesky factorization of corrin
    *
    * Use IMSL function imsl_f_lin_sol_posdef to generate
    * the NVAR by NVAR upper triangular matrix chol from
    * the Cholesky decomposition R*RT of input correlation
    * matrix corrin:
    */
    imsl_f_lin_sol_posdef (NVAR, corrin, NULL,
        IMSL_FACTOR_USER, chol,
        IMSL_FACTOR_ONLY,
        0);

    kmax = lmax / 100;

    for (kk = 1; kk <= 3; kk++) {
        gcdevt = (float *) malloc(kmax * NVAR * sizeof(float));

        printf("\n# of vectors in multivariate sequence: %7d\n\n",
            kmax);

        /* use Congruential RN generator, with multiplier 16807 */

```

```

imsls_random_option(1);

/* set RN generator seed to be 123457 */
imsls_random_seed_set(123457);

for (k = 0; k < kmax; k++) {
  /*
   * generate a NVAR-length random Gaussian Copula
   * variate output vector gcvart which is uniformly
   * distributed on the interval [0,1] and imprinted
   * with correlation information from input Cholesky
   * matrix chol:
   */
  imsls_f_random_mvar_gaussian_copula(NVAR, chol,
    IMSLS_RETURN_USER, gcvart,
    0);

  for (j = 0; j < 3; j++) {
    /*
     * invert Gaussian Copula probabilities to deviates
     * using variable-specific inversions: j = 0: Chi
     * Square; j = 1: F; j = 2: Normal(0,1); will end
     * up with deviate sequences ready for mapping to
     * canonical correlation matrix:
     */
    if (j == 0) {
      /* convert probs into ChiSquare(df=10) deviates */
      gcdevt[k*NVAR + j] =
        imsls_f_chi_squared_inverse_cdf(gcvart[j], arg1);

    } else if (j == 1) {
      /* convert probs into F(dfn=15,dfd=10) deviates */
      gcdevt[k*NVAR + j] =
        imsls_f_F_inverse_cdf(gcvart[j], arg2, arg1);

    } else {
      /*
       * convert probs into Normal(mean=0,variance=1)
       * deviates:
       */
      gcdevt[k*NVAR + j] =
        imsls_f_normal_inverse_cdf(gcvart[j]);
    }
  }
}

/*
 * extract Canonical Correlation matrix from arbitrarily
 * distributed deviate sequences gcdevt (k=1..kmax, j=1..NVAR)
 * which have been imprinted with corrin (i=1..NVAR, j=1..NVAR)
 * above:
 */
imsls_f_canonical_correlation(kmax, NVAR, gcdevt,
  IMSLS_RETURN_USER, corrout,
  0);

for (i = 1; i < NVAR; i++) {
  for (j = 0; j <= i-1; j++) {
    rs00 = corrin[i*NVAR + j];
    rs = corrout[i*NVAR + j];
  }
}

```

```

        relerr = fabs((rs - rs00)/rs00);
        printf(" CorrOut(%d,%d) = %10.6f; relerr = %10.6f\n",
              i, j, corROUT[i*NVAR + j], relerr);
    }
}
free(gcdevt);
kmax *= 10;
}
}

```

Output

Off-diagonal elements of Input Correlation Matrix:

```

CorrIn(1,0) = -0.948683
CorrIn(2,0) =  0.816496
CorrIn(2,1) = -0.645497

```

Off-diagonal elements of Output Correlation Matrices
calculated from Gaussian Copula imprinted multivariate
sequence:

of vectors in multivariate sequence: 150

```

CorrOut(1,0) = -0.940215; relerr =  0.008926
CorrOut(2,0) =  0.794511; relerr =  0.026927
CorrOut(2,1) = -0.616082; relerr =  0.045570

```

of vectors in multivariate sequence: 1500

```

CorrOut(1,0) = -0.947444; relerr =  0.001306
CorrOut(2,0) =  0.808306; relerr =  0.010031
CorrOut(2,1) = -0.635650; relerr =  0.015255

```

of vectors in multivariate sequence: 15000

```

CorrOut(1,0) = -0.948263; relerr =  0.000443
CorrOut(2,0) =  0.817261; relerr =  0.000936
CorrOut(2,1) = -0.646206; relerr =  0.001098

```

random_mvar_t_copula

Given a Cholesky factorization of a correlation matrix, generates pseudorandom numbers from a Student's t Copula distribution.

Synopsis

```
#include <imsls.h>

float *imsls_f_random_mvar_t_copula(float df, int n, float chol[], ..., 0)
```

The type *double* function is `imsls_d_random_mvar_t_copula`.

Required Arguments

float **df** (Input)
Degrees of freedom. **df** must be greater than 2.

int **n** (Input)
Number of random numbers to generate.

float **chol[]** (Input)
An array of size $n \times n$ containing the upper-triangular Cholesky factorization of the correlation matrix of order **n**.

Return Value

An array of length **n** containing the pseudorandom numbers from a multivariate Student's t Copula distribution.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_random_mvar_t_copula (float df, int n, float chol[],
    IMSLS_RETURN_USER, r[],
    0)
```

Optional Arguments

`IMSLS_RETURN_USER, r[]` (Output)

User-supplied array of length `n` containing the pseudorandom numbers from a multivariate Student's t Copula distribution.

Description

Function `imsls_f_random_mvar_t_copula` generates pseudorandom numbers from a multivariate Student's t Copula distribution which are uniformly distributed on the interval (0,1) representing the probabilities associated with Student's t deviates with `df` degrees of freedom imprinted with correlation information from input upper-triangular Cholesky matrix `chol`. Cholesky matrix `chol` is defined as the "square root" of a user-defined correlation matrix. That is, `chol` is an upper triangular matrix such that the transpose of `chol` times `chol` is the correlation matrix. First, a length `n` array of independent random normal deviates with mean 0 and variance 1 is generated, and then this deviate array is post-multiplied by Cholesky matrix `chol`. Each of the `n` elements of the resulting vector of Cholesky-imprinted random deviates is then divided by

$$\sqrt{s/\nu},$$

where $\nu = \text{df}$ and s is a random deviate taken from a chi-squared distribution with `df` degrees of freedom. Each element of the Cholesky-imprinted standard normal $N(0,1)$ array is a linear combination of normally distributed random numbers and is therefore itself normal, and the division of each element by

$$\sqrt{s/\nu}$$

insures that each element of the resulting array is Student's t distributed. Finally, each element of the Cholesky-imprinted Student's t array is mapped to an output probability using the Student's t cumulative distribution function (CDF) with `df` degrees of freedom.

Random deviates from arbitrary marginal distributions which are imprinted with the correlation information contained in Cholesky matrix `chol` can then be generated by inverting the output probabilities using user-specified inverse CDF functions.

Example: Using Student's t Copulas to Imprint and Extract Correlation Information

This example uses function `imsls_f_random_mvar_t_copula` to generate a multivariate sequence `tcdevt` whose marginal distributions are user-defined and imprinted with a user-specified input correlation matrix `corrin` and then uses function `imsls_f_canonical_correlation` to extract an output canonical correlation matrix `corrout` from this multivariate random sequence.

This example illustrates two useful copula related procedures. The first procedure generates a random multivariate sequence with arbitrary user-defined marginal deviates whose dependence is specified by a user-defined correlation matrix. The second procedure is the inverse of the first: an arbitrary multivariate deviate input sequence is first mapped to a corresponding sequence of empirically derived variates, i.e. cumulative distribution function values representing the probability that each random variable has a value less than or equal to the input deviate. The variates are then inverted, using the inverse standard normal CDF function, to $N(0,1)$ deviates; and finally, a canonical covariance matrix is extracted from the multivariate $N(0,1)$ sequence using the standard sum of products.

This example demonstrates that function `imsls_f_random_mvar_t_copula` correctly imbeds the user-defined correlation information into an arbitrary marginal distribution sequence by extracting the canonical correlation from these sequences and showing that they differ from the original correlation matrix by a small relative error.

Recall that a Gaussian Copula array sequence, whose probabilities are mapped directly from Cholesky-imprinted $N(0,1)$ deviates, has the property that the relative error between the input and output correlation matrices generally decreases as the number of multivariate sequence vectors increases. This is understandable because the correlation imprinting and extraction processes both act upon $N(0,1)$ marginal distributions, and one would expect that a larger sample would therefore result in more accurate imprinting and extraction of correlation information.

In contrast, the imprinting of correlation information onto the Student's t vector sequence is accomplished by imprinting onto an $N(0,1)$ array and then dividing the array components by a scaled chi-squared random deviate, thereby introducing noise into the imprinting process. (An array of Student's t deviates cannot be Cholesky-imprinted directly, because a linear combination of Student's t deviates is not Student's t distributed.) A larger sample would thus contain additional correlation information and additional noise, so the accuracy would be expected to plateau. This is illustrated in the example below, which should be compared with the Gaussian Copula example given for CNL function [imsls_f_random_mvar_gaussian_copula](#).

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

#define NVAR 3
```

```

int main()
{
    int lmax=15000, i, j, k, kmax, kk;
    float chol[NVAR*NVAR], tcvart[NVAR], *tcdevt, corROUT[NVAR*NVAR],
        df=5.0, relerr, arg1=10.0, arg2=15.0, rs, rs00;

    float corrin[] = {
        1.0, -0.9486832, 0.8164965,
        -0.9486832, 1.0, -0.6454972,
        0.8164965, -0.6454972, 1.0
    };

    printf("Off-diagonal elements of Input Correlation Matrix:\n\n");

    for (i = 1; i < NVAR; i++) {
        for (j = 0; j < i; j++) {
            printf(" CorrIn(%d,%d) = %10.6f\n",
                i, j, corrin[i*NVAR + j]);
        }
    }

    printf("\n Degrees of freedom df = %6.2f\n", df);
    printf("\n Imprinted random sequences distributions:");
    printf("\n 1: Chi, 2: F, 3: Normal;\n");

    printf("\nOff-diagonal elements of Output Correlation Matrices\n");
    printf("calculated from Student's t Copula imprinted\n");
    printf("multivariate sequence:\n");

    /*
    * Compute the Cholesky factorization of corrin
    */
    /* Use IMSL function imsl_f_lin_sol_posdef to generate
    * the NVAR by NVAR upper triangular matrix chol from
    * the Cholesky decomposition R*RT of input correlation
    * matrix corrin:
    */
    imsl_f_lin_sol_posdef (NVAR, corrin, NULL,
        IMSL_FACTOR_USER, chol,
        IMSL_FACTOR_ONLY,
        0);

    kmax = lmax / 100;

    for (kk = 1; kk <= 3; kk++) {
        tcdevt = (float *) malloc(kmax * NVAR * sizeof(float));

        printf("\n# of vectors in multivariate sequence: %7d\n\n",
            kmax);

        /* use Congruential RN generator, with multiplier 16807 */
        imsls_random_option(1);

        /* set RN generator seed to be 123457 */
        imsls_random_seed_set(123457);

        for (k = 0; k < kmax; k++) {
            /*
            * generate a NVAR-length random Student's t Copula

```

```

* variate output vector tcvart which is uniformly
* distributed on the interval [0,1] and imprinted
* with correlation information from input Cholesky
* matrix chol:
*/
imsls_f_random_mvar_t_copula(df, NVAR, chol,
    IMSLS_RETURN_USER, tcvart,
    0);

for (j = 0; j < 3; j++) {
    /*
    * invert Student's t Copula probabilities to
    * deviates using variable-specific
    * inversions: j = 0: Chi Square; j = 1: F;
    * j = 2: Normal(0,1); will end up with deviate
    * sequences ready for mapping to canonical
    * correlation matrix:
    */
    if (j == 0) {
        /* convert probs into ChiSquare(df=10) deviates */
        tcdevt[k*NVAR + j] =
            imsls_f_chi_squared_inverse_cdf(tcvart[j], arg1);

    } else if (j == 1) {
        /* convert probs into F(dfn=15,dfd=10) deviates */
        tcdevt[k*NVAR + j] =
            imsls_f_F_inverse_cdf(tcvart[j], arg2, arg1);

    } else {
        /*
        * convert probs into Normal(mean=0,variance=1)
        * deviates:
        */
        tcdevt[k*NVAR + j] =
            imsls_f_normal_inverse_cdf(tcvart[j]);
    }
}

/*
* extract Canonical Correlation matrix from arbitrarily
* distributed deviate sequences tcdevt (k=1..kmax, j=1..NVAR)
* which have been imprinted with corrin (i=1..NVAR, j=1..NVAR)
* above:
*/
imsls_f_canonical_correlation (kmax, NVAR, tcdevt,
    IMSLS_RETURN_USER, corROUT,
    0);

for (i = 1; i < NVAR; i++) {
    for (j = 0; j <= i-1; j++) {
        rs00 = corrin[i*NVAR + j];
        rs = corROUT[i*NVAR + j];
        relerr = fabs((rs - rs00)/rs00);
        printf(" CorrOut(%d,%d) = %10.6f; relerr = %10.6f\n",
            i, j, corROUT[i*NVAR + j], relerr);
    }
}
free (tcdevt);
kmax *= 10;

```

```
}  
}
```

Output

Off-diagonal elements of Input Correlation Matrix:

```
CorrIn(1,0) = -0.948683  
CorrIn(2,0) =  0.816496  
CorrIn(2,1) = -0.645497
```

Degrees of freedom df = 5.00

Imprinted random sequences distributions:

1: Chi, 2: F, 3: Normal;

Off-diagonal elements of Output Correlation Matrices
calculated from Student's t Copula imprinted
multivariate sequence:

of vectors in multivariate sequence: 150

```
CorrOut(1,0) = -0.953573; relerr = 0.005154  
CorrOut(2,0) =  0.774720; relerr = 0.051166  
CorrOut(2,1) = -0.621418; relerr = 0.037303
```

of vectors in multivariate sequence: 1500

```
CorrOut(1,0) = -0.944316; relerr = 0.004603  
CorrOut(2,0) =  0.810164; relerr = 0.007756  
CorrOut(2,1) = -0.636348; relerr = 0.014174
```

of vectors in multivariate sequence: 15000

```
CorrOut(1,0) = -0.946770; relerr = 0.002017  
CorrOut(2,0) =  0.808564; relerr = 0.009715  
CorrOut(2,1) = -0.636321; relerr = 0.014216
```

canonical_correlation



[more...](#)

Given an input array of deviate values, generates a canonical correlation array.

Synopsis

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

```
float *imsls_f_canonical_correlation(int nseq, int nvar, float devt[ ], ..., 0)
```

The type *double* function is `imsls_d_canonical_correlation`.

Required Arguments

int **nseq** (Input)

Number of steps in each deviate variable sequence.

int **nvar** (Input)

Number of deviate variables.

float **devt[]** (Input)

An array of length `nseq × nvar` of deviate values containing `nseq` row elements for each of `nvar` variables (columns).

Return Value

An array of length `nvar × nvar` containing the canonical correlation array.

Synopsis with Optional Arguments

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

```
float *imsls_f_canonical_correlation(int nseq, int nvar, float devt[ ],
    IMSLS_RETURN_USER, corr[ ],
    0)
```

Optional Arguments

`IMSL_RETURN_USER, corr[]` (Output)

User-supplied array of length `nvar × nvar` containing canonical correlation array.

Description

Function `imsls_f_canonical_correlation` generates a canonical correlation matrix from an arbitrarily distributed multivariate deviate sequence `devt` with `nvar` deviate variables, `nseq` elements in each deviate sequence, and a Gaussian Copula dependence structure.

Function `imsls_f_canonical_correlation` first maps each of the $j = 0, \dots, \text{nvar}-1$ input deviate sequences `devt`[$k = 0, \dots, \text{nseq}-1$][j] into a corresponding sequence of variates, say V_{kj} (where variates are values of the empirical cumulative probability function, $CDF(x)$, defined as the probability that random deviate variable $X \leq x$). The variate matrix element V_{kj} is then mapped into standard normal $N(0,1)$ distributed deviates z_{kj} using the inverse standard normal CDF `imsls_f_normal_inverse_cdf`(V_{kj}) and then the standard covariance estimator

$$C_{ij} = \frac{1}{m} \sum_{k=1}^m z_{ki} z_{kj}$$

(where $m = \text{nseq}$ and i and j have values between 1 and `nvar`) is used to calculate the canonical correlation matrix `corr`, where $C_{ij} = \text{corr}[i-1][j-1]$ = the return value canonical correlation array.

If a multivariate distribution has Gaussian marginal distributions, then the standard “empirical” correlation matrix given above is “unbiased”, i.e. an accurate measure of dependence among the variables. But when the marginal distributions depart significantly from Gaussian, i.e. are skewed or flattened, then the empirical correlation may become biased. One way to remove such bias from dependence measures is to map the non-Gaussian-distributed marginal deviates to $N(0,1)$ deviates (by mapping the non-Gaussian marginal deviates to empirically derived marginal CDF variate values, then inverting the variates to $N(0,1)$ deviates as described above), and calculating the standard empirical correlation matrix from these $N(0,1)$ deviates as in the equation above. The resulting “canonical correlation” matrix thereby avoids the bias that would occur if the empirical correlation matrix were extracted from the non-Gaussian marginal distributions directly.

The canonical correlation matrix may be of value in such applications as Markowitz portfolio optimization, where an unbiased measure of dependence is required to evaluate portfolio risk, defined in terms of the portfolio variance which is in turn defined in terms of the correlation among the component portfolio instruments.

The utility of the canonical correlation derives from the observation that a “copula” multivariate distribution with uniformly-distributed deviates (corresponding to the CDF probabilities associated with the marginal deviates) may be mapped to arbitrarily distributed marginals, so that an unbiased dependence estimator derived from one set

of marginals $N(0,1)$ (distributed marginals) can be used to represent the dependence associated with arbitrarily-distributed marginals. The “Gaussian Copula” (whose variate arguments are derived from $N(0,1)$ marginal deviates) is a particularly useful structure for representing multivariate dependence.

Example: Using Gaussian Copulas to Imprint and Extract Correlation Information

This example uses function `imsls_f_random_mvar_gaussian_copula` to generate a multivariate sequence `gcdevt` whose marginal distributions are user-defined and imprinted with a user-specified input correlation matrix `corrin` and then uses function `imsls_f_canonical_correlation` to extract an output canonical correlation matrix `corrout` from this multivariate random sequence.

This example illustrates two useful copula related procedures. The first procedure generates a random multivariate sequence with arbitrary user-defined marginal deviates whose dependence is specified by a user-defined correlation matrix. The second procedure is the inverse of the first: an arbitrary multivariate deviate input sequence is first mapped to a corresponding sequence of empirically derived variates, i.e. cumulative distribution function values representing the probability that each random variable has a value less than or equal to the input deviate. The variates are then inverted, using the inverse standard normal CDF function, to $N(0,1)$ deviates; and finally, a canonical covariance matrix is extracted from the multivariate $N(0,1)$ sequence using the standard sum of products.

This example demonstrates that function `imsls_f_random_mvar_gaussian_copula` correctly embeds the user-defined correlation information into an arbitrary marginal distribution sequence by extracting the canonical correlation from these sequences and showing that they differ from the original correlation matrix by a small relative error, which generally decreases as the number of multivariate sequence vectors increases.

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

#define NVAR 3

int main()
{
    int lmax=15000, i, j, k, kmax, kk;
    float chol[NVAR*NVAR], gcvart[NVAR], *gcdevt, corrout[NVAR*NVAR],
        relerr, arg1=10.0, arg2=15.0, rs, rs00;
    float corrin[] = {
        1.0, -0.9486832, 0.8164965,
        -0.9486832, 1.0, -0.6454972,
        0.8164965, -0.6454972, 1.0
    };

    printf("Off-diagonal elements of Input Correlation Matrix:\n\n");
```

```

for (i = 1; i < NVAR; i++) {
    for (j = 0; j < i; j++) {
        printf(" CorrIn(%d,%d) = %10.6f\n",
            i, j, corrin[i*NVAR + j]);
    }
}

printf("\nOff-diagonal elements of Output Correlation Matrices\n");
printf("calculated from Gaussian Copula imprinted multivariate\n");
printf("sequence:\n");
/*
* Compute the Cholesky factorization of corrin
*
* Use IMSL function imsl_f_lin_sol_posdef to generate
* the NVAR by NVAR upper triangular matrix chol from
* the Cholesky decomposition R*RT of input correlation
* matrix corrin:
*/
imsl_f_lin_sol_posdef (NVAR, corrin, NULL,
    IMSL_FACTOR_USER, chol,
    IMSL_FACTOR_ONLY,
    0);

kmax = lmax / 100;

for (kk = 1; kk <= 3; kk++) {
    gcdevt = (float *) malloc(kmax * NVAR * sizeof(float));

    printf("\n# of vectors in multivariate sequence: %7d\n\n",
        kmax);

    /* use Congruential RN generator, with multiplier 16807 */
    imsls_random_option(1);

    /* set RN generator seed to be 123457 */
    imsls_random_seed_set(123457);

    for (k = 0; k < kmax; k++) {
        /*
        * generate a NVAR-length random Gaussian Copula
        * variate output vector gcvart which is uniformly
        * distributed on the interval [0,1] and imprinted
        * with correlation information from input Cholesky
        * matrix chol:
        */
        imsls_f_random_mvar_gaussian_copula(NVAR, chol,
            IMSLS_RETURN_USER, gcvart,
            0);

        for (j = 0; j < 3; j++) {
            /*
            * invert Gaussian Copula probabilities to deviates
            * using variable-specific inversions: j = 0: Chi
            * Square; j = 1: F; j = 2: Normal(0,1); will end
            * up with deviate sequences ready for mapping to
            * canonical correlation matrix:
            */
            if (j == 0) {
                /* convert probs into ChiSquare(df=10) deviates */
                gcdevt[k*NVAR + j] =

```

```

        imsls_f_chi_squared_inverse_cdf(gcvar[j], arg1);
    } else if (j == 1) {
        /* convert probs into F(dfn=15,dfd=10) deviates */
        gcdevt[k*NVAR + j] =
            imsls_f_F_inverse_cdf(gcvar[j], arg2, arg1);
    } else {
        /*
        * convert probs into Normal(mean=0,variance=1)
        * deviates:
        */
        gcdevt[k*NVAR + j] =
            imsls_f_normal_inverse_cdf(gcvar[j]);
    }
}
}

/*
* extract Canonical Correlation matrix from arbitrarily
* distributed deviate sequences gcdevt (k=1..kmax, j=1..NVAR)
* which have been imprinted with corrin (i=1..NVAR, j=1..NVAR)
* above:
*/
imsls_f_canonical_correlation(kmax, NVAR, gcdevt,
    IMSLS_RETURN_USER, corROUT,
    0);

for (i = 1; i < NVAR; i++) {
    for (j = 0; j <= i-1; j++) {
        rs00 = corrin[i*NVAR + j];
        rs = corROUT[i*NVAR + j];
        relerr = fabs((rs - rs00)/rs00);
        printf(" CorrOut(%d,%d) = %10.6f; relerr = %10.6f\n",
            i, j, corROUT[i*NVAR + j], relerr);
    }
}
free(gcdevt);
kmax *= 10;
}
}

```

Outputs

Off-diagonal elements of Input Correlation Matrix:

```

CorrIn(1,0) = -0.948683
CorrIn(2,0) =  0.816496
CorrIn(2,1) = -0.645497

```

Off-diagonal elements of Output Correlation Matrices
calculated from Gaussian Copula imprinted multivariate
sequence:

of vectors in multivariate sequence: 150

```

CorrOut(1,0) = -0.940215; relerr =  0.008926
CorrOut(2,0) =  0.794511; relerr =  0.026927

```

CorrOut(2,1) = -0.616082; relerr = 0.045570

of vectors in multivariate sequence: 1500

CorrOut(1,0) = -0.947444; relerr = 0.001306

CorrOut(2,0) = 0.808306; relerr = 0.010031

CorrOut(2,1) = -0.635650; relerr = 0.015255

of vectors in multivariate sequence: 15000

CorrOut(1,0) = -0.948263; relerr = 0.000443

CorrOut(2,0) = 0.817261; relerr = 0.000936

CorrOut(2,1) = -0.646206; relerr = 0.001098

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 `imsls_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

Function `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. Function `imsls_f_random_order_normal` uses the function `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>

int 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 `imsls_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

Function `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>

int 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. Parameter **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, or
    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`

or

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[l])` 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[l])` 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[l])` 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_observations` containing the generated time series.

Description

Function `imsls_f_random_arma` simulates an ARMA(p, q) process, $\{W_t\}$, for $t = 1, 2, \dots, n$ (with $n = n_observations$, $p = p$, and $q = q$). The model is

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

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \dots - \phi_p B^p$$

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \dots - \theta_q B^q$$

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 following model:

$$X[i] = \text{constant} + \text{ar}[0] \cdot X[i - \text{ar_lags}[0]] + \dots + \\ \text{ar}[p - 1] \cdot X[i - \text{ar_lags}[p - 1]] +$$

$$A[i] - \text{ma}[0] \cdot A[i - \text{ma_lags}[0]] - \dots - \text{ma}[q - 1] \cdot A[i - \text{ma_lags}[q - 1]]$$

where the constant is related to the mean of the series,

$$\bar{W}$$

as follows:

$$\text{constant} = \bar{W} \cdot (1 - \text{ar}[0] - \dots - \text{ar}[q - 1])$$

and where

$$X[t] = W[t], t = 0, 1, \dots, n_{\text{observations}} - 1$$

and

$$W[t] = w_{\text{initial}}[t + p], \quad t = -p, -p + 1, \dots, -2, -1$$

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.

```
#include <imsls.h>

int 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 3 autoregressive parameters and 2 moving average parameters. The start values are 0.1, 0.05 and 0.0375. Constant and noise are also input.

```

#include <imsls.h>

int 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 than *neub*, the time *tend* is reached before *neub* events are realized.

Return Value

An array of length *neub* containing the times to events in the first *ne* elements. To release this space, use `imsls_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,  
IMSL_RETURN_USER, float r[ ],  
IMSL_FCN_W_DATA, float ftheta ( ), void *data,  
0)
```

Optional Arguments

`IMSL_RETURN_USER, float r[]` (Output)

User-supplied array of length *neub* containing the times to events in the first *ne* elements.

`IMSL_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 [Passing Data to User-Supplied Functions](#) at the beginning of this manual for more details.

Description

Function `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 $t_0 = \mathbf{tbegin}$ and $t_1 = \mathbf{tend}$. Function `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

$$A(t) = \int_0^t \lambda^*(t) dt$$

is called the *integrated rate function*. In `imsls_f_random_npp`, $\lambda^*(t)$ is taken to be a constant $\lambda^*(= \mathbf{theta_max})$ so that at time t_i , the time of the next event t_{i+1} is obtained by generating and cumulating exponential random numbers

$$E_{1,i}^*, E_{2,i}^*, \dots$$

with parameter λ^* , until for the first time

$$u_{j,i} \leq (t_i + E_{1,i}^* + \dots + E_{j,i}^*) / \lambda^*$$

where the $u_{j,i}$ are independent uniform random numbers between 0 and 1. This process is continued until the specified number of events, \mathbf{neub} , is realized or until the time, \mathbf{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 $\mathbf{theta_max}$ is actually greater than the maximum of $\lambda(t)$ in $(t_0, t_1]$, the function 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, \mathbf{tend} must be set large enough to allow for the required number of events to be realized. Care must be taken; however, that \mathbf{ftheta} is defined over the entire interval.

After simulating a given number of events, the next event can be generated by setting \mathbf{tbegin} to the time of the last event (the sum of the elements in \mathbf{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

$$\lambda(t) = 0.6342 \exp(0.001427t)$$

for $0 < t \leq 20$.

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 $0.6342 \exp(0.02854) = 0.652561$.

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

float ftheta (float t);

int 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 ");
    printf("%d events in the process:\n", ne);

    for (i=0; i<ne; i++) printf("\t%f\n", r[i]);
}

float ftheta (float t)
{
    return 0.6342*exp(0.001427*t);
}
```

Output

```
Inter-event times for the first 5 events in the process:
0.052660
0.407979
0.258399
0.019767
0.167641
```

Fatal Errors

IMSL_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

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 `imsls_free`.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_random_permutation(int k,
                               IMSLS_RETURN_USER, int ir[],
                               0)
```

Optional Arguments

`IMSL_RETURN_USER, int ir[]` (Output)
User-supplied array of length *k* containing the random permutation of the integers from 1 to *k*.

Description

Function `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>

int 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 **imsls_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

Function `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. Function `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 <imsls.h>

int 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 `imsls_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

Function `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.

Examples

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 <imsls.h>

int 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

Function `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 `imsls_f_random_sample` one observation at a time. This is the way `imsls_f_random_sample` may be used to sample from a large data file. Notice that the number of records need not be known in advance.

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

int 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.

<code>generator_option</code>	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 $X_t = X_{t-1563} \oplus X_{t-96}$ 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:

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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.

	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 $X_t = X_{t-1563} \oplus X_{t-96}$ is used

Description

The function `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

int 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 izeed1)
```

Required Arguments

int izeed1 (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 *izeed1*.

Description

Given a seed, *izeed1*, *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 *izeed1*, 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 function *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 *izeed1* 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>
#include <stdio.h>

int 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)
            imsls_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 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 function 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_f_random_table_get (float **table,  
    IMSLS_RETURN_USER, float r[ ],  
    0)
```

Optional Arguments

`IMLS_RETURN_USER, float r[]` (Output)

User-supplied array of length 128 containing the table used in the shuffled generators.

Description

Function `imsls_f_random_table_get` retrieves the current table used in the shuffled generator. A reason for doing this would be to restart a simulation, using function `imsls_f_random_table_set` to reset the table. To restart a simulation using a shuffled generator, both the seed and the table must be reset (see

[random_GF5SR_table_get](#) example). 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.

Example

See function [ims1s_random_GF5SR_table_get](#).

random_GF5SR_table_set

Sets the current table used in the GF5SR generator.

Synopsis

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

```
void imsls_f_random_GF5SR_table_set (int table[ ])
```

Required Arguments

int **table** [] (Input)

Array of length 1565 used in the GF5SR 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_GF5SR_table_set`, on the next invocation of a function to generate random numbers using a GF5SR method, the appropriate array will be reinitialized.

Example

See function [imsls_random_GF5SR_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

IMSL_RETURN_USER, int r[] (Output)
User-supplied array of length 1565 containing the table used in the GFSR generators.

Description

Function `imsls_f_random_GFSR_table_get` retrieves the current table used in the GFSR generator. A reason for doing this would be to restart a simulation, using function `imsls_f_random_GFSR_table_set` to reset the table. To restart a simulation using a GFSR generator, both the seed and the table must be reset (see [example](#)). The tables for the GFSR 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 GFSR generators.

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

int 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);

    imsls_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);
```

```
    imsls_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);

    imsls_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);

    imsls_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);

    imsls_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);

imsls_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_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

IMLS_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 fourth and first streams are therefore the same.

```
#include <imsls.h>

int main()
{
    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);
    imsls_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);
    imsls_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);
    imsls_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);  
    imsls_free(r);  
    return 0;  
}
```

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_get`. 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_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

IMSL_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 fourth and first streams are therefore the same.

```
#include <imsls.h>

int main()
{
    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);
    imsls_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);
    imsls_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);
    imsls_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);  
    imsls_free(r);  
    return 0;  
}
```

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_get`. 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



[more...](#)

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: $\lfloor base^{m/2-1} \rfloor$, where $m = \lfloor \log B / \log base \rfloor$ 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 $x_1, \dots, x_n \in [0,1]^d$, $d \geq 1$, 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

$$E = [0, t_1) \times \dots \times [0, t_d), 0 \leq t_j \leq 1, 1 \leq j \leq d,$$

λ is the Lebesgue measure, and $A(E; n)$ is the number of the x_j contained in E .

The sequence x_1, x_2, \dots of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d , such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all $n > 1$.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the optional argument `IMSL_BASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence x_1, x_2, \dots , 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, $0 \leq a_i(n) < b$.

The j -th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n) b^{-k-1}, 1 \leq j \leq d$$

The generator matrix for the series, $c_{kd}^{(j)}$, is defined to be

$$c_{kd}^{(j)} = j^{d-k} c_{kd}$$

and c_{kd} is an element of the Pascal matrix,

$$c_{kd} = \begin{cases} \frac{d!}{c!(d-c)!} & k \leq d \\ 0 & k > d \end{cases}$$

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 "imsls.h"

int main()
{
    Imsls_faure *state;
    float *x;
    int ndim = 3;
    int k;

    state = imsls_faure_sequence_init(ndim, 0);

    for (k = 0; k < 5; k++) {
        x = imsls_f_faure_next_point(state, 0);
        printf("%10.3f %10.3f %10.3f\n", x[0], x[1], x[2]);
        imsls_free(x);
    }

    imsls_faure_sequence_free(state);
    return 0;
}
```

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

Data Mining

Functions

Apriori — Market Basket Analysis

Computes frequent itemsets	apriori	1394
Computes frequent itemsets using aggregation	aggr_apriori	1401
Prints an <code>lmsls_f_apriori_itemsets</code> data structure	write_apriori_itemsets	1412
Prints an <code>lmsls_f_association_rule</code> data structure	write_association_rules	1413
Frees memory allocated for an <code>lmsls_f_apriori_itemsets</code> data structure	free_apriori_itemsets	1414
Frees memory allocated for an <code>lmsls_f_association_rules</code> data structure	free_association_rules	1415

Decision Trees

<i>Decision Trees – An Overview</i>		1416
Generates a decision tree for a single response variable and two or more predictor variables	decision_tree	1419
Computes predicted values using a decision tree	decision_tree_predict	1455
Prints a decision tree	decision_tree_print	1462
Frees the memory associated with a decision tree	decision_tree_free	1467
Frees the memory associated with an array of decision trees	bagged_trees_free	1468
Performs stochastic gradient boosting of decision trees	gradient_boosting	1469

Genetic Algorithms

<i>Genetic Algorithms – An Overview</i>		1492
Genetic Algorithm Data Structures		
Creates a chromosome	ga_chromosome	1501
Copies one chromosome to another	ga_copy_chromosome	1507
Clones an existing chromosome	ga_clone_chromosome	1508
Creates an individual	ga_individual	1510
Copies the contents of one individual into another individual	ga_copy_individual	1515
Clones an existing individual	ga_clone_individual	1516
Applies mutation to an individual	ga_mutate	1518
Decodes an individual's chromosome into its phenotype	ga_decode	1520
Encodes an individual's phenotype into its chromosome	ga_encode	1521
Frees memory allocated to an individual	ga_free_individual	1522
Creates a population from an array of individuals	ga_population	1523

Creates a population of randomly selected individuals	ga_random_population	1530
Copies a population into an existing population	ga_copy_population	1539
Creates a copy of a population	ga_clone_population	1540
Add individuals to a population	ga_grow_population	1542
Creates a new population by merging two populations	ga_merge_population	1544
Frees memory allocated to a population	ga_free_population	1546
Genetic Algorithm Search and Optimization		
Applies a genetic algorithm to find individuals with maximum fitness.	genetic_algorithm	1547
Naive Bayes		
Naive Bayes – An Overview		1569
Trains a Naive Bayes classifier	naive_bayes_trainer	1571
Classifies patterns using a previously trained Naive Bayes classifier.	naive_bayes_classification	1590
Frees memory allocated for a Naive Bayes classifier	nb_classifier_free	1599
Writes a Naive Bayes classifier to an ASCII file	nb_classifier_write	1600
Retrieves a Naive Bayes classifier	nb_classifier_read	1605
Neural Networks		
Neural Networks – An Overview		1611
Neural Network Data Structures		
Multilayer Feedforward Neural Networks.		1622
Initializes a data structure for training a neural network	mlff_network_init	1632
Multilayered feedforward neural network.	mlff_network	1634
Frees memory allocated for an Imsls_f_NN_Network data structure.	mlff_network_free	1646
Writes a trained neural network to an ASCII file.	mlff_network_write	1647
Retrieves a neural network from a file previously saved.	mlff_network_read	1653
Initializes weights for neural network.	mlff_initialize_weights	1659
Forecasting Neural Networks		
Trains a multilayered feedforward neural network	mlff_network_trainer	1674
Calculates forecasts for trained multilayered feedforward neural networks.	mlff_network_forecast	1686
Classification Neural Networks		
Trains a neural network for classification.	mlff_classification_trainer	1693
Calculates classifications from a trained neural network	mlff_pattern_classification	1719

Preprocessing Data Filters

Encodes or decodes continuous input attributes	scale_filter	1734
Encodes a time series into lagged values	time_series_filter	1741
Encodes a time series into lagged values of a nominal classification attribute	time_series_class_filter	1744
Encodes or decodes a nominal input attributes	unsupervised_nominal_filter	1749
Encodes or decodes ordinal input attributes	unsupervised_ordinal_filter	1753

Self-Organizing Maps

Trains a Kohonen network	kohonenSOM_trainer	1758
Calculates forecasts using a trained Kohonen network	kohonenSOM_forecast	1769

Support Vector Machines

Support Vector Machines – An Overview		1771
Trains a Support Vector Machines classifier	support_vector_trainer	1773
Classifies patterns using a previously trained Support Vector Machines classifier	support_vector_classification	1791
Frees memory allocated for a Support Vector Machines classifier	svm_classifier_free	1796

Data Mining Usage Notes

Data mining is a collection of statistical and analytical methods for extracting useful information from large databases. The problem of extracting information from large databases is common to government, industry, engineering and sciences.

Apriori

The Apriori algorithm is used for association rule discovery. Association rules are statements of the form, "if X, then Y", given with some measure of confidence. The main application for association rule discovery is market basket analysis, in which X and Y are products or groups of products, and the occurrences are individual transactions, or "market baskets." The results help sellers learn relationships between the products they sell, supporting better marketing decisions. Besides market basket analysis, association rule discovery has been used in the areas of text mining and bioinformatics. The function [imsls_f_apriori](#) implements the Apriori algorithm. The function [imsls_f_aggr_apriori](#) performs the Apriori algorithm on subsets of transactions and aggregates the results.

For more details, see the *Description* section of [imsls_f_apriori](#).

Decision Trees

Decision trees are data mining methods for predicting a single response variable based on multiple predictor variables. If the response variable is categorical or discrete, the data mining problem is a classification problem; whereas if the response is continuous, the problem is a type of regression problem. Decision trees are generally applicable in both situations. The function [imsls_f_decision_tree](#) includes four of the most widely used algorithms for decision trees — the C4.5 method, ALACART, CHAID, and QUEST. The function [imsls_f_decision_tree_predict](#) applies a decision tree to a new set of data.

For a detailed overview, see [Decision Trees – An Overview](#) and the *Description* section of [imsls_f_decision_tree](#).

Random Decision Trees

The ensemble method known as random forest ([Breiman, 2001](#)) fits a collection of decision trees on bootstrap samples. In addition, the set of predictor variables is randomized before each branching or splitting decision within the decision tree algorithms. This extra randomization reduces correlation among the different trees in the ensemble. The method is available in the decision tree function [imsls_f_decision_tree](#).

Gradient Boosting

The function `ims1s_f_gradient_boosting` implements the stochastic gradient tree boosting algorithm of [Friedman \(1999\)](#). The algorithm combines the outputs of relatively weak classifiers or predictive models to achieve iteratively better and better accuracy in either regression problems (the response variable is continuous) or classification problems (the response variable has two or more discrete values). Gradient boosting is an ensemble method, but instead of using independent trees, gradient boosting forms a sequence of trees, iteratively and judiciously re-weighted to minimize prediction errors. In particular, the decision tree at iteration $m+1$ is estimated on pseudo-residuals generated using the decision tree at step m . Hence, successive trees are dependent on previous trees. The algorithm in gradient boosting iterates for a fixed number of times and stops, rather than iterating until a convergence criteria is met. The number of iterations is therefore a parameter in the model. Using a randomly selected subset of the training data in each iteration has been shown to substantially improve efficiency and robustness. Thus, the method is called stochastic gradient boosting. For further discussion, see [Hastie, et al. \(2009\)](#).

Genetic Algorithms

The original genetic algorithm is generally attributed to John Holland and his students from the University of Michigan. During the 1970s they investigated the use of concepts in genetics and biology in optimizing a function. Since that original work, many variations of the original algorithm have been developed by pioneers working in the interface between genetics, computer science and statistics to solve complex problems. These include traditional optimization and search problems in engineering, decision making, game solutions, engineering and pattern recognition.

The genetic algorithm operates on a population of individuals designed to represent the problem being solved. Each individual is rated according to a fitness function designed to measure how close that individual is to solving the problem. For optimization problems, the fitness function is usually constructed from the function being optimized. For other problems, the fitness function can be more complex defined only by the algorithm being investigated. A chess program, for example, might use a fitness function that scores the quality of a board position represented by each individual.

The solution represented by each individual in a population is encoded into the individual chromosome. The fitness function calculates a fitness value for each individual from the information in the individual chromosome. An investor might search for the best set of trading rules for optimizing the returns from the individual investment.

In this case, chromosomes would contain encoded representations of different variations of candidate trading rules. One binary bit might indicate whether a particular technical indicator was being used. Another part of the chromosome might be encoded to indicate how that indicator would be used to buy and sell investments. The fitness function would calculate a rate of return for each individual based upon actual historical data.

Several functions are available for building, cloning and copying chromosomes and individuals:

C Stat Library Function	Description
<code>imsls_f_ga_chromosome</code>	Creates the structure for a chromosome.
<code>imsls_f_ga_clone_chromosome</code>	Creates an exact duplicate of an existing chromosome.
<code>imsls_f_ga_copy_chromosome</code>	Copies the information contained in one chromosome into another.
<code>imsls_f_ga_individual</code>	Creates an individual using an existing chromosome.
<code>imsls_f_ga_clone_individual</code>	Creates an exact duplicate of an existing individual.
<code>imsls_f_ga_copy_individual</code>	Copies the information from one individual into another.

Solving any problem using a genetic algorithm always begins by creating a chromosome used for representing the problem. Four data types can be represented in a chromosome: binary, nominal, integer and real, or continuous attributes. Binary attributes are mapped directly into a chromosome as zeros and ones. A nominal attribute is represented as integers 0, 1, ..., $k-1$, where k is the maximum number of classes for that attribute. Integer and real attributes are mapped into a binary representation by dividing the range of the attribute into a finite number of subintervals. The range and number of intervals is supplied by the user when the chromosome is constructed. Either base-2 or Gray encoding can be used to encode integer and real attributes.

By default, encoding and decoding of chromosomes is automatic. That is each individual not only carries the chromosome but it also carries the original phenotype values encoded in the chromosome. Before the fitness of an individual is evaluated by calling the user's fitness function, the information in its chromosome is decoded into phenotype values. If this is too time consuming, automatic encoding and decoding can be disabled and done within the fitness functions. The functions `imsls_f_ga_encode` and `imsls_f_ga_decode` have been provided to encode and decode the chromosome of individuals, if needed. The routine `imsls_f_ga_mutate` has been provided to allow users to create their own genetic algorithm instead of using `imsls_f_genetic_algorithm`.

The memory allocated to a chromosome data structure can be released using the `imsls_free` function. However, the function `imsls_f_ga_free_individual` has been provided to release memory allocated to an individual.

The genetic algorithm implemented in `imsls_f_genetic_algorithm` evolves an initial population of individuals through several generations, searching for the optimum individuals. The initial population can be created using one of several functions. The easiest approach is to create a population of randomly selected individuals using `imsls_f_ga_random_population`. However, in some cases it might be better to initialize the population using an array of individuals selected based upon their fitness or diversity. The function `imsls_f_ga_population` can create a population data structure from an array of individuals.

In some cases it might be useful to restart a genetic algorithm search using a previous generation. The function `imsls_f_ga_clone_population` can be used to create an exact duplicate of a population. The function `imsls_f_ga_copy_population` replaces the individuals in one population with those from another. Two pop-

ulations can be merged using [imsls_f_ga_merge_population](#) and individuals can be added to an existing population using [imsls_f_ga_grow_population](#). Memory allocated to a population can be released using [imsls_f_ga_free_population](#).

The actual search or optimization using an initial population is conducted using [imsls_f_genetic_algorithm](#). This function returns the fittest individual found during the search. Also available are convergence statistics, including generation statistics, and the final population. This information can be used to evaluate the quality of the solution and if an additional search is warranted, the final population might be used as an initial population for that search.

For a detailed overview, see [Genetic Algorithms – An Overview](#).

Naive Bayes

Naive Bayes is a classification algorithm. First a classifier is trained using [imsls_f_naive_bayes_trainer](#). Once this is done [imsls_f_naive_bayes_classification](#) can be used to classify patterns with unknown classifications using the trained classifier represented in an *Imsls_f_naive_bayes* data structure.

In addition, [imsls_f_nb_classifier_write](#) can be used to store the data structure created by [imsls_f_naive_bayes_trainer](#) for future use. The function [imsls_f_nb_classifier_read](#) restores a Naive Bayes data structure from a file written using [imsls_f_nb_classifier_write](#).

Classification problems can be solved using other algorithms such as discriminant analysis and neural networks. In general these alternatives have smaller classification error rates, but they are too slow for large classification problems. During training [imsls_f_naive_bayes_trainer](#) uses the non-missing training data to estimate two-way correlations among the attributes. Higher order correlations are assumed to be zero. This can increase the classification error rate, but it significantly reduces the time needed to train the classifier.

In addition, the Naive Bayes algorithm is the only classification algorithm that can handle data with missing values. Other algorithms such as discriminant analysis do not allow missing values in the data. This is a significant limitation for applying other techniques to a larger database.

Memory allocated to the Naive Bayes data structure created by [imsls_f_naive_bayes_trainer](#) and [imsls_f_nb_classifier_read](#), can be released using [imsls_f_nb_classifier_free](#).

For a detailed overview, see [Naive Bayes – An Overview](#).

Neural Networks

Neural networks can be used for forecasting and classification. A neural network data structure is first created using `imsls_f_mlff_network_init` and `imsls_f_mlff_network`. Although forecasting and classification neural networks are initialized and represented using the same data structure, separate functions are provided for forecasting and classification in order to make them easier to use and to reduce network training times.

Once the network architecture is created, the network can be trained using `imsls_f_mlff_network_trainer` for forecasting problems and `imsls_f_mlff_classification_trainer` for classification problems. By default these algorithms initialize the network weights, but weight initialization can be controlled using `imsls_f_mlff_initialize_weights`.

Once a network is trained either `imsls_mlff_network_forecast` or `imsls_f_mlff_pattern_classification` is used to produce forecasts or to classify unknown patterns.

In many cases, network training will be completed on one computer and forecasting or classification might be done on another. The function `imsls_f_mlff_network_write` stores a trained network to a file which can be restored using `imsls_f_mlff_network_read`.

Memory allocated to a neural network data structure can be released using `imsls_f_mlff_network_free`.

For a detailed overview, see [Neural Networks – An Overview](#).

Data Filtering

The first step in this process is to filter data from its raw form into formats required by sophisticated analytical algorithms.

Data fall into two major categories: continuous and categorical. Many algorithms, such as neural network forecasting, perform better if continuous data are mapped into a common scale. The function `imsls_f_scale_filter` implements several techniques for automatically scaling continuous data, including several variations of z-score scaling. If the continuous data represent a time series, `imsls_f_time_series_filter` and `imsls_f_time_series_class_filter` can be used to create a matrix of lagged values required as input to forecasting neural networks.

Categorical data must also be mapped into a corresponding numerical representation before they can be used in solving forecasting and classification problems. There are two types of categorical data: *ordinal* and *nominal*. Ordinal data have a natural ordering among the categories, such as a school grade. Nominal data are categories without a natural ordering, such as eye color. The function `imsls_f_unsupervised_ordinal_filter` encodes and decodes ordinal data into the range [0, 1] using cumulative percentages. The function `imsls_f_unsupervised_nominal_filter` uses binary encoding to map nominal data into a matrix of zeros and ones.

Kohonen Self-organizing Map

A self-organizing map (SOM), also known as a Kohonen map or Kohonen SOM, is a technique for gathering high-dimensional data into clusters that are constrained to lie in low dimensional space, usually two dimensions. It is a widely used technique for the purpose of feature extraction and visualization for very high dimensional data. The Kohonen SOM is equivalent to an artificial neural network having inputs linked to every node in the network. The creation of a Kohonen map involves two steps: training and forecasting. Training builds the map using input examples, and forecasting classifies new input. The functions [imsls_f_kohonenSOM_trainer](#) and [imsls_f_kohonenSOM_forecast](#) achieve these two steps.

For more details, see the *Description* sections of [imsls_f_kohonenSOM_trainer](#) and [imsls_f_kohonenSOM_forecast](#).

Support Vector Machines

Support Vector Machines (SVM) is a class of learning algorithms that can be used for classification, regression, and distribution estimation. First, a classifier is trained using [imsls_f_support_vector_trainer](#) specifying either a classification, distribution, or regression type model. Then [imsls_f_support_vector_classification](#) can be used for classification, distribution estimation, or regression on patterns with unknown classifications using the trained classifier model represented in an *imsls_f_svm_classifier* data structure.

For a detailed overview, see [Support Vector Machines – An Overview](#)

apriori

Computes the frequent itemsets in a transaction set.

Synopsis

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

```
Imsls_f_apriori_itemsets *imsls_f_apriori (int n, int x[ ], int max_num_products, ..., 0)
```

The type *double* function is `imsls_d_apriori`.

Required Arguments

int n (Input)

Number of (transaction, item) pairs in **x**.

int x[] (Input)

Array of size **n** x 2, each row of which represents a transaction id and item id pair.

int max_num_products (Input)

Maximum number of unique items or products that may be present in the transactions.

`max_num_products` must be greater than or equal to the number of items in **x**.

Return Value

Pointer to an `Imsls_f_apriori_itemsets` data structure containing the frequent itemsets in the transaction set **x**. If no value can be computed, then `NULL` is returned. To release this space, use `imsls_free_apriori_itemsets`. Please see [Data Structures](#) for a description of this data structure.

Synopsis with Optional Arguments

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

```
Imsls_f_apriori_itemsets *imsls_f_apriori (int n, int x[ ], int max_num_products,
```

```
      IMSLS_MAX_SET_SIZE, int max_set_size,
```

```
      IMSLS_MIN_SUPPORT, double min_pct_support,
```

```

IMSLS_ASSOCIATION_RULES, float confidence, float lift,
    Imsls_f_association_rules **assoc_rules,

```

```

0)

```

Optional Arguments

```

IMSLS_MAX_SET_SIZE, int max_set_size (Input)

```

Maximum size of an itemset. Only frequent itemsets with `max_set_size` or fewer items are considered in the analysis.

Default: `max_set_size = 5`.

```

IMSLS_MIN_SUPPORT, double min_pct_support (Input)

```

Minimum percentage of transactions in which an item or itemset must be present to be considered frequent. `min_pct_support` must be in the interval [0,1].

Default: `min_pct_support = 0.1`.

```

IMSLS_ASSOCIATION_RULES, float confidence, float lift, Imsls_f_association_rules
    **assoc_rules (Input/Output)

```

Computes the strong association rules among itemsets.

```

    float confidence (Input)

```

The minimum confidence used to determine the strong association rules. `confidence` must be in the interval [0,1]. `lift` is the other criterion that determines whether an association is "strong." If either criterion, `confidence` or `lift`, is exceeded, the association rule is considered "strong."

```

    float lift (Input)

```

The minimum lift used to determine the strong association rules. `lift` must be non-negative. `confidence` is the other criterion that determines whether an association is "strong." If either criterion, `confidence` or `lift`, is exceeded, the association rule is considered "strong."

```

    Imsls_f_association_rules **assoc_rules (Output)

```

Address of a pointer to an `Imsls_f_association_rules` data structure containing the strong association rules among the itemsets. If no value can be computed, then `NULL` is returned. To release this space, use [imsls_f_free_association_rules](#).

Description

The function `imsls_f_apriori` performs the Apriori algorithm for association rule discovery. Association rules are statements of the form, "if X, then Y", given with some measure of confidence. The main application for association rule discovery is market basket analysis, where X and Y are products or groups of products, and the occurrences are individual transactions, or "market baskets." The results help sellers learn relationships between the different products they sell, supporting better marketing decisions. There are other applications for association rule discovery, such as the problem areas of text mining and bioinformatics. The Apriori algorithm (Agrawal and Srikant, 1994) is one of the most popular algorithms for association rule discovery in transactional datasets.

For distributed data or data larger than physical memory, see `imsls_f_aggr_apriori`.

In the first and most critical stage, the Apriori algorithm mines the transactions for frequent itemsets. An itemset is frequent if it appears in more than a minimum number of transactions. The number of transactions containing an itemset is known as its "support", and the minimum support (as a percentage of transactions) is a control parameter in the algorithm. The algorithm begins by finding the frequent single items. Then the algorithm generates all two-item sets from the frequent single items and determines which among them are frequent. From the collection of frequent pairs, Apriori forms candidate three-item subsets and determines which are frequent, and so on. The algorithm stops when either a maximum itemset size is reached, or when none of the candidate itemsets are frequent. In this way, the Apriori algorithm exploits the apriori-property: for an itemset to be frequent, all of its proper subsets must also be frequent. At each step the problem is reduced to only the frequent subsets.

In the second stage, the algorithm generates association rules. These are of the form, $X \Rightarrow Y$ (read, "if X, then Y"), where Y and X are disjoint frequent itemsets. The confidence measure associated with the rule is defined as the proportion of transactions containing X that also contain Y. Denote the support of X (the number of transactions containing X) as S_X , and S_Z is the support of $Z = X \cup Y$. The confidence of the rule $X \Rightarrow Y$ is the ratio, S_Z/S_X . Note that the confidence ratio is the conditional probability

$$P[X|Y] = \frac{P[XY]}{P[X]}$$

where $P[XY]$ denotes the probability of both X and Y. The probability of an itemset X is estimated by S_X/N , where N is the total number of transactions.

Another measure of the strength of the association is known as the lift, which is the ratio $(S_ZN) / (S_XS_Y)$. Lift values close to 1.0 suggest the sets are independent, and that they occur together by chance. Large lift values indicate a strong association. A minimum confidence threshold and a lift threshold can be specified.

Data Structures

The data structures output by `imsls_f_apriori` are described below. (For `imsls_d_apriori`, the structure names are `Imsls_d_apriori_itemsets`, `Imsls_d_association_rules`, and `Imsls_d_rule_components` where type `float` becomes `double`).

Structure definitions are provided for informational purposes and may be subject to change.

Table 35 - `Imsls_f_apriori_itemsets`

Field	Description
<code>int n_itemsets</code>	Length of array <code>itemsets</code> containing the <code>Imsls_apriori_items</code> structures.
<code>Imsls_apriori_items *itemsets</code>	Array of <code>Imsls_apriori_items</code> structures containing the set of frequent items and the support for that set.
<code>int n_trans</code>	Number of transactions.
<code>int max_num_products</code>	Maximum number of products.
<code>int max_set_size</code>	Maximum itemset size.
<code>double min_pct_support</code>	Minimum percentage of transactions.

Table 36 - `Imsls_apriori_items`

Field	Description
<code>int n_items</code>	Length of <code>items</code> .
<code>int *items</code>	Array containing the set of frequent items.
<code>int support</code>	The number of transactions in which the item appears.

Table 37 - `Imsls_f_association_rules`

Field	Description
<code>int n_rules</code>	Length of array <code>rules</code> containing the <code>Imsls_f_rule_components</code> structures.
<code>Imsls_f_rule_components *rules</code>	Array containing the association rules.

Table 38 - Imsls_f_rule_components

Field	Description
<i>int</i> n_x	Length of x.
<i>int</i> *x	Array containing the X components of the association rule.
<i>int</i> n_y	Length of y.
<i>int</i> *y	Array containing the Y components of the association rule.
<i>int</i> support[3]	Support for Z, X and Y components of the association rule.
<i>float</i> confidence	Confidence of the association rule.
<i>float</i> lift	Lift of the association rule.

Example

This example applies Apriori to find the frequent itemsets and strong association rules. The data are 50 transactions involving five different product IDs. The minimum support percentage is set to 0.30, giving a minimum required support of 15 transactions.

```
#include <imsls.h>

#define N 144

int main() {
    int max_num_products = 5, max_set_size = 10;
    float confidence = 0.8, lift = 2.0;
    double min_pct_support = 0.3;
    int x[N][2] = {
        {1, 3}, {1, 2}, {1, 1}, {2, 1}, {2, 2}, {2, 4}, {2, 5},
        {3, 3}, {4, 4}, {4, 3}, {4, 5}, {4, 1}, {5, 5}, {6, 1},
        {6, 2}, {6, 3}, {7, 5}, {7, 3}, {7, 2}, {8, 3}, {8, 4},
        {8, 1}, {8, 5}, {8, 2}, {9, 4}, {10, 5}, {10, 3}, {11, 2},
        {11, 3}, {12, 4}, {13, 4}, {14, 2}, {14, 3}, {14, 1}, {15, 3},
        {15, 5}, {15, 1}, {16, 2}, {17, 3}, {17, 5}, {17, 1}, {18, 5},
        {18, 1}, {18, 2}, {18, 3}, {19, 2}, {20, 4}, {21, 1}, {21, 4},
        {21, 2}, {21, 5}, {22, 5}, {22, 4}, {23, 2}, {23, 5}, {23, 3},
        {23, 1}, {23, 4}, {24, 3}, {24, 1}, {24, 5}, {25, 3}, {25, 5},
        {26, 1}, {26, 4}, {26, 2}, {26, 3}, {27, 2}, {27, 3}, {27, 1},
        {27, 5}, {28, 5}, {28, 3}, {28, 4}, {28, 1}, {28, 2}, {29, 4},
        {29, 5}, {29, 2}, {30, 2}, {30, 4}, {30, 3}, {31, 2}, {32, 5},
        {32, 1}, {32, 4}, {33, 4}, {33, 1}, {33, 5}, {33, 3}, {33, 2},
        {34, 3}, {35, 5}, {35, 3}, {36, 3}, {36, 5}, {36, 4}, {36, 1},
        {36, 2}, {37, 1}, {37, 3}, {37, 2}, {38, 4}, {38, 2}, {38, 3},
        {39, 3}, {39, 2}, {39, 1}, {40, 2}, {40, 1}, {41, 3}, {41, 5},
        {41, 1}, {41, 4}, {41, 2}, {42, 5}, {42, 1}, {42, 4}, {43, 3},
        {43, 2}, {43, 4}, {44, 4}, {44, 5}, {44, 2}, {44, 3}, {44, 1},
```

```

        {45, 4}, {45, 5}, {45, 3}, {45, 2}, {45, 1}, {46, 2}, {46, 4},
        {46, 5}, {46, 3}, {46, 1}, {47, 4}, {47, 5}, {48, 2}, {49, 1},
        {49, 4}, {49, 3}, {50, 3}, {50, 4}
    };
    imsls_f_apriori_itemsets *itemsets = NULL;
    imsls_f_association_rules *assoc_rules = NULL;

    /* Compute and print the strong association rules. */
    itemsets = imsls_f_apriori(N, &x[0][0], max_num_products,
        IMSLS_MAX_SET_SIZE, max_set_size,
        IMSLS_MIN_SUPPORT, min_pct_support,
        IMSLS_ASSOCIATION_RULES, confidence, lift, &assoc_rules,
        0);
    imsls_f_write_apriori_itemsets(itemsets);
    imsls_f_write_association_rules(assoc_rules);

    imsls_f_free_apriori_itemsets(itemsets);
    imsls_f_free_association_rules(assoc_rules);
}

```

Output

Frequent Itemsets (Out of 50 Transactions):

Size	Support	Itemset
1	27	{ 1 }
1	30	{ 2 }
1	33	{ 3 }
1	27	{ 4 }
1	27	{ 5 }
2	20	{ 1 2 }
2	22	{ 1 3 }
2	16	{ 1 4 }
2	19	{ 1 5 }
2	22	{ 2 3 }
2	16	{ 2 4 }
2	15	{ 2 5 }
2	16	{ 3 4 }
2	19	{ 3 5 }
2	17	{ 4 5 }
3	17	{ 1 2 3 }
3	15	{ 1 3 5 }

Association Rules (itemset X implies itemset Y):

```

X = {1} ==> Y = {3}
  supp(X)=27, supp(Y)=33, supp(X U Y)=22
  conf= 0.81, lift=1.23
X = {1 2} ==> Y = {3}
  supp(X)=20, supp(Y)=33, supp(X U Y)=17
  conf= 0.85, lift=1.29

```

Warning Errors

IMSLS_MIN_SUPPORT_NOT_MET

No items met minimum support of #.

Fatal Errors

IMSLS_NEED_IARG_GE

"name" = #. "name" must be greater than or equal to #.

aggr_apriori

Computes the frequent itemsets in a transaction set using aggregation.

Synopsis

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

```
void imsls_f_aggr_apriori (Imsls_keyword step, ..., 0)
```

The type *double* function is `imsls_d_aggr_apriori`.

Required Arguments

Imsls_keyword `step` (Input)

One optional argument must be supplied to indicate which calculation step is to be performed.

`step` is the name of the optional argument, defined as follows:

<code>step</code>	Description
<code>IMSLS_FREQUENT_ITEMSETS</code>	Compute the frequent itemsets.
<code>IMSLS_UNION</code>	Compute the union of two itemsets.
<code>IMSLS_COUNT</code>	Count the occurrence of each itemset in the transaction data set.
<code>IMSLS_SUM</code>	Add the counts of two itemsets.
<code>IMSLS_UPDATE_FREQ_ITEMSETS</code>	Update the set of frequent itemsets.
<code>IMSLS_ASSOCIATION_RULES</code>	Compute the strong association rules.

Synopsis with Optional Arguments

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

```
void imsls_f_aggr_apriori (
```

```
    IMSLS_FREQUENT_ITEMSETS, int n, int x[ ], int max_num_products,  
    int max_set_size,
```

```
    double min_pct_support, Imsls_f_apriori_itemsets **itemsets, or
```

```
    IMSLS_UNION, Imsls_f_apriori_itemsets *itemsets1,
```

```

    Imsls_f_apriori_itemsets *itemsets2, Imsls_f_apriori_itemsets **cand_itemsets, or
    IMSLS_COUNT, Imsls_f_apriori_itemsets *cand_itemsets, int n, int x[ ], int **freq, or
    IMSLS_SUM, int n_itemsets, int prev_freq1[ ], int prev_freq2[ ], int freq[ ], or
    IMSLS_UPDATE_FREQ_ITEMSETS, Imsls_f_apriori_itemsets *cand_itemsets,
    int n_itemsets, int freq[ ],
    Imsls_f_apriori_itemsets **itemsets, or
    IMSLS_ASSOCIATION_RULES, Imsls_f_apriori_itemsets *itemsets, float confidence,
    float lift, Imsls_f_association_rules **assoc_rules,
    0)

```

Optional Arguments

```

IMSLS_FREQUENT_ITEMSETS, int n, int x[ ], int max_num_products, int max_set_size,
double min_pct_support, Imsls_f_apriori_itemsets **itemsets (Input/Output)

```

Computes the frequent itemsets in a transaction set.

int n (Input)

Number of (transaction, item) pairs in **x**.

int x[] (Input)

Array of size **n** x 2, each row of which represents a transaction id and item id pair. The algorithm assumes that an individual transaction is complete within a single dataset. That is, there is no matching of transaction ids between different data sets.

int max_num_products (Input)

Maximum number of items or products that may be present in the aggregation of all transactions.

int max_set_size (Input)

Maximum size of an itemset. Only frequent itemsets with **max_set_size** or fewer items are considered in the analysis.

double min_pct_support (Input)

Minimum percentage of transactions in which an item or itemset must be present to be considered frequent. **min_pct_support** must be in the interval [0,1].

*Imsls_f_apriori_itemsets **itemsets* (Output)

Address of a pointer to an `Imsls_f_apriori_itemsets` data structure containing the frequent itemsets in the transaction set `x`. If no value can be computed, then `NULL` is returned. To release this space, use [imsls_free_apriori_itemsets](#).

or

`IMSLS_UNION`, *Imsls_f_apriori_itemsets *itemsets1*, *Imsls_f_apriori_itemsets *itemsets2*,
*Imsls_f_apriori_itemsets **cand_itemsets* (Input/Output)

Computes the union of two itemsets.

*Imsls_f_apriori_itemsets *itemsets1* (Input)

Pointer to an `Imsls_f_apriori_itemsets` data structure containing the frequent itemsets for the union.

*Imsls_f_apriori_itemsets *itemsets2* (Input)

Pointer to an `Imsls_f_apriori_itemsets` data structure containing the frequent itemsets for the union.

*Imsls_f_apriori_itemsets **cand_itemsets* (Output)

Address of a pointer to an `Imsls_f_apriori_itemsets` data structure containing the union of two itemsets. If no value can be computed, then `NULL` is returned. To release this space, use [imsls_free_apriori_itemsets](#).

or

`IMSLS_COUNT`, *Imsls_f_apriori_itemsets *cand_itemsets*, *int n*, *int x[]*, *int **freq* (Input/Output)

Counts the frequency of each itemset in a transaction data set.

Imsls_f_apriori_itemsets cand_itemsets (Input)

Candidate itemsets and the corresponding number of transactions.

int n (Input)

Number of transaction/item pairs in `x`.

int x[] (Input)

Array of size `n x 2`, each row of which represents a transaction id and item id pair. The algorithm assumes that an individual transaction is complete within a single dataset. That is, there are no matching of transaction ids between different data sets.

*int **freq* (Output)

Address of an internally allocated array of length `cand_itemsets->n_itemsets` containing the number of occurrences of each itemset in `x`. To release this space, use `imsls_free`.

or

`IMSLS_SUM, int n_itemsets, int prev_freq1[], int prev_freq2[], int **freq` (Input/Output)

Sum up the itemset frequencies in `prev_freq1` and `prev_freq2` and return in `freq`.

int n_itemsets (Input)

Length of `prev_freq1` and `prev_freq2` which corresponds to the number of itemsets.

int prev_freq1[] (Input)

Array of length `n_itemsets` containing the itemset frequencies counted over one or more blocks of transaction data.

int prev_freq2[] (Input)

Array of length `n_itemsets` containing the itemset frequencies counted over a second set of blocks of transaction data.

*int **freq* (Output)

Array of length `n_itemsets` containing the sum of the frequencies.

or

`IMSLS_UPDATE_FREQ_ITEMSETS, imsls_f_apriori_itemsets *cand_itemsets, int n_itemsets, int freq[], imsls_f_apriori_itemsets **itemsets` (Input/Output)

Updates the set of frequent items.

*imsls_f_apriori_itemsets *cand_itemsets* (Input)

Candidate itemsets and the corresponding number of transactions.

int n_itemsets (Input)

Length of `freq`.

int freq[] (Input)

Array of length `n_itemsets` containing the frequencies for each itemset in `cand_itemsets`.

Imsls_f_apriori_itemsets ****itemsets** (Output)

Address of a pointer to an `Imsls_f_apriori_itemsets` data structure containing the frequent itemsets. If no value can be computed, then `NULL` is returned. To release this space, use `imsls_free_apriori_itemsets`.

or

`IMSLS_ASSOCIATION_RULES`, *Imsls_f_apriori_itemsets* ***itemsets**, *float confidence*, *float lift*, *Imsls_f_association_rules* ****assoc_rules** (Input/Output)

Computes the strong association rules among itemsets.

Imsls_f_apriori_itemsets ***itemsets** (Input)

A pointer to an `Imsls_f_itemsets` data structure containing the itemsets.

float confidence (Input)

The minimum confidence used to determine the strong association rules. `confidence` must be in the interval [0,1]. `lift` is the other criterion that determines whether an association is "strong." If either criterion, `confidence` or `lift` is exceeded, the association rule is considered "strong."

float lift (Input)

The minimum lift used to determine the strong association rules. `lift` must be non-negative. `confidence` is the other criterion which determines whether an association is "strong." If either criterion, `confidence` or `lift` is exceeded, the association rule will be considered "strong."

Imsls_f_association_rules ****assoc_rules** (Output)

Address of a pointer to an `Imsls_f_association_rules` data structure containing the strong association rules among the itemsets. If no value can be computed, then `NULL` is returned. To release this space, use `imsls_free_association_rules`.

Description

The function `imsls_f_aggr_apriori` performs the Apriori algorithm for association rule discovery. Association rules are statements of the form, "if X, then Y", given with some measure of confidence. The main application for association rule discovery is market basket analysis, where X and Y are products or groups of products, and the occurrences are individual transactions, or "market baskets." The results help sellers learn relationships

between the different products they sell, supporting better marketing decisions. There are other applications for association rule discovery, such as the problem areas of text mining and bioinformatics. The Apriori algorithm (Agrawal and Srikant, 1994) is one of the most popular algorithms for association rule discovery in transactional datasets. For a full description of the Apriori algorithm, see [imsls_f_apriori](#).

The interface to the function `imsls_f_aggr_apriori` is designed to complete the analysis over a series of steps, with each step requiring a call to the function. With this design, Apriori can be applied to separate blocks of transactions. For each dataset or block, call `imsls_f_aggr_apriori` with `IMSLS_FREQUENT_ITEMSETS` to obtain the frequent itemsets from each block. The same parameter settings, such as minimum support percentage, must be used in each separate call. Then, call `imsls_f_aggr_apriori` with the keyword `IMSLS_UNION` sequentially to obtain the union of all the frequent itemsets. The resulting set serves as the "candidate" itemsets for the global set of transactions.

An itemset which is frequent in one transaction set may or may not be frequent in the larger collection. To find the itemsets that are frequent over the entire set of transactions, `imsls_f_aggr_apriori` performs another pass through the individual blocks, this time counting the occurrences of each of the itemsets in each of the transaction sets. This step can be done in parallel, using keyword `IMSLS_COUNT`. The next step is then to sum up the individual counts before filtering for the frequent itemsets. This is achieved with the keyword `IMSLS_SUM`, applied successively to pairs of previous counts. After this step, the frequencies of each itemset over all of the transactions are known and it remains to be seen if any meet the threshold to be considered "frequent". The final step in determining the frequent itemsets is `IMSLS_UPDATE_FREQ_ITEMSETS`. Once the frequent itemsets are known, the strong association rules can be found using the step, `IMSLS_ASSOCIATION_RULES`, although this is not a special step in the aggregation. The method is due to Savasere, Omiecinski, and Navathe (1995) and is also summarized and compared with other approaches in Rajaraman and Ullman (2011).

Since `imsls_f_aggr_apriori` can operate on subsets of data, it can be used when physical memory cannot hold the entire data set. Additionally, this design may be useful in parallel computing environments where nodes can be programmed to calculate intermediate results in parallel.

Data Structures

The data structures used by `imsls_f_aggr_apriori` are described below. (For `imsls_d_aggr_apriori`, the structure names are `Imsls_d_apriori_itemsets`, `Imsls_d_association_rules`, and `Imsls_d_rule_components` where type `float` becomes `double`).

Structure definitions are provided for informational purposes and may be subject to change.

Table 39 - `Imsls_f_apriori_itemsets`

Field	Description
<code>int n_itemsets</code>	Length of array <code>itemsets</code> containing the <code>Imsls_apriori_items</code> structures.
<code>Imsls_apriori_items *itemsets</code>	Array of <code>Imsls_apriori_items</code> structures containing the set of frequent items and the support for that set.
<code>int n_trans</code>	Number of transactions.
<code>int max_num_products</code>	Maximum number of products.
<code>int max_set_size</code>	Maximum itemset size.
<code>double min_pct_support</code>	Minimum percentage of transactions.

Table 40 - `Imsls_apriori_items`

Field	Description
<code>int n_items</code>	Length of <code>items</code> .
<code>int *items</code>	Array containing the set of frequent items.
<code>int support</code>	Number of transactions in which the item appears.

Table 41 - `Imsls_f_association_rules`

Field	Description
<code>int n_rules</code>	Length of array <code>rules</code> containing the <code>Imsls_f_rule_components</code> structures.
<code>Imsls_f_rule_components *rules</code>	Array containing the association rules.

Table 42 - `Imsls_f_rule_components`

Field	Description
<code>int n_x</code>	Length of <code>x</code> .
<code>int *x</code>	Array containing the X components of the association rule.
<code>int n_y</code>	Length of <code>y</code> .

Table 42 - Imsls_f_rule_components

Field	Description
<i>int</i> *y	Array containing the Y components of the association rule.
<i>int</i> support[3]	Support for Z, X and Y components of the association rule.
<i>float</i> confidence	Confidence of the association rule.
<i>float</i> lift	Lift of the association rule.

Example

This example shows how to apply Apriori to separate blocks of data and combine results. The data are two separate blocks of 50 transactions involving five different product IDs. The minimum support percentage is set to 0.30, providing a minimum required support of 30 transactions overall.

```
#include <imsls.h>

#define N1 144
#define N2 147

int main() {
    int i;
    int max_num_products = 5, max_set_size = 4;
    float confidence = 0.8, lift = 2.0;
    double min_pct_support = 0.30;
    int x1[N1][2] = {
        {1, 3}, {1, 2}, {1, 1}, {2, 1}, {2, 2}, {2, 4}, {2, 5},
        {3, 3}, {4, 4}, {4, 3}, {4, 5}, {4, 1}, {5, 5}, {6, 1},
        {6, 2}, {6, 3}, {7, 5}, {7, 3}, {7, 2}, {8, 3}, {8, 4},
        {8, 1}, {8, 5}, {8, 2}, {9, 4}, {10, 5}, {10, 3}, {11, 2},
        {11, 3}, {12, 4}, {13, 4}, {14, 2}, {14, 3}, {14, 1}, {15, 3},
        {15, 5}, {15, 1}, {16, 2}, {17, 3}, {17, 5}, {17, 1}, {18, 5},
        {18, 1}, {18, 2}, {18, 3}, {19, 2}, {20, 4}, {21, 1}, {21, 4},
        {21, 2}, {21, 5}, {22, 5}, {22, 4}, {23, 2}, {23, 5}, {23, 3},
        {23, 1}, {23, 4}, {24, 3}, {24, 1}, {24, 5}, {25, 3}, {25, 5},
        {26, 1}, {26, 4}, {26, 2}, {26, 3}, {27, 2}, {27, 3}, {27, 1},
        {27, 5}, {28, 5}, {28, 3}, {28, 4}, {28, 1}, {28, 2}, {29, 4},
        {29, 5}, {29, 2}, {30, 2}, {30, 4}, {30, 3}, {31, 2}, {32, 5},
        {32, 1}, {32, 4}, {33, 4}, {33, 1}, {33, 5}, {33, 3}, {33, 2},
        {34, 3}, {35, 5}, {35, 3}, {36, 3}, {36, 5}, {36, 4}, {36, 1},
        {36, 2}, {37, 1}, {37, 3}, {37, 2}, {38, 4}, {38, 2}, {38, 3},
        {39, 3}, {39, 2}, {39, 1}, {40, 2}, {40, 1}, {41, 3}, {41, 5},
        {41, 1}, {41, 4}, {41, 2}, {42, 5}, {42, 1}, {42, 4}, {43, 3},
        {43, 2}, {43, 4}, {44, 4}, {44, 5}, {44, 2}, {44, 3}, {44, 1},
        {45, 4}, {45, 5}, {45, 3}, {45, 2}, {45, 1}, {46, 2}, {46, 4},
        {46, 5}, {46, 3}, {46, 1}, {47, 4}, {47, 5}, {48, 2}, {49, 1},
        {49, 4}, {49, 3}, {50, 3}, {50, 4}
    };
    int x2[N2][2] = {
        {1, 2}, {1, 1}, {1, 4}, {1, 3}, {2, 2}, {2, 5}, {2, 3},
        {2, 1}, {2, 4}, {3, 5}, {3, 4}, {4, 2}, {5, 4}, {5, 2},
```

```

    {5, 3}, {5, 5}, {6, 3}, {6, 5}, {7, 2}, {7, 5}, {7, 4},
    {7, 1}, {7, 3}, {8, 2}, {9, 2}, {9, 4}, {10, 4}, {10, 2},
    {11, 4}, {11, 1}, {12, 3}, {12, 1}, {12, 5}, {12, 2}, {13, 2},
    {14, 3}, {14, 4}, {14, 2}, {15, 2}, {16, 5}, {16, 2}, {16, 4},
    {17, 1}, {18, 2}, {18, 3}, {18, 4}, {19, 3}, {19, 1}, {19, 2},
    {19, 4}, {20, 5}, {20, 1}, {21, 5}, {21, 4}, {21, 1}, {21, 3},
    {22, 4}, {22, 1}, {22, 5}, {23, 1}, {23, 2}, {24, 4}, {25, 4},
    {25, 3}, {26, 5}, {26, 2}, {26, 3}, {26, 4}, {26, 1}, {27, 2},
    {27, 1}, {27, 5}, {27, 3}, {28, 1}, {28, 2}, {28, 3}, {28, 4},
    {29, 5}, {29, 2}, {29, 1}, {30, 5}, {30, 3}, {30, 2}, {30, 4},
    {31, 4}, {31, 1}, {32, 1}, {32, 2}, {32, 3}, {32, 4}, {32, 5},
    {33, 3}, {33, 2}, {33, 4}, {33, 5}, {33, 1}, {34, 3}, {34, 4},
    {34, 5}, {34, 2}, {35, 2}, {35, 3}, {36, 3}, {36, 5}, {36, 4},
    {37, 1}, {37, 4}, {37, 2}, {37, 3}, {37, 5}, {38, 5}, {38, 3},
    {38, 1}, {38, 2}, {39, 2}, {39, 5}, {40, 4}, {40, 2}, {41, 4},
    {42, 4}, {43, 5}, {43, 4}, {44, 5}, {44, 4}, {44, 3}, {44, 2},
    {44, 1}, {45, 1}, {45, 2}, {45, 3}, {45, 5}, {45, 4}, {46, 3},
    {46, 4}, {47, 4}, {47, 5}, {47, 2}, {47, 3}, {48, 5}, {48, 3},
    {48, 2}, {48, 1}, {48, 4}, {49, 4}, {49, 5}, {50, 4}, {50, 1}
};
imsls_f_apriori_itemsets *itemsets1 = NULL, *itemsets2 = NULL,
    *cand_itemsets = NULL, *itemsets = NULL;
int *prev_freq1 = NULL, *prev_freq2 = NULL, *freq = NULL;
imsls_f_association_rules *assoc_rules = NULL;

/* Find frequent itemsets in x1 and x2. */
imsls_f_aggr_apriori(IMSLS_FREQUENT_ITEMSETS,
    N1, &x1[0][0], max_num_products,
    max_set_size, min_pct_support, &itemsets1,
    0);
imsls_f_aggr_apriori(IMSLS_FREQUENT_ITEMSETS,
    N2, &x2[0][0], max_num_products,
    max_set_size, min_pct_support, &itemsets2,
    0);

/* Take the union of itemsets1 and itemsets2. */
imsls_f_aggr_apriori(IMSLS_UNION,
    itemsets1, itemsets2, &cand_itemsets,
    0);

/* Count the frequencies of each candidate itemset in
each of the data sets */
imsls_f_aggr_apriori(IMSLS_COUNT, cand_itemsets,
    N1, &x1[0][0], &prev_freq1,
    0);
imsls_f_aggr_apriori(IMSLS_COUNT, cand_itemsets,
    N2, &x2[0][0], &prev_freq2,
    0);

/* Sum the frequencies. */
imsls_f_aggr_apriori(IMSLS_SUM, cand_itemsets->n_itemsets,
    prev_freq1, prev_freq2, &freq,
    0);

/* Determine which of the candidate itemsets are frequent. */
imsls_f_aggr_apriori(IMSLS_UPDATE_FREQ_ITEMSETS,
    cand_itemsets, cand_itemsets->n_itemsets, freq, &itemsets,
    0);

/* Generate the strong association rules. */

```

```

    imsls_f_aggr_apriori(IMSLS_ASSOCIATION_RULES,
        itemsets, confidence, lift, &assoc_rules,
        0);

    imsls_f_write_apriori_itemsets(itemsets);
    imsls_f_write_association_rules(assoc_rules);

    imsls_f_free_apriori_itemsets(itemsets1);
    imsls_f_free_apriori_itemsets(itemsets2);
    imsls_f_free_apriori_itemsets(cand_itemsets);
    imsls_f_free_apriori_itemsets(itemsets);
    imsls_f_free_association_rules(assoc_rules);
    imsls_free(prev_freq1);
    imsls_free(prev_freq2);
    imsls_free(freq);
}

```

Output

Frequent Itemsets (Out of 100 Transactions):

Size	Support	Itemset
1	51	{ 1 }
1	63	{ 2 }
1	60	{ 3 }
1	63	{ 4 }
1	54	{ 5 }
2	37	{ 1 2 }
2	38	{ 1 3 }
2	33	{ 1 4 }
2	35	{ 1 5 }
2	44	{ 2 3 }
2	38	{ 2 4 }
2	34	{ 2 5 }
2	38	{ 3 4 }
2	38	{ 3 5 }
2	37	{ 4 5 }
3	32	{ 1 2 3 }
3	31	{ 2 3 4 }

Association Rules (itemset X implies itemset Y):

```

X = {1 2} ==> Y = {3}
  supp(X)=37, supp(Y)=60, supp(X U Y)=32
  conf= 0.86, lift=1.44
X = {1 3} ==> Y = {2}
  supp(X)=38, supp(Y)=63, supp(X U Y)=32
  conf= 0.84, lift=1.34
X = {2 4} ==> Y = {3}
  supp(X)=38, supp(Y)=60, supp(X U Y)=31
  conf= 0.82, lift=1.36
X = {3 4} ==> Y = {2}
  supp(X)=38, supp(Y)=63, supp(X U Y)=31
  conf= 0.82, lift=1.29

```

Warning Errors

IMSLS_MIN_SUPPORT_NOT_MET No items met minimum support of #.

Fatal Errors

IMSLS_NEED_IARG_GE "name" = #. "name" must be greater than or equal to #.

IMSLS_NEED_IARG_GT "name" = #. "name" must be greater than #.

IMSLS_INEQUALITY_VIOLATION_12 "name1" = # must equal "name2" = #.

write_apriori_itemsets

Prints frequent itemsets.

Synopsis

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

```
void imsls_f_write_apriori_itemsets (Imsls_f_apriori_itemsets *itemsets)
```

The type *double* function is `imsls_d_write_apriori_itemsets`.

Required Arguments

Imsls_f_apriori_itemsets *itemsets (Input)

A pointer to an `Imsls_f_apriori_itemsets` data structure containing the itemsets.

Description

The function `imsls_f_write_apriori_itemsets` prints frequent itemsets stored in an `Imsls_f_apriori_itemsets` structure.

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`).

Example

See `imsls_f_apriori` or `imsls_f_aggr_apriori`.

write_association_rules

Prints association rules.

Synopsis

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

```
void imsls_f_write_association_rules (Imsls_f_association_rules *assoc_rules)
```

The type *double* function is `imsls_d_write_association_rules`.

Required Arguments

Imsls_f_association_rules *assoc_rules (Input)

A pointer to an `Imsls_f_association_rules` data structure containing the association rules.

Description

The function `imsls_f_write_association_rules` prints the strong association rules stored in an `Imsls_f_association_rules` structure.

Output is written to the file specified by the function `imsls_output_file`. The default output file is standard output (corresponding to the file pointer `stdout`).

Example

See `imsls_f_apriori` or `imsls_f_aggr_apriori`.

free_apriori_itemsets

Frees the memory allocated within a frequent itemsets structure.

Synopsis

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

```
void imsls_f_free_apriori_itemsets (Imsls_f_apriori_itemsets *itemsets)
```

The type *double* function is `imsls_d_free_apriori_itemsets`.

Required Arguments

*Imsls_f_apriori_itemsets *itemsets* (Input)

A pointer to an `Imsls_f_apriori_itemsets` data structure containing the itemsets. See [imsls_f_apriori](#).

Description

Frees the memory allocated within a frequent itemsets structure.

Example

See [imsls_f_apriori](#) or [imsls_f_aggr_apriori](#).

free_association_rules

Frees the memory allocated within an association rules structure.

Synopsis

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

```
void imsls_f_free_association_rules (Imsls_f_association_rules *assoc_rules)
```

The type *double* function is `imsls_d_free_association_rules`.

Required Arguments

Imsls_f_association_rules ***assoc_rules** (Input)

A pointer to an `Imsls_f_association_rules` data structure containing the association rules.

See [imsls_f_association_rules](#).

Description

Frees the memory allocated within an association rules structure.

Example

See the examples in [imsls_f_apriori](#) or [imsls_f_aggr_apriori](#).

Decision Trees – An Overview

Decision trees are data mining methods for predicting a single response variable based on multiple predictor variables. If the response variable is categorical or discrete, the data mining problem is a classification problem, whereas if the response is continuous, the problem is a type of regression problem. Decision trees are generally applicable in both situations.

A trivial but illustrative example regards the decision to play golf or not, depending on the weather. The training data, from [Quinlan \(1993\)](#), is given in [Table 43](#) and a decision tree fit to the data is shown in [Figure 22](#). Other examples include predicting the chance of survival for heart attack patients based on age, blood pressure and other vital signs; scoring loan applications based on credit history, income, and education; classifying an email as spam based on its characteristics, and so on.

Tree-growing algorithms have similar steps: starting with all observations in a root node, a predictor variable is selected to split the dataset into two or more child nodes or branches. The form of the split depends on the type of predictor and on specifics of the algorithm. If the predictor is categorical, taking discrete values {A, B, C, D} for example, the split may consist of two or more proper subsets, such as {A}, {B, C}, and {D}. If the predictor is continuous, a split will consist of two or more intervals, such as $X \leq 2$, $X > 2$. The splitting procedure is then repeated for each child node and continued in such manner until one of several possible stopping conditions is met. The result of the decision tree algorithm is a tree structure with a root and a certain number of branches (or nodes). Each branch defines a subset or partition of the data and, conditional on that subset of data, a predicted value for the response variable. A traversal of a branch of the tree thus leads to a prediction, or *decision* about the response variable. To predict a new *out of sample* observation, we find the terminal node to which the observation belongs by traversing the tree and finding the data subset (branch) that contains the observation.

For example, the decision tree in [Figure 22](#) can be expressed as a set of rules: If the weather is sunny, don't play golf. If the weather is overcast, play golf. If the weather is rainy and there is no wind, play golf. On the other hand, if it is rainy and windy, don't play golf.

Decision trees are intuitive and can be very effective predictive tools. As with any predictive model, a decision tree should be tested on hold-out datasets or refined using K-fold cross-validation to prevent over-fitting.

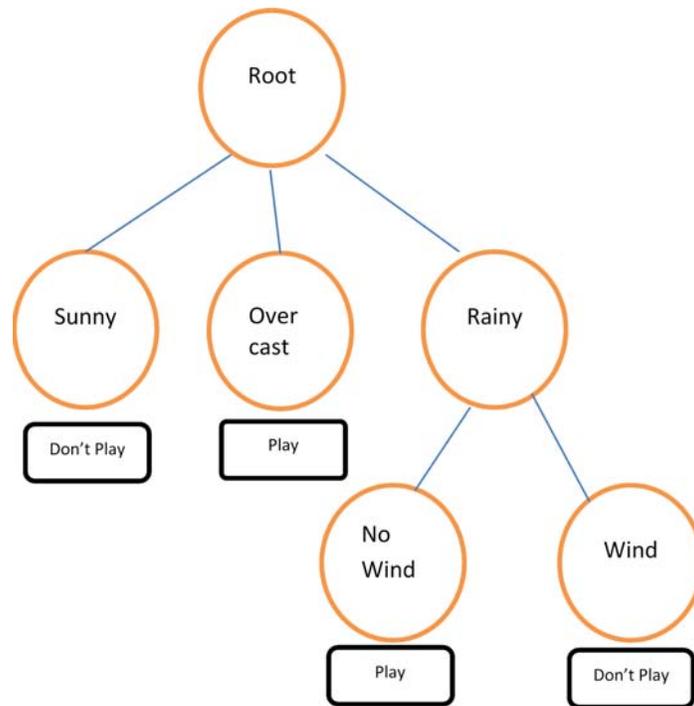


Figure 22, Play Golf? This tree has a size of 6, 4 terminal nodes, and a height or depth of 2.

Table 43 – Golf training data

Outlook	Temperature	Humidity	Wind	Play
sunny	85	85	FALSE	don't play
sunny	80	90	TRUE	don't play
overcast	83	78	FALSE	play
rainy	70	96	FALSE	play
rainy	68	80	FALSE	play
rainy	65	70	TRUE	don't play
overcast	64	65	TRUE	play
sunny	72	95	FALSE	don't play
sunny	69	70	FALSE	play
rainy	75	80	FALSE	play
sunny	75	70	TRUE	play
overcast	72	90	TRUE	play

Table 43 – Golf training data

Outlook	Temperature	Humidity	Wind	Play
overcast	81	75	FALSE	play
rainy	71	80	TRUE	don't play
overcast	81	75	FALSE	play
rainy	71	80	TRUE	don't play

decision_tree

Generates a decision tree for a single response variable and two or more predictor variables.

Synopsis

```
#include <imsls.h>

imsls_f_decision_tree* imsls_f_decision_tree (int n, int n_cols, float xy[ ],
      int response_col_idx, int var_type[ ], ..., 0)
```

The type *double* function is `imsls_d_decision_tree`.

Required Arguments

int `n` (Input)

The number of rows in `xy`.

int `n_cols` (Input)

The number of columns in `xy`.

float `xy[]` (Input)

Array of size `n × n_cols` containing the data.

int `response_col_idx` (Input)

Column index of the response variable.

int `var_type[]` (Input)

Array of length `n_cols` indicating the type of each variable.

	Type
0	Categorical
1	Ordered Discrete (Low, Med., High)
2	Quantitative or Continuous
3	Ignore this variable

NOTE: When the variable type (*var_type*) is specified as Categorical (0), the numbering of the categories must begin at 0. For example, if there are three categories, they must be represented as 0, 1 and 2 in the *xy* array.

The number of classes for a categorical response variable is determined by the largest value discovered in the data. To set this value in another way, see optional arguments [IMSLS_PRIORS](#) or [IMSLS_COST_MATRIX](#). Also note that a warning message is displayed if a class level in 0, 1, ..., *n_classes*-1 has a 0 count in the data.

Return Value

A pointer to a structure of type *ImSls_f_decision_tree*. If an error occurs, NULL is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

ImSls_f_decision_tree *imsls_f_decision_tree (int n, int n_cols, float xy[ ],
    int response_col_idx, int var_type[ ],
    IMSLS_METHOD, int method,
    IMSLS_CRITERIA, int criteria,
    IMSLS_RATIO,
    IMSLS_WEIGHTS, float weights[ ],
    IMSLS_COST_MATRIX, int n_classes, float cost_matrix[ ],
    IMSLS_CONTROL, int params[ ],
    IMSLS_COMPLEXITY, float complexity,
    IMSLS_N_SURROGATES, int n_surrogates,
    IMSLS_ALPHAS, float alphas[ ],
    IMSLS_PRIORS, int n_classes, float priors[ ],
    IMSLS_N_FOLDS, int n_folds,
    IMSLS_N_SAMPLE, int n_samples,
    IMSLS_RANDOM_FEATURES,
    IMSLS_N_RANDOM_FEATURES, int n_features,
    IMSLS_TOLERANCE, float tol,
    IMSLS_RANDOM_SEED, int seed,
    IMSLS_PRINT, int print_level,
    IMSLS_TEST_DATA, int n_test, float xy_test[ ],
    IMSLS_TEST_DATA_WEIGHTS, float weights_test[ ],
    IMSLS_ERROR_SS, float *pred_err_ss,
    IMSLS_PREDICTED, float **predictions,
    IMSLS_PREDICTED_USER, float predictions[ ],
```

```

IMSLS_CLASS_ERROR, float **class_errors,
IMSLS_CLASS_ERROR_USER, float class_errors[ ],
IMSLS_MEAN_ERROR, float *mean_error,
IMSLS_OUT_OF_BAG_PREDICTED, float **oob_predicted,
IMSLS_OUT_OF_BAG_PREDICTED_USER, float oob_predicted[ ],
IMSLS_OUT_OF_BAG_MEAN_ERROR, float *out_of_bag_mean_error,
IMSLS_OUT_OF_BAG_CLASS_ERROR, float **out_of_bag_class_errors,
IMSLS_OUT_OF_BAG_CLASS_ERROR_USER, float out_of_bag_class_errors[ ],
IMSLS_OUT_OF_BAG_VAR_IMPORTANCE, float **out_of_bag_var_importance,
IMSLS_OUT_OF_BAG_VAR_IMPORTANCE_USER, float out_of_bag_var_importance[ ],
IMSLS_RETURN_TREES, Imcls_f_decision_tree ***bagged_trees,
0)

```

Optional Arguments

`IMSLS_METHOD`, *int* `method` (Input)

Specifies the tree generation method. The key for the variable type index is provided above.

<code>method</code>	Method	Response var_type	Predictor var_type
0	C4.5	0	0, 1, 2
1	ALACART (Breiman, et. al.)	0, 1, 2	0, 1, 2
2	CHAID	0, 1, 2	0
3	QUEST	0	0, 1, 2

Default: `method = 0`.

`IMSLS_CRITERIA`, *int* `criteria` (Input)

Specifies which criteria the ALACART method and the C4.5 method should use in the gain calculations to determine the best split at each node.

<code>criteria</code>	Measure
0	Shannon Entropy
1	Gini Index
2	Deviance

Default: `criteria = 0`.

Shannon Entropy – A measure of randomness or uncertainty.

For a categorical variable having C distinct values over the data set S , the Shannon Entropy is defined as

$$\sum_{i=1}^C p_i \log(p_i)$$

where

$$p_i = Pr(Y = i)$$

and where

$$p_i \log(p_i) := 0 \\ \text{if } p_i = 0$$

Gini Index – A measure of statistical dispersion.

For a categorical variable having C distinct values over the data set S , the Gini Index is defined as

$$I(S) = \sum_{\substack{i,j=1 \\ i \neq j}}^C p(i|S) = 1 - \sum_{i=1}^C p^2(i|S)$$

where $p(i|S)$ denotes the probability that the variable is equal to the state i on the data set, S .

Deviance – A measure of the quality of fit.

For a categorical variable having C distinct values over a data set S , the Deviance measure is

$$\sum_{i=1}^C n_i \log(p_i)$$

where

$$p_i = Pr(Y = i)$$

and where

$$n_i$$

is the number of cases with $Y = i$ on the node.

IMSLR_RATIO, (Input)

If present, the ALACART method and C4.5 method each uses a gain ratio instead of just the gain to determine the best split.

Default: Uses gain.

IMSLS_WEIGHTS, *float weights* [] (Input)

An array of length **n** containing case weights.

Default: `weights[i] = 1.0`.

IMSLS_COST_MATRIX, *int n_classes, float cost_matrix* [] (Input)

An array of length **n_classes** x **n_classes** containing the cost matrix for a categorical response variable, where **n_classes** is the number of classes the response variable may assume. The cost matrix has elements $C(i, j)$ = cost of misclassifying a response in class *j* as in class *i*. The diagonal elements of the cost matrix must be 0.

Default: `cost_matrix[i] = 1.0`, for *i* on the off-diagonal, `cost_matrix[i] = 0.0`, for *i* on the diagonal.

IMSLS_CONTROL, *int params* [] (Input)

Array of length 5 containing parameters to control the maximum size of the tree and other stopping rules.

params[i]	name	Action
0	<code>min_n_node</code>	Do not split a node if one of its child nodes will have fewer than <code>min_n_node</code> observations.
1	<code>min_split</code>	Do not split a node if the node has fewer than <code>min_split</code> observations
2	<code>max_x_cats</code>	Allow for up to <code>max_x_cats</code> for categorical predictor variables.
3	<code>max_size</code>	Stop growing the tree once it has reached <code>max_size</code> number of nodes.
4	<code>max_depth</code>	Stop growing the tree once it has reached <code>max_depth</code> number of levels.

Default: `params[] = {7, 21, 10, 100, 10}`

IMSLS_COMPLEXITY, *float complexity* (Input)

The minimum complexity parameter to use in cross-validation. Complexity must be ≥ 0 .

Default: `complexity = 0.0`.

IMSLS_N_SURROGATES, *int n_surrogates* (Input)

Indicates the number of surrogate splits. Only used if `method = 1`.

Default: `n_surrogates = 0`.

IMSLS_ALPHAS, *float alphas* [] (Input)

An array of length 3 containing the significance levels. `alphas[0]` = significance level for split variable selection (CHAID and QUEST); `alphas[1]` = significance level for merging categories of a variable (CHAID), and `alphas[2]` = significance level for splitting previously merged categories

(CHAID). Valid values are in the range $0 < \text{alphas}[i] < 1.0$, and $\text{alphas}[2] \leq \text{alphas}[1]$. Setting $\text{alphas}[2] = -1.0$ disables splitting of merged categories.
Default: $\text{alphas}[] = \{0.05, 0.05, -1.0\}$

IMSLS_PRIORS, *int n_classes*, *float priors[]* (Input)

An array of length *n_classes*, where *n_classes* is the number of classes the response variable may assume, containing prior probabilities for class membership. The argument is ignored for continuous response variables ($\text{var_type}[\text{response_col_idx}] = 2$). By default, the prior probabilities are estimated from the data.

IMSLS_N_FOLDS, *int n_folds* (Input)

The number of folds to use in cross validation tree selection. *n_folds* must be between 1 and *n*, inclusive. If *n_folds* = 1 the full data set is used once to generate the decision tree. In other words, no cross-validation is performed. If $1 < n/n_folds \leq 3$, then leave-one-out cross validation is performed.

Default: *n_folds* = 10.

IMSLS_N_SAMPLE, *int n_samples* (Input)

The number of bootstrap samples to use in bootstrap aggregation (*bagging*) when predicted values are requested. To obtain predictions produced by bagging, set *n_samples* > 0 and use one of **IMSLS_PREDICTED** or **IMSLS_PREDICTED_USER**.

Default: *n_samples* = 0 unless random features or out-of-bag calculations are requested; then, *n_samples* = 50.

IMSLS_RANDOM_FEATURES, (Input)

If present, the decision tree splitting rules at each node are decided from a random subset of predictors. Use this argument to generate a random forest for predictions. Use the argument **IMSLS_N_SAMPLE** to control the number of trees and **IMSLS_N_RANDOM_FEATURES** to control the number of random features.

Default: No random feature selection. The algorithms use all predictors in every selection.

IMSLS_N_RANDOM_FEATURES, *int n_random_features* (Input)

The number of predictors in each random subset from which to select during random feature selection, when it is activated.

Default: For categorical variables, *n_random_features* = the nearest integer $\leq p/3$ or 1; for regression variables, *n_random_features* = the nearest integer $\leq \sqrt{p/3}$ or 1, whichever is larger, where *p* = the number of variables.

IMSLS_TOLERANCE, *float tol* (Input)

Error tolerance to use in the algorithm.

Default: $\text{tol} = 100.0 * \text{imsls_f_machine}(4)$.

IMSLS_RANDOM_SEED, *int seed* (Input)

The seed of the random number generator used in sampling or cross-validation. By changing the value of **seed** on different calls to `imsls_f_decision_tree`, with the same data set, calls may produce slightly different results. Setting **seed** to zero forces random number seed determination by the system clock.

Default: **seed** = 0

IMSLS_PRINT, *int print_level* (Input)

print_level	Action
0	No printing
1	Prints final results only.
2	Prints intermediate and final results.

Default: **print_level** = 0.

IMSLS_TEST_DATA, *int n_test*, *float xy_test[]* (Input)

xy_test is an array of size **n_test** x **ncols** containing hold-out or test data for which predictions are requested. When this optional argument is present, the number of observations in **xy_test**, **n_test**, must be greater than 0. The response variable may have missing values in **xy_test**, but it must be in the same column and the predictors must be in the same columns as they are in **xy**. If the test data is not provided but predictions are requested, then **xy** is used and the predictions are the fitted values.

Default: **xy_test** = **xy**

IMSLS_TEST_DATA_WEIGHTS, *float weights_test[]* (Input)

An array of size **n_test** containing frequencies or weights for each observation in **xy_test**. This argument is ignored if **IMSLS_TEST_DATA** is not present.

Default: **weights_test** = **weights**.

IMSLS_ERROR_SS, *float *pred_err_ss* (Output)

The fitted data error mean sum of squares in the absence of test data (**xy_test**). When test data is provided, the prediction error mean sum of squares is returned.

IMSLS_PREDICTED, *float **predicted* (Output)

Address of a pointer to an array of length **n** containing the fitted or predicted value of the response variable for each case in the input data or test data, if provided.

IMSLS_PREDICTED_USER, *float predicted[]* (Output)

Storage for the array of the fitted or predicted value for each case is provided by the user.

IMSLS_CLASS_ERROR, *float **class_errors* (Output)

Address of a pointer to an array of length $2 \times (\text{n_classes} + 1)$ containing classification errors for each level of the categorical response variable, along with the total occurrence in the input data or test data, and overall totals.

To illustrate, if `class_errors[2*j] = 20`, and `class_errors[2*j+1] = 105`, then we know there were 20 misclassifications of class level `j` out of a total of 105 in the data.

IMSLS_CLASS_ERROR_USER, *float class_errors[]* (Output)

Storage for the array of the class errors is provided by the user.

IMSLS_MEAN_ERROR, *float *mean_error* (Output)

The fitted data error mean sum of squares (continuous response) or misclassification percentage (categorical response) in the absence of test data (`xy_test`). When test data is provided, the prediction mean error is returned.

IMSLS_OUT_OF_BAG_PREDICTED, *float **oob_predicted* (Output)

Address of a pointer to an array of length `n` containing the out-of-bag predicted value of the response variable for every case in the input data when bagging is performed.

IMSLS_OUT_OF_BAG_PREDICTED_USER, *float oob_predicted[]* (Output)

Storage for the array of the out-of-bag predicted values is provided by the user.

IMSLS_OUT_OF_BAG_MEAN_ERROR, *float *out_of_bag_mean_error* (Output)

The out-of-bag predictive error mean sum of squares (continuous response) or misclassification percentage (categorical response) on the input data, when bagging is performed.

IMSLS_OUT_OF_BAG_CLASS_ERROR, *float ***out_of_bag_class_errors* (Output)

Address of a pointer to an array of length $2 \times (\text{n_classes} + 1)$ containing out-of-bag classification errors for each level of the categorical response variable, along with the total occurrence in the input data, and overall totals. This output is populated only when bagging (`n_samples > 1`) is requested.

IMSLS_OUT_OF_BAG_CLASS_ERROR_USER, *float out_of_bag_class_errors[]* (Output)

Storage for the array of the out-of-bag class errors is provided by the user.

IMSLS_OUT_OF_BAG_VAR_IMPORTANCE, *float **out_of_bag_var_importance* (Output)

Address of a pointer to an array of length `n_preds` containing the out-of-bag variable importance measure for each predictor. This output is populated only when bagging (`n_samples > 1`) is requested.

IMSLS_OUT_OF_BAG_VAR_IMPORTANCE_USER, *float out_of_bag_var_importance[]* (Output)

Storage for the array of the out-of-bag variable importance is provided by the user.

`IMSL_S_RETURN_TREES`, `lmsls_f_decision_tree ***bagged_trees` (Output)

Address of a pointer to an array of length `n_samples` containing the collection of trees generated during the algorithm. To release this space, use `lmsls_bagged_trees_free`.

Description

This implementation includes four of the most widely used algorithms for decision trees. Below is a brief summary of each approach.

C4.5

The method C4.5 (Quinlan, 1995) is a tree partitioning algorithm for a categorical response variable and categorical or quantitative predictor variables. The procedure follows the general steps outlined above, using as splitting criterion the information *gain* or *gain ratio*. Specifically, the *entropy* or *uncertainty* in the response variable with C categories over the full training sample S is defined as

$$E(S) = - \sum_{i=1}^C p_i \log(p_i)$$

Where $p_i = \Pr[Y = i | S]$ is the probability that the response takes on category i on the dataset S . This measure is widely known as the Shannon Entropy. Splitting the dataset further may either increase or decrease the entropy in the response variable. For example, the entropy of Y over a partitioning of S by X , a variable with K categories, is given by

$$E(S, X) = - \sum_{k=1}^K \sum_{i=1}^{C_k} p(S_k) E(S_k)$$

If any split defined by the values of a categorical predictor decreases the entropy in Y , then it is said to yield *information gain*:

$$g(S, X) = E(S) - E(S, X)$$

The best splitting variable according to the information gain criterion is the variable yielding the largest information gain, calculated in this manner. A modified criterion is the *gain ratio*:

$$gR(S, X) = \frac{E(S) - E(S, X)}{E_X(S)}$$

where

$$E_x(S) = - \sum_{k=1}^K v_k \log(v_k)$$

with

$$v_k = \Pr[X = k | S]$$

Note that $E_x(S)$ is just the entropy of the variable X over S . The gain ratio is thought to be less biased toward predictors with many categories. C4.5 treats the continuous variable similarly, except that only binary splits of the form $X \leq d$ and $X > d$ are considered, where d is a value in the range of X on S . The best split is determined by the split variable and split point that gives the largest criterion value. It is possible that no variable meets the threshold for further splitting at the current node, in which case growing stops and the node becomes a *terminal* node. Otherwise, the node is split creating two or more child nodes. Then, using the dataset partition defined by the splitting variable and split value, the very same procedure is repeated for each child node. Thus a collection of nodes and child-nodes are generated, or, in other words, the tree is *grown*. The growth stops after one or more different conditions are met.

ALACART

ALACART implements the method of [Breiman, Friedman, Olshen and Stone](#) (1984), the original authors and developers of CART™. CART™ is the trademarked name for Classification and Regression Trees. In ALACART, only binary splits are considered for categorical variables. That is, if X has values $\{A, B, C, D\}$, splits into only two subsets are considered, e.g., $\{A\}$ and $\{B, C, D\}$, or $\{A, B\}$ and $\{C, D\}$, are allowed, but a three-way split defined by $\{A\}$, $\{B\}$ and $\{C, D\}$ is not.

For classification problems, ALACART uses a similar criterion to information gain called *impurity*. The method searches for a split that reduces the node impurity the most. For a given set of data S at a node, the node impurity for a C -class categorical response is a function of the class probabilities

$$I(S) = \phi(p(1 | S), p(2 | S), \dots, p(C | S))$$

The measure function $\phi(\cdot)$ should be 0 for “pure” nodes, where all Y are in the same class, and maximum when Y is uniformly distributed across the classes.

As only binary splits of a subset S are considered (S_1, S_2 such that $S = S_1 \cup S_2$ and $S = S_1 \cap S_2 = \emptyset$), the reduction in impurity when splitting S into S_1, S_2 is

$$\Delta I = I(S) - q_1 I(S_1) - q_2 I(S_2)$$

where $q_j = \Pr[S_j]$, $j = 1, 2$ — the node probability.

The gain criteria and the reduction in impurity ΔI are similar concepts and equivalent when I is entropy and when only binary splits are considered. Another popular measure for the impurity at a node is the *Gini* index, given by

$$I(S) = \sum_{\substack{i,j=1 \\ i \neq j}}^C p(i|S)p(j|S) = 1 - \sum_{i=1}^C p^2(i|S)$$

If Y is an ordered response or continuous, the problem is a regression problem. ALACART generates the tree using the same steps, except that node-level measures or loss-functions are the mean squared error (MSE) or mean absolute error (MAD) rather than node impurity measures.

CHAID

The third method is appropriate only for categorical or discrete ordered predictor variables. Due to [Kass \(1980\)](#), CHAID is an acronym for chi-square automatic interaction detection. At each node, as above, CHAID looks for the best splitting variable. The approach is as follows: given a predictor variable X , perform a 2-way chi-squared test of association between each possible pair of categories of X with the categories of Y . The least significant result is noted and, if a threshold is met, the two categories of X are merged. Treating this merged category as a single category, repeat the series of tests and determine if there is further merging possible. If a merged category consists of three or more of the original categories of X , CHAID calls for a step to test whether the merged categories should be split. This is done by forming all binary partitions of the merged category and testing each one against Y in a 2-way test of association. If the most significant result meets a threshold, then the merged category is split accordingly. As long as the threshold in this step is smaller than the threshold in the merge step, the splitting step and the merge step will not cycle back and forth. Once each predictor is processed in this manner, the predictor with the most significant qualifying 2-way test with Y is selected as the splitting variable, and its last state of merged categories define the split at the given node. If none of the tests qualify (by having an adjusted p-value smaller than a threshold), then the node is not split. This growing procedure continues until one or more stopping conditions are met.

QUEST

The fourth method, the QUEST algorithm ([Loh and Shih, 1997](#)), is appropriate for a categorical response variable and predictors of either categorical or quantitative type. For each categorical predictor, QUEST performs a multi-way chi-square test of association between the predictor and Y . For every continuous predictor, QUEST performs an ANOVA test to see if the means of the predictor vary among the groups of Y . Among these tests, the variable with the most significant result is selected as a potential splitting variable, say, X_j . If the p-value (adjusted for multiple tests) is less than the specified splitting threshold, then X_j is the splitting variable for the current node. If not, QUEST performs for each continuous variable X a Levene's test of homogeneity to see if the variance of X varies within the different groups of Y . Among these tests, we again find the predictor with the most significant result, say X_i . If its p-value (adjusted for multiple tests) is less than the splitting threshold, X_i is the splitting variable. Otherwise, the node is not split.

Assuming a splitting variable is found, the next step is to determine how the variable should be split. If the selected variable X_j is continuous, a split point d is determined by quadratic discriminant analysis (QDA) of X_j into two populations determined by a binary partition of the response Y . The goal of this step is to group the classes of Y into two subsets or super classes, A and B . If there are only two classes in the response Y , the super classes are obvious. Otherwise, calculate the means and variances of X_j in each of the classes of Y . If the means are all equal, put the largest-sized class into group A and combine the rest to form group B . If they are not all equal, use a k -means clustering method ($k = 2$) on the class means to determine A and B .

X_j in A and in B is assumed to be normally distributed with estimated means $\bar{x}_{j|A}$, $\bar{x}_{j|B}$, and variances $S^2_{j|A}$, $S^2_{j|B}$, respectively. The quadratic discriminant is the partition $X_j \leq d$ and $X_j > d$ such that $\Pr(X_j, A) = \Pr(X_j, B)$. The discriminant rule assigns an observation to A if $x_{ij} \leq d$ and to B if $x_{ij} > d$. For d to maximally discriminate, the probabilities must be equal.

If the selected variable X_j is categorical, it is first transformed using the method outlined in Loh and Shih (1997) and then QDA is performed as above. The transformation is related to the discriminant coordinate (CRIMCOORD) approach due to [Gnanadesikan](#) (1977).

Minimal-Cost Complexity Pruning

One way to address overfitting is to grow the tree as large as possible, and then use some logic to prune it back. Let T represent a decision tree generated by any of the methods above. The idea (from Breiman, et. al.) is to find the smallest sub-tree of T that minimizes the cost complexity measure:

$$R_\delta(T) = R(T) + \delta |\tilde{T}|,$$

\tilde{T} denotes the set of terminal nodes, $|\tilde{T}|$ represents the number of terminal nodes, and $\delta \geq 0$ is a cost-complexity parameter. For a categorical target variable

$$R(T) = \sum_{t \in \tilde{T}} R(t) = \sum_{t \in \tilde{T}} r(t) p(t)$$

$$r(t) = \min_i \sum_j C(i|j) p(j|t)$$

$$p(t) = \Pr[x \in t],$$

$$\text{and } p(j|t) = \Pr[y = j | x \in t],$$

and $C(i|j)$ is the cost for misclassifying the actual class j as i . Note that $C(j|j) = 0$ and $C(i|j) > 0$, for $i \neq j$.

When the target is continuous (and the problem is a regression problem), the metric is instead the mean squared error

$$R(T) = \sum_{t \in \tilde{T}} R(t) = \frac{1}{N} \sum_{t \in \tilde{T}} \sum_{y_n \in t} (y_n - \hat{y}(t))^2$$

This software implements the optimal pruning algorithm 10.1, page 294 in Breiman, et. al (1984). The result of the algorithm is a sequence of sub-trees $T_{\max} > T_1 > T_2 > \dots > T_{M-1} > \{t_0\}$ obtained by pruning the fully generated tree, T_{\max} , until the sub-tree consists of the single root node, $\{t_0\}$. Corresponding to the sequence of sub-trees is the sequence of complexity values, $0 \leq \delta_{\min} = \delta_1 < \delta_2 < \dots < \delta_{M-1} < \delta_M$ where M is the number of steps it takes in the algorithm to reach the root node. The sub-trees represent the optimally pruned sub-trees for the sequence of complexity values. The minimum complexity δ_{\min} can be set via an optional argument.

V-Fold Cross-Validation

In V-fold cross validation, the training data is partitioned randomly into V approximately equally sized sub-samples. The model is then trained V different times with each of the sub-samples removed in turn. The cross-validated estimate of the risk of a decision function $R(d_k)$ is

$$R^{CV}(d_k) = \frac{1}{N} \sum_{v=1}^V \sum_{(x_n, y_n) \in \eta_v} L(y_n, d_k^v(x_n))$$

where $L(y, d(x))$ is the loss incurred when the decision is $d(x)$ for the actual, y . The symbol η denotes the full training data set, and d_k^v denotes the set of decisions corresponding to T_k^v , the k^{th} optimally pruned tree using the training sample $\eta - \eta_v$ and $\delta_k^v = \sqrt{\delta_k \delta_{k+1}}$, where δ_k, δ_{k+1} come from the pruning on the full data set, η .

For example, if the problem is classification, priors are estimated from the data N_j/N and T represents any tree,

$$\begin{aligned} R^{CV}(T) &= \sum_j R^{CV}(j) \hat{\pi}_j = \sum_j (\sum_i C(i|j) Q^{CV}(i|j)) \frac{N_j}{N} = \sum_j (\sum_i C(i|j) \frac{N^{ij}}{N_j}) \frac{N_j}{N} \\ &= \frac{1}{N} \sum_j (\sum_i C(i|j) N^{ij}) \end{aligned}$$

where

$$Q^*(i|j) = P(d(X) = i | Y = j)$$

$$R^*(j) = \sum_i C(i|j)Q^*(i|j)$$

$$R^*(d) = \sum_j R^*(j)\pi_j$$

and

$$N^{ij} = \sum_{v=1}^V N_v^{ij}$$

is the overall number of j cases misclassified as i . The standard error of $R^{CV}(d_k)$ is approximated with

$$SE\left(R^{CV}(d_k)\right) = \sqrt{s^2/N}$$

where

$$s^2 = \frac{1}{N} \sum_{\eta} [L(Y_n, d_k^{(v_n)}(x_n)) - R^{CV}(d_k)]^2$$

the “sample variance” of the cross-validated estimates.

Final selection rules include:

1. Select T_{k_1} such that $R^{CV}(T_{k_1}) = \min_k R^{CV}(T_k)$
2. Select T_{k_2} such that k_2 is the largest k satisfying $R^{CV}(T_{k_2}) \leq R^{CV}(T_{k_1}) + SE(R^{CV}(T_{k_1}))$.
3. For a specified complexity parameter δ_{fin} , select T_{k_3} such that $R^{CV}(T_{k_3}) \leq \min_k R^{CV}(T_k) + \delta_{fin} |\tilde{T}_k|$

Bagging

Bagging is a resampling approach for generating predictions. In particular, bagging stands for *bootstrap aggregating*. In the procedure, m bootstrap samples of size n are drawn from the training set of size n . Bootstrap sampling is *sampling with replacement* so that almost every sample has repeated observations, and some observations will

be left out. A decision tree is grown treating each sample as a separate training set. The tree is then used to generate predictions for the test data. For each test case, the m predictions are combined by averaging the output (regression) or voting (classification) to obtain a final prediction.

The bagged predictions are generated for the test data if test data is provided. Otherwise, the bagged predictions are generated for the input data (training data). The out-of-bag predictions are available outputs as well, but these are always for the input data. An out-of-bag prediction of a particular observation is combined using only those bootstrap samples that do not include that observation.

Bagging leads to "improvements for unstable procedures," such as neural nets, classification and regression trees, and subset selection in linear regression. On the other hand, it can mildly degrade the performance of stable methods such as K-nearest neighbors (Breiman, 1996).

Random Trees

A random forest is an ensemble of decision trees. Like bootstrap aggregation, a tree is fit to each of m bootstrap samples from the training data. Each tree is then used to generate predictions. For a regression problem (continuous response variable), the m predictions are combined into a single predicted value by averaging. For classification (categorical response variable), majority vote is used.

A random forest also randomizes the predictors. That is, in every tree, the splitting variable at every node is selected from a random subset of the predictors. Randomization of the predictors reduces correlation among individual trees. The random forest was invented by Leo Breiman in 2001 (Breiman, 2001). Random ForestsTM is the trademark term for this approach. Also see Hastie, Tibshirani, and Friedman, 2009, for further discussion.

To generate predictions or fitted values using a random forest, use the optional argument, `IMSLS_RANDOM_FEATURES`. The number of trees is equivalent to the number of bootstrap samples and can be set using `IMSLS_N_SAMPLE`. The number of random features can also be set using an optional argument.

Missing Values

Any observation or case with a missing response variable is eliminated from the analysis. If a predictor has a missing value, each algorithm will skip that case when evaluating the given predictor. When making a prediction for a new case, if the split variable is missing, the prediction function applies *surrogate* split-variables and splitting rules in turn, if they are estimated with the decision tree. Otherwise, the prediction function returns the prediction from the most recent non-terminal node. In this implementation, only ALACART estimates surrogate split variables when requested.

Structure Definitions

Table 44 – Structure `lmsls_f_decision_tree`

Name	Type	Description
<code>n_classes</code>	<i>int</i>	Number of classes assumed by the response variable, if the response variable is categorical
<code>n_levels</code>	<i>int</i>	Number of levels or depth of tree
<code>n_nodes</code>	<i>int</i>	Number of nodes or size of tree
<code>nodes</code>	<i>lmsls_f_tree_node*</i>	Pointer to an array of <code>tree_node</code> structures of size <code>n_nodes</code>
<code>n_preds</code>	<i>int</i>	Number of predictors used in the model
<code>n_surrogates</code>	<i>int</i>	Number of surrogate splits searched for at each node. Available for <code>method=1</code>
<code>pred_type</code>	<i>int*</i>	Pointer to an array of length <code>n_preds</code> containing the type of each predictor variable
<code>pred_n_values</code>	<i>int*</i>	Pointer to an array of length <code>n_preds</code> containing the number of values of each predictor variable
<code>response_type</code>	<i>int</i>	Type of the response variable
<code>terminal_nodes</code>	<i>int*</i>	Pointer to an array of length <code>n_nodes</code> indicating which nodes are terminal nodes

Table 45 – Structure `tree_node`

Name	Type	Description
<code>children_ids</code>	<i>int*</i>	Pointer to an array of length <code>n_children</code> containing the IDs of the children nodes
<code>cost</code>	<i>float</i>	Misclassification cost (in-sample cost measure at the current node)
<code>n_cases</code>	<i>int</i>	Number of cases of the training data that fall into the current node
<code>n_children</code>	<i>int</i>	Number of children of the current node
<code>node_id</code>	<i>int</i>	Node ID, where the root node corresponds to <code>node_id=0</code>
<code>node_prob</code>	<i>float*</i>	Estimate of the probability of a new case belonging to the node
<code>node_split_value</code>	<i>float</i>	Value around which the node will be split, if the node variable is of continuous type

Table 45 – Structure `tree_node`

<code>node_values_ind</code>	<i>int*</i>	Values of the split variable for the current node, if <code>node_var_id</code> has type 0 or 1
<code>node_var_id</code>	<i>int</i>	ID of the variable that defined the split in the parent node
<code>parent_id</code>	<i>int</i>	ID of the parent of the node with ID <code>node_id</code>
<code>predicted_class</code>	<i>int</i>	Predicted class at the current node, for response variables of categorical type
<code>predicted_val</code>	<i>float</i>	Predicted value of the response variable if the response variable is of continuous type
<code>surrogate_info</code>	<i>float*</i>	Array containing the surrogate split information
<code>y_probs</code>	<i>float*</i>	Pointer to the array of class probabilities at the current node, if the response variable is of categorical type

Examples

Example 1

In this example, we use a small data set with response variable, Play, which indicates whether a golfer plays (1) or does not play (0) golf under weather conditions measured by Temperature, Humidity, Outlook (Sunny (0), Overcast (1), Rainy (2)), and Wind (True (0), False (1)). A decision tree is generated by C4.5 and the ALACART methods. The control parameters are adjusted because of the small data size and no cross-validation or pruning is performed. The maximal trees are printed out using `lmsls_f_decision_tree_print`. Notice that C4.5 splits on Outlook, then Humidity and Wind, while ALACART splits on Outlook, then Temperature.

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

int main(){

    float xy[] =
    {
        0, 85, 85, 0, 0,
        0, 80, 90, 1, 0,
        1, 83, 78, 0, 1,
        2, 70, 96, 0, 1,
        2, 68, 80, 0, 1,
        2, 65, 70, 1, 0,
        1, 64, 65, 1, 1,
        0, 72, 95, 0, 0,
        0, 69, 70, 0, 1,
        2, 75, 80, 0, 1,
        0, 75, 70, 1, 1,
        1, 72, 90, 1, 1,
        1, 81, 75, 0, 1,
        2, 71, 80, 1, 0
```

```

};

int n = 14;
int ncols = 5;
int response_col_idx = 4;
int method = 1;
int var_type[] = {0, 2, 2, 0, 0};
int control[] = {2, 3, 10, 50, 10};

const char* names[] = {"Outlook", "Temperature", "Humidity", "Wind",
    "Play"};
const char* class_names[] = {"Don't Play", "Play"};
const char* var_levels[] = {"Sunny", "Overcast", "Rainy", "False", "True"};

imsls_f_decision_tree *tree = NULL;

tree=imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_N_FOLDS, 1,
    IMSLS_CONTROL, control,
    0);

printf("Decision Tree using Method C4.5:\n\n");
imsls_f_decision_tree_print(tree,
    IMSLS_VAR_NAMES, names,
    IMSLS_CLASS_NAMES, class_names,
    IMSLS_CATEG_NAMES, var_levels,
    0);

imsls_f_decision_tree_free(tree);

tree=imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_N_FOLDS, 1,
    IMSLS_METHOD, method,
    IMSLS_CONTROL, control,
    0);

printf("Decision Tree using Method ALACART:\n\n");
imsls_f_decision_tree_print(tree,
    IMSLS_VAR_NAMES, names,
    IMSLS_CLASS_NAMES, class_names,
    IMSLS_CATEG_NAMES, var_levels,
    0);

imsls_f_decision_tree_free(tree);
}

```

Output

```

Decision Tree using Method C4.5:

Decision Tree:

Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  4  5
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play
    Node 1: Cost = 0.143, N= 5, Level = 1, Child nodes:  2  3
    Rule:  Outlook  in: { Sunny }

```

```
P(Y=0)= 0.600
P(Y=1)= 0.400
Predicted Y: Don't Play
Node 2: Cost = 0.000, N= 2, Level = 2
Rule: Humidity <= 77.500
P(Y=0)= 0.000
P(Y=1)= 1.000
Predicted Y: Play
Node 3: Cost = 0.000, N= 3, Level = 2
Rule: Humidity > 77.500
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Y: Don't Play
Node 4: Cost = 0.000, N= 4, Level = 1
Rule: Outlook in: { Overcast }
P(Y=0)= 0.000
P(Y=1)= 1.000
Predicted Y: Play
Node 5: Cost = 0.143, N= 5, Level = 1, Child nodes: 6 7
Rule: Outlook in: { Rainy }
P(Y=0)= 0.400
P(Y=1)= 0.600
Predicted Y: Play
Node 6: Cost = 0.000, N= 3, Level = 2
Rule: Wind in: { False }
P(Y=0)= 0.000
P(Y=1)= 1.000
Predicted Y: Play
Node 7: Cost = 0.000, N= 2, Level = 2
Rule: Wind in: { True }
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Y: Don't Play
Decision Tree using Method ALACART:

Decision Tree:

Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes: 1 8
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y: Play
Node 1: Cost = 0.357, N= 10, Level = 1, Child nodes: 2 7
Rule: Outlook in: { Sunny Rainy }
P(Y=0)= 0.500
P(Y=1)= 0.500
Predicted Y: Don't Play
Node 2: Cost = 0.214, N= 8, Level = 2, Child nodes: 3 6
Rule: Temperature <= 77.500
P(Y=0)= 0.375
P(Y=1)= 0.625
Predicted Y: Play
Node 3: Cost = 0.214, N= 6, Level = 3, Child nodes: 4 5
Rule: Temperature <= 73.500
P(Y=0)= 0.500
P(Y=1)= 0.500
Predicted Y: Don't Play
Node 4: Cost = 0.071, N= 4, Level = 4
```

```

Rule: Temperature <= 70.500
P(Y=0)= 0.250
P(Y=1)= 0.750
Predicted Y: Play
Node 5: Cost = 0.000, N= 2, Level = 4
Rule: Temperature > 70.500
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Y: Don't Play
Node 6: Cost = 0.000, N= 2, Level = 3
Rule: Temperature > 73.500
P(Y=0)= 0.000
P(Y=1)= 1.000
Predicted Y: Play
Node 7: Cost = 0.000, N= 2, Level = 2
Rule: Temperature > 77.500
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Y: Don't Play
Node 8: Cost = 0.000, N= 4, Level = 1
Rule: Outlook in: { Overcast }
P(Y=0)= 0.000
P(Y=1)= 1.000
Predicted Y: Play

```

Example 2

This example applies the QUEST method to a simulated data set with 50 cases and three predictors of mixed-type. A maximally grown tree under the default controls and the optimally pruned sub-tree obtained from cross-validation and minimal cost complexity pruning are produced. Notice that the optimally pruned tree consists of just the root node, whereas the maximal tree has five nodes and three levels.

```

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

int main(){
    float xy[50*4] =
    {
        2, 25.928690, 0, 0,
        1, 51.632450, 1, 1,
        1, 25.784321, 0, 2,
        0, 39.379478, 0, 3,
        2, 24.650579, 0, 2,
        2, 45.200840, 0, 2,
        2, 52.679600, 1, 3,
        1, 44.283421, 1, 3,
        2, 40.635231, 1, 3,
        2, 51.760941, 0, 3,
        2, 26.303680, 0, 1,
        2, 20.702299, 1, 0,
        2, 38.742729, 1, 3,
        2, 19.473330, 0, 0,
        1, 26.422110, 0, 0,
        2, 37.059860, 1, 0,
        1, 51.670429, 1, 3,
        0, 42.401562, 0, 3,
        2, 33.900269, 1, 2,
    }
}

```

```
1, 35.432819, 0, 0,
1, 44.303692, 0, 1,
0, 46.723869, 0, 2,
1, 46.992619, 0, 2,
0, 36.059231, 0, 3,
2, 36.831970, 1, 1,
1, 61.662571, 1, 2,
0, 25.677139, 0, 3,
1, 39.085670, 1, 0,
0, 48.843410, 1, 1,
1, 39.343910, 0, 3,
2, 24.735220, 0, 2,
1, 50.552509, 1, 3,
0, 31.342630, 1, 3,
1, 27.157949, 1, 0,
0, 31.726851, 0, 2,
0, 25.004080, 0, 3,
1, 26.354570, 1, 3,
2, 38.123428, 0, 1,
0, 49.940300, 0, 2,
1, 42.457790, 1, 3,
0, 38.809479, 1, 1,
0, 43.227989, 1, 1,
0, 41.876240, 0, 3,
2, 48.078201, 0, 2,
0, 43.236729, 1, 0,
2, 39.412941, 0, 3,
1, 23.933460, 0, 2,
2, 42.841301, 1, 3,
2, 30.406691, 0, 1,
0, 37.773891, 0, 2
};

int n = 50;
int ncols = 4;
int method = 3;
int var_type[] = {0, 2, 0, 0};
int response_col_idx = 3;

Imsls_f_decision_tree *tree = NULL;

tree=imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_METHOD, method,
    IMSLS_RANDOM_SEED, 123457,
    IMSLS_PRINT, 1,
    0);

printf("\nMaximal tree: \n\n");
imsls_f_decision_tree_print(tree,
    IMSLS_PRINT_MAX,
    0);

printf("\nOptimally pruned subtree: \n\n");
imsls_f_decision_tree_print(tree, 0);

imsls_f_decision_tree_free(tree);
}
```

Output

Growing the maximal tree using method QUEST:

Cross-Validation:

Tree	complexity	CV-err	CV-Std.Error
0	0.00000	0.70406	0.08044
1	0.02000	0.72641	0.08562
2	0.04000	0.72814	0.08598

Select tree number 2, cost complexity parameter = 0.04000:

Maximal tree:

Decision Tree:

Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes: 1 2

P(Y=0)= 0.180

P(Y=1)= 0.180

P(Y=2)= 0.260

P(Y=3)= 0.380

Predicted Y: 3

Node 1: Cost = 0.220, N= 17, Level = 1

Rule: X1 <= 35.031

P(Y=0)= 0.294

P(Y=1)= 0.118

P(Y=2)= 0.353

P(Y=3)= 0.235

Predicted Y: 2

Node 2: Cost = 0.360, N= 33, Level = 1, Child nodes: 3 4

Rule: X1 > 35.031

P(Y=0)= 0.121

P(Y=1)= 0.212

P(Y=2)= 0.212

P(Y=3)= 0.455

Predicted Y: 3

Node 3: Cost = 0.180, N= 19, Level = 2

Rule: X1 <= 43.265

P(Y=0)= 0.211

P(Y=1)= 0.211

P(Y=2)= 0.053

P(Y=3)= 0.526

Predicted Y: 3

Node 4: Cost = 0.160, N= 14, Level = 2

Rule: X1 > 43.265

P(Y=0)= 0.000

P(Y=1)= 0.214

P(Y=2)= 0.429

P(Y=3)= 0.357

Predicted Y: 2

Optimally pruned subtree:

Decision Tree:

Node 0: Cost = 0.620, N= 50, Level = 0

P(Y=0)= 0.180

P(Y=1)= 0.180

```
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y: 3
Pruned at Node id 0.
```

Example 3

This example uses the dataset Kyphosis. The 81 cases represent 81 children who have undergone surgery to correct a type of spinal deformity known as Kyphosis. The response variable is the presence or absence of Kyphosis after the surgery. Three predictors are Age of the patient in months, Start, the number of the vertebra where the surgery started, and Number, the number of vertebra involved in the surgery. This example uses the method QUEST to produce a maximal tree. It also requests predictions for a test-data set consisting of 10 “new” cases.

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

int main()
{
    float xy[81*4] =
    {
        0, 71, 3, 5,
        0, 158, 3, 14,
        1, 128, 4, 5,
        0, 2, 5, 1,
        0, 1, 4, 15,
        0, 1, 2, 16,
        0, 61, 2, 17,
        0, 37, 3, 16,
        0, 113, 2, 16,
        1, 59, 6, 12,
        1, 82, 5, 14,
        0, 148, 3, 16,
        0, 18, 5, 2,
        0, 1, 4, 12,
        0, 168, 3, 18,
        0, 1, 3, 16,
        0, 78, 6, 15,
        0, 175, 5, 13,
        0, 80, 5, 16,
        0, 27, 4, 9,
        0, 22, 2, 16,
        1, 105, 6, 5,
        1, 96, 3, 12,
        0, 131, 2, 3,
        1, 15, 7, 2,
        0, 9, 5, 13,
        0, 8, 3, 6,
        0, 100, 3, 14,
        0, 4, 3, 16,
        0, 151, 2, 16,
        0, 31, 3, 16,
        0, 125, 2, 11,
        0, 130, 5, 13,
        0, 112, 3, 16,
        0, 140, 5, 11,
        0, 93, 3, 16,
```

```
0, 1, 3, 9,
1, 52, 5, 6,
0, 20, 6, 9,
1, 91, 5, 12,
1, 73, 5, 1,
0, 35, 3, 13,
0, 143, 9, 3,
0, 61, 4, 1,
0, 97, 3, 16,
1, 139, 3, 10,
0, 136, 4, 15,
0, 131, 5, 13,
1, 121, 3, 3,
0, 177, 2, 14,
0, 68, 5, 10,
0, 9, 2, 17,
1, 139, 10, 6,
0, 2, 2, 17,
0, 140, 4, 15,
0, 72, 5, 15,
0, 2, 3, 13,
1, 120, 5, 8,
0, 51, 7, 9,
0, 102, 3, 13,
1, 130, 4, 1,
1, 114, 7, 8,
0, 81, 4, 1,
0, 118, 3, 16,
0, 118, 4, 16,
0, 17, 4, 10,
0, 195, 2, 17,
0, 159, 4, 13,
0, 18, 4, 11,
0, 15, 5, 16,
0, 158, 5, 14,
0, 127, 4, 12,
0, 87, 4, 16,
0, 206, 4, 10,
0, 11, 3, 15,
0, 178, 4, 15,
1, 157, 3, 13,
0, 26, 7, 13,
0, 120, 2, 13,
1, 42, 7, 6,
0, 36, 4, 13
};

float xy_test[10*4] =
{
0, 71, 3, 5,
1, 128, 4, 5,
0, 1, 4, 15,
0, 61, 6, 10,
0, 113, 2, 16,
1, 82, 5, 14,
0, 148, 3, 16,
0, 1, 4, 12,
0, 1, 3, 16,
0, 175, 5, 13
};
```

```

int n = 81;
int ncols = 4;
int response_col_idx = 0;
int method = 3;
int control[] = {5, 10, 10, 50, 10};
int var_type[] = {0, 2, 2, 2};

int n_test = 10;
int i, idx;

float *predictions;
float pred_err_ss;
const char* names[] = {"Age", "Number", "Start"};
const char* class_names[] = {"Absent", "Present"};
const char* response_name[] = {"Kyphosis"};
imsls_f_decision_tree *tree = NULL;

tree=imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_METHOD, method,
    IMSLS_N_FOLDS, 1,
    IMSLS_CONTROL, control,
    IMSLS_TEST_DATA, n_test, xy_test,
    IMSLS_PRINT, 2,
    IMSLS_PREDICTED, &predictions,
    IMSLS_ERROR_SS, &pred_err_ss,
    0);

imsls_f_decision_tree_print(tree,
    IMSLS_RESP_NAME, response_name,
    IMSLS_VAR_NAMES, names,
    IMSLS_CLASS_NAMES, class_names,
    0);

printf("\nPredictions for test data:\n");
printf("%5s%8s%7s%10s\n", names[0], names[1], names[2],
    response_name[0]);
for(i=0;i<n_test;i++){
    printf("%5.0f%8.0f%7.0f",
        xy_test[i*ncols+1],
        xy_test[i*ncols+2],
        xy_test[i*ncols+3]);
    idx = (int)predictions[i];
    printf("%10s\n", class_names[idx]);
}
printf("\nMean squared prediction error: %f\n", pred_err_ss);

imsls_f_decision_tree_free(tree);
imsls_free(predictions);
}

```

Output

```

The response variable has 0 missing values.
Growing the maximal tree using method QUEST:
Node 2 is a terminal node. It has 7 cases--too few cases to split.
Node 3 is a terminal node. It has 6 cases--too few cases to split.
Node 5 is a terminal node. It has 6 cases--too few cases to split.
Node 8 is an terminal node. The split is too thin having count 2.
Node 10 is a terminal node. It has 6 cases--too few cases to split.

```

Node 11 is a terminal node, because it is pure.
Node 11 is a terminal node. It has 7 cases--too few cases to split.
Node 13 is a terminal node. It has 5 cases--too few cases to split.
Node 14 is a terminal node, because it is pure.

Decision Tree:

Node 0: Cost = 0.210, N= 81, Level = 0, Child nodes: 1 4
P(Y=0)= 0.790
P(Y=1)= 0.210
Predicted Kyphosis Absent
Node 1: Cost = 0.074, N= 13, Level = 1, Child nodes: 2 3
Rule: Start <= 5.155
P(Y=0)= 0.538
P(Y=1)= 0.462
Predicted Kyphosis Absent
Node 2: Cost = 0.025, N= 7, Level = 2
Rule: Age <= 84.030
P(Y=0)= 0.714
P(Y=1)= 0.286
Predicted Kyphosis Absent
Node 3: Cost = 0.025, N= 6, Level = 2
Rule: Age > 84.030
P(Y=0)= 0.333
P(Y=1)= 0.667
Predicted Kyphosis Present
Node 4: Cost = 0.136, N= 68, Level = 1, Child nodes: 5 6
Rule: Start > 5.155
P(Y=0)= 0.838
P(Y=1)= 0.162
Predicted Kyphosis Absent
Node 5: Cost = 0.012, N= 6, Level = 2
Rule: Start <= 8.862
P(Y=0)= 0.167
P(Y=1)= 0.833
Predicted Kyphosis Present
Node 6: Cost = 0.074, N= 62, Level = 2, Child nodes: 7 12
Rule: Start > 8.862
P(Y=0)= 0.903
P(Y=1)= 0.097
Predicted Kyphosis Absent
Node 7: Cost = 0.062, N= 28, Level = 3, Child nodes: 8 9
Rule: Start <= 13.092
P(Y=0)= 0.821
P(Y=1)= 0.179
Predicted Kyphosis Absent
Node 8: Cost = 0.025, N= 15, Level = 4
Rule: Age <= 91.722
P(Y=0)= 0.867
P(Y=1)= 0.133
Predicted Kyphosis Absent
Node 9: Cost = 0.037, N= 13, Level = 4, Child nodes: 10 11
Rule: Age > 91.722
P(Y=0)= 0.769
P(Y=1)= 0.231
Predicted Kyphosis Absent
Node 10: Cost = 0.037, N= 6, Level = 5
Rule: Number <= 3.450
P(Y=0)= 0.500
P(Y=1)= 0.500

```

Predicted Kyphosis Absent
Node 11: Cost = 0.000, N= 7, Level = 5
Rule: Number > 3.450
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Kyphosis Absent
Node 12: Cost = 0.012, N= 34, Level = 3, Child nodes: 13 14
Rule: Start > 13.092
P(Y=0)= 0.971
P(Y=1)= 0.029
Predicted Kyphosis Absent
Node 13: Cost = 0.012, N= 5, Level = 4
Rule: Start <= 14.864
P(Y=0)= 0.800
P(Y=1)= 0.200
Predicted Kyphosis Absent
Node 14: Cost = 0.000, N= 29, Level = 4
Rule: Start > 14.864
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Kyphosis Absent

```

Predictions for test data:

Age	Number	Start	Kyphosis
71	3	5	Absent
128	4	5	Present
1	4	15	Absent
61	6	10	Absent
113	2	16	Absent
82	5	14	Absent
148	3	16	Absent
1	4	12	Absent
1	3	16	Absent
175	5	13	Absent

Mean squared prediction error: 0.010000

Example 4

For the Kyphosis dataset of *Example 3*, this example produces random forest predictions using the optional arguments for random feature selection.

```

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

#define NOBS 81
#define NCLASSES 2
#define NPRED 3
#define NTEST 10

int main(){
    float xy[81 * 4] = {
        0, 71, 3, 5,
        0, 158, 3, 14,
        1, 128, 4, 5,
        0, 2, 5, 1,
        0, 1, 4, 15,
        0, 1, 2, 16,

```

```
0, 61, 2, 17,
0, 37, 3, 16,
0, 113, 2, 16,
1, 59, 6, 12,
1, 82, 5, 14,
0, 148, 3, 16,
0, 18, 5, 2,
0, 1, 4, 12,
0, 168, 3, 18,
0, 1, 3, 16,
0, 78, 6, 15,
0, 175, 5, 13,
0, 80, 5, 16,
0, 27, 4, 9,
0, 22, 2, 16,
1, 105, 6, 5,
1, 96, 3, 12,
0, 131, 2, 3,
1, 15, 7, 2,
0, 9, 5, 13,
0, 8, 3, 6,
0, 100, 3, 14,
0, 4, 3, 16,
0, 151, 2, 16,
0, 31, 3, 16,
0, 125, 2, 11,
0, 130, 5, 13,
0, 112, 3, 16,
0, 140, 5, 11,
0, 93, 3, 16,
0, 1, 3, 9,
1, 52, 5, 6,
0, 20, 6, 9,
1, 91, 5, 12,
1, 73, 5, 1,
0, 35, 3, 13,
0, 143, 9, 3,
0, 61, 4, 1,
0, 97, 3, 16,
1, 139, 3, 10,
0, 136, 4, 15,
0, 131, 5, 13,
1, 121, 3, 3,
0, 177, 2, 14,
0, 68, 5, 10,
0, 9, 2, 17,
1, 139, 10, 6,
0, 2, 2, 17,
0, 140, 4, 15,
0, 72, 5, 15,
0, 2, 3, 13,
1, 120, 5, 8,
0, 51, 7, 9,
0, 102, 3, 13,
1, 130, 4, 1,
1, 114, 7, 8,
0, 81, 4, 1,
0, 118, 3, 16,
0, 118, 4, 16,
0, 17, 4, 10,
```

```
    0, 195, 2, 17,
    0, 159, 4, 13,
    0, 18, 4, 11,
    0, 15, 5, 16,
    0, 158, 5, 14,
    0, 127, 4, 12,
    0, 87, 4, 16,
    0, 206, 4, 10,
    0, 11, 3, 15,
    0, 178, 4, 15,
    1, 157, 3, 13,
    0, 26, 7, 13,
    0, 120, 2, 13,
    1, 42, 7, 6,
    0, 36, 4, 13 };

float xytest[10 * 4] = { 0, 71, 3, 5,
    1, 128, 4, 5,
    0, 1, 4, 15,
    0, 61, 6, 10,
    0, 113, 2, 16,
    1, 82, 5, 14,
    0, 148, 3, 16,
    0, 1, 4, 12,
    0, 1, 3, 16,
    0, 175, 5, 13 };

int N = NOBS;
int nclasses = NCLASSES;
int npreds = NPREDS;
int ncols = NPREDS + 1;
int response_idx = 0;
int var_type[] = { 0, 2, 2, 2 };
int ntest = NTEST;
int i;
long seed = 123457;
float *predictions = NULL;
Imsls_f_decision_tree *tree = NULL;

tree = imsls_f_decision_tree(N, ncols,
    xy,
    response_idx,
    var_type,
    IMSLS_METHOD, 1,
    IMSLS_N_FOLDS, 1,
    IMSLS_PREDICTED, &predictions,
    IMSLS_TEST_DATA, ntest, xytest,
    0);

printf("Single tree predictions vs. actuals:\n\n");
for (i = 0; i < ntest; i++){
    printf("%d\t%f \t %f\n", i + 1, predictions[i],
        xytest[i*ncols + response_idx]);
}

imsls_f_decision_tree_free(tree);
imsls_free(predictions);

tree = imsls_f_decision_tree(N, ncols,
    xy,
```

```

        response_idx,
        var_type,
        IMSLS_METHOD, 1,
        IMSLS_N_FOLDS, 1,
        IMSLS_PREDICTED, &predictions,
        IMSLS_TEST_DATA, ntest, xytest,
        IMSLS_RANDOM_FEATURES,
        IMSLS_N_RANDOM_FEATURES, 2,
        IMSLS_RANDOM_SEED, seed,
        IMSLS_N_SAMPLE, 100,
        0);

    printf("\n\nRandom forest predictions vs. actuals:\n\n");
    for (i = 0; i < ntest; i++){
        printf("%d\t%f\t%f\n", i + 1, predictions[i],
            xytest[i*ncols + response_idx]);
    }

    imsls_f_decision_tree_free(tree);
    imsls_free(predictions);
}

```

Output

Single tree predictions vs. actuals:

1	0.000000	0.000000
2	0.000000	1.000000
3	0.000000	0.000000
4	1.000000	0.000000
5	0.000000	0.000000
6	0.000000	1.000000
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000
10	0.000000	0.000000

Random forest predictions vs. actuals:

1	0.000000	0.000000
2	1.000000	1.000000
3	0.000000	0.000000
4	0.000000	0.000000
5	0.000000	0.000000
6	0.000000	1.000000
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000
10	0.000000	0.000000

Example 5

In example 5, the random forest is used to produce predictions for Fisher's Iris data.

```

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

#define NOBS 150

```

```
#define NCLASSES 3
#define NPREDS 4

int main(){
    int i = 0;
    int n = NOBS;
    int ncols = NPREDS + 1;
    int nclasses = NCLASSES;
    int response_idx = 0;
    int var_type[] = { 0, 2, 2, 2, 2 };

    float iris_xy[150 * 5];
    float *iris_data = NULL;
    char *classLabel[] = { "Setosa", "Versicolour", "Virginica",
        "Total" };
    char *colLabel[] = { "Species", "Number of Errors",
        "Total N" };

    float out_of_bag_mean_error = 0.0;
    float *out_of_bag_class_errors = NULL;
    Imsls_f_decision_tree *tree = NULL;

    iris_data = imsls_f_data_sets(3, 0);

    for (i = 0; i < n*ncols; i++){
        iris_xy[i] = iris_data[i];
    }
    for (i = 0; i < n; i++){
        iris_xy[i*ncols + response_idx] -= 1;
    }

    tree = imsls_f_decision_tree(n, ncols,
        iris_xy,
        response_idx,
        var_type,
        IMSLS_METHOD, 1,
        IMSLS_N_FOLDS, 1,
        IMSLS_OUT_OF_BAG_MEAN_ERROR, &out_of_bag_mean_error,
        IMSLS_OUT_OF_BAG_CLASS_ERROR, &out_of_bag_class_errors,
        IMSLS_RANDOM_FEATURES,
        IMSLS_RANDOM_SEED, 123457,
        0);

    imsls_f_write_matrix("Out of bag errors by class", nclasses + 1, 2,
        out_of_bag_class_errors,
        IMSLS_ROW_LABELS, classLabel,
        IMSLS_COL_LABELS, colLabel, 0);

    printf("\nOut-of-bag mean error = %3.2f\n", out_of_bag_mean_error);

    imsls_f_decision_tree_free(tree);
    imsls_free(out_of_bag_class_errors);
    imsls_free(iris_data);
}
```

Output

Out of bag errors by class		
Species	Number of Errors	Total N
Setosa	0	50
Versicolour	3	50
Virginica	6	50
Total	9	150
Out-of-bag mean error = 0.06		

Example 6

In this example, a random forest is used to predict the categorical response on simulated data. The data is random with no real relationship among the predictors and the response variable, reflected by the high number of errors. Furthermore, the variable importance measure is slightly negative for the predictors, another symptom of noisy data.

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

#define NOBS 50
#define NCLASSES 3
#define NPREDS 3

int main(){
    int n = NOBS;
    int nclasses = NCLASSES;
    int ncols = NPREDS + 1;
    int var_type[] = { 0, 2, 0, 0 };
    int response_idx = 0;
    float xy[50 * 4] =
    {
        2, 25.92869, 0, 0,
        1, 51.63245, 1, 1,
        1, 25.78432, 0, 2,
        0, 39.37948, 0, 3,
        2, 24.65058, 0, 2,
        2, 45.20084, 0, 2,
        2, 52.67960, 1, 3,
        1, 44.28342, 1, 3,
        2, 40.63523, 1, 3,
        2, 51.76094, 0, 3,
        2, 26.30368, 0, 1,
        2, 20.70230, 1, 0,
        2, 38.74273, 1, 3,
        2, 19.47333, 0, 0,
        1, 26.42211, 0, 0,
        2, 37.05986, 1, 0,
        1, 51.67043, 1, 3,
        0, 42.40156, 0, 3,
        2, 33.90027, 1, 2,
        1, 35.43282, 0, 0,
        1, 44.30369, 0, 1,
        0, 46.72387, 0, 2,
        1, 46.99262, 0, 2,
        0, 36.05923, 0, 3,
        2, 36.83197, 1, 1,
    }
```

```

1, 61.66257, 1, 2,
0, 25.67714, 0, 3,
1, 39.08567, 1, 0,
0, 48.84341, 1, 1,
1, 39.34391, 0, 3,
2, 24.73522, 0, 2,
1, 50.55251, 1, 3,
0, 31.34263, 1, 3,
1, 27.15795, 1, 0,
0, 31.72685, 0, 2,
0, 25.00408, 0, 3,
1, 26.35457, 1, 3,
2, 38.12343, 0, 1,
0, 49.94030, 0, 2,
1, 42.45779, 1, 3,
0, 38.80948, 1, 1,
0, 43.22799, 1, 1,
0, 41.87624, 0, 3,
2, 48.07820, 0, 2,
0, 43.23673, 1, 0,
2, 39.41294, 0, 3,
1, 23.93346, 0, 2,
2, 42.84130, 1, 3,
2, 30.40669, 0, 1,
0, 37.77389, 0, 2
};

float out_of_bag_mean_error = 0.0;
float *out_of_bag_class_errors = NULL;
float *variable_importance = NULL;
imsls_f_decision_tree *tree = NULL;

tree = imsls_f_decision_tree(n, ncols,
    xy,
    response_idx,
    var_type,
    IMSLS_METHOD, 0,
    IMSLS_N_FOLDS, 1,
    IMSLS_OUT_OF_BAG_MEAN_ERROR, &out_of_bag_mean_error,
    IMSLS_OUT_OF_BAG_CLASS_ERROR, &out_of_bag_class_errors,
    IMSLS_OUT_OF_BAG_VAR_IMPORTANCE, &variable_importance,
    IMSLS_RANDOM_FEATURES,
    IMSLS_RANDOM_SEED, 123457,
    0);

imsls_f_write_matrix("Errors by class", nclasses + 1, 2,
    out_of_bag_class_errors, 0);
printf("\nout-of-bag mean error = %f\n", out_of_bag_mean_error);
imsls_f_write_matrix("Variable importance", ncols - 1, 1,
    variable_importance, 0);

imsls_f_decision_tree_free(tree);
imsls_free(out_of_bag_class_errors);
imsls_free(variable_importance);
}

```

Output

```

Errors by class
1          2

```

1	11	15
2	14	16
3	15	19
4	40	50

out-of-bag mean error = 0.800000

Variable importance

1	-0.01758
2	0.00133
3	-0.00881

Example 7

In this example, the bagged trees generated in a random forest are returned.

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

#define NOBS 150
#define NPREDS 4

int main(){
    int i = 0;
    int n = NOBS;
    int ncols = NPREDS + 1;
    int response_idx = 0;
    int control[] = { 7, 21, 10, 4, 3 };
    int var_type[] = { 0, 2, 2, 2, 2 };
    float iris_xy[150 * 5];
    float *iris_data = NULL;
    Imsls_f_decision_tree *tree = NULL;
    Imsls_f_decision_tree **bagged_trees = NULL;

    iris_data = imsls_f_data_sets(3, 0);

    for (i = 0; i < n*ncols; i++){
        iris_xy[i] = iris_data[i];
    }
    for (i = 0; i < n; i++){
        iris_xy[i*ncols + response_idx] -= 1;
    }

    tree = imsls_f_decision_tree(n, ncols,
        iris_xy,
        response_idx,
        var_type,
        IMSLS_METHOD, 1,
        IMSLS_N_FOLDS, 1,
        IMSLS_CONTROL, control,
        IMSLS_RETURN_TREES, &bagged_trees,
        IMSLS_RANDOM_FEATURES,
        IMSLS_RANDOM_SEED, 123457,
        0);

    /* Print the first and the last bagged tree:*/
    imsls_f_decision_tree_print(bagged_trees[0], 0);
    imsls_f_decision_tree_print(bagged_trees[49], 0);
}
```

```
imsls_f_decision_tree_free(tree);
imsls_f_bagged_trees_free(50, bagged_trees);
imsls_free(iris_data);
}
```

Output

Decision Tree:

Node 0: Cost = 0.633, N= 150, Level = 0, Child nodes: 1 2

P(Y=0)= 0.347

P(Y=1)= 0.287

P(Y=2)= 0.367

Predicted Y: 2

Node 1: Cost = 0.113, N= 67, Level = 1

Rule: X0 <= 5.550

P(Y=0)= 0.746

P(Y=1)= 0.224

P(Y=2)= 0.030

Predicted Y: 0

Node 2: Cost = 0.200, N= 83, Level = 1

Rule: X0 > 5.550

P(Y=0)= 0.024

P(Y=1)= 0.337

P(Y=2)= 0.639

Predicted Y: 2

Decision Tree:

Node 0: Cost = 0.660, N= 150, Level = 0, Child nodes: 1 2

P(Y=0)= 0.333

P(Y=1)= 0.340

P(Y=2)= 0.327

Predicted Y: 1

Node 1: Cost = 0.000, N= 50, Level = 1

Rule: X2 <= 2.600

P(Y=0)= 1.000

P(Y=1)= 0.000

P(Y=2)= 0.000

Predicted Y: 0

Node 2: Cost = 0.327, N= 100, Level = 1

Rule: X2 > 2.600

P(Y=0)= 0.000

P(Y=1)= 0.510

P(Y=2)= 0.490

Predicted Y: 1

Warning Errors

IMSLS_NO_SURROGATES	Use of surrogates is limited to method 1 (ALACART).
IMSLS_NO_CONVERGENCE	Convergence was not achieved.
IMSLS_EMPTY_CLASS_LEVEL	The count of class level # in the training data is zero.

Fatal Errors

IMSLS_INVALID_METHOD2	Choose a valid tree generation method; 0 (C4.5), 1 (ALACART), 2 (CHAID) or 3 (QUEST).
IMSLS_VALUE_GT_ZERO	The value of # must be strictly positive.

decision_tree_predict

Computes predicted values using a decision tree.

Synopsis

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

```
float* imsls_f_decision_tree_predict (int n, int n_cols, float x[ ], int var_type[ ],  
    imsls_f_decision_tree *tree, ..., 0)
```

The type *double* function is `imsls_d_decision_tree_predict`.

Required Arguments

int n (Input)

The the number of rows in **x**.

int n_cols (Input)

The number of columns in **x**.

float x[] (Input)

Array of size **n** × **n_cols** containing the data.

int var_type[] (Input)

Array of length **n_cols** indicating the type of each variable.

	Type
0	Categorical
1	Ordered Discrete (Low, Med., High)
2	Quantitative or Continuous
3	Ignore this variable

*imsls_f_decision_tree *tree* (Input)

An estimated decision tree.

Return Value

An array of length `n` containing the predicted values. If an error occurs, `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float* imsls_f_decision_tree_predict (int n, int n_cols, float x[ ], int var_type[ ],
    imsls_f_decision_tree *tree,
    IMSLS_N_SURROGATES, int n_surrogates,
    IMSLS_X_RESPONSE_COL, int response_col_idx,
    IMSLS_WEIGHTS, float weights[ ],
    IMSLS_X_NODE_IDS, int **node_ids,
    IMSLS_X_NODE_IDS_USER, int node_ids[ ],
    IMSLS_ERROR_SS, float *pred_err_ss,
    IMSLS_RETURN_USER, float predictions[ ],
    0)
```

Optional Arguments

`IMSLS_N_SURROGATES, int n_surrogates` (Input)

Indicates the number of surrogate splits for use in methods that find surrogate splits in order to handle missing values.

Default: `n_surrogates = 0`.

`IMSLS_WEIGHTS, float weights[]` (Input)

An array of length `n` containing case weights.

Default: `weights[i]=1.0`.

`IMSLS_X_RESPONSE_COL, int response_col_idx` (Input)

The column index of the response variable, if present in the data. A negative value indicates there is no response column.

Default: `response_col_idx = -1`.

`IMSLS_X_NODE_IDS, int **node_ids` (Output)

Address of a pointer to the internally allocated array of length `n` containing for each row in `x`, the terminal node of the tree to which the observation belongs.

`IMSLS_X_NODE_IDS_USER, int node_ids[]` (Output)

Storage for `node_ids` is provided by the user.

`IMSLS_ERROR_SS`, *float** `pred_err_ss` (Output)

The prediction error mean sum of squares, available when values for the response are present in the data.

`IMSLS_RETURN_USER`, *float* `predictions[]` (Input)

Storage for the return value is provided by the user.

Description

To predict a new set of cases using a fitted or estimated decision tree, `imsls_f_decision_tree_predict` finds the terminal node of the tree to which each new case belongs. The predicted value is then the predicted value of that node. This is a matter of “putting the data through the tree.” For example, suppose the following weather conditions:

Temperature = 70

Humidity = 82

Outlook = Rainy

Wind = FALSE

According to the C4.5 decision tree in Example 1 for `imsls_f_decision_tree`, will the golfer play golf or not, under these conditions? The tree splits the root node on Outlook into three nodes: {Sunny, Rainy, and Overcast}. Rainy defines node 5. Node 5 is split into child nodes 6 and 7, according to the presence of wind. If there is wind, Node 7, the prediction is “Don’t Play.” If there is no wind, Node 6, the prediction is “Play.” Therefore, the new observation belongs to Node 6, and the tree predicts that the golfer will play under the given weather conditions. In the ALACART decision tree, Node 4 is the terminal node, and the associated prediction is “Play.”

Comments

1. Users can request predictions and error sum of squares directly from `imsls_f_decision_tree` or use this separate prediction function when it is not necessary to re-estimate a decision tree.
2. If requested, the prediction mean sum of squared error (mean squared prediction error) is computed when actual response values are available in the data.
3. For cases with missing values in predictors that are involved in the splitting rules of the tree, `imsls_f_decision_tree_predict` uses surrogate rules if available and when requested. Otherwise, predicted values are missing, and the error sum of squares does include that case.

Example

Using the kyphosis data of Example 2 for `imsls_f_decision_tree`, this example illustrates using a separate call to `imsls_f_decision_tree_predict` to obtain the predicted values for a new set of observations (`xy_test`).

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

int main()
{
    float xy[81*4] =
    {
        0, 71, 3, 5,
        0, 158, 3, 14,
        1, 128, 4, 5,
        0, 2, 5, 1,
        0, 1, 4, 15,
        0, 1, 2, 16,
        0, 61, 2, 17,
        0, 37, 3, 16,
        0, 113, 2, 16,
        1, 59, 6, 12,
        1, 82, 5, 14,
        0, 148, 3, 16,
        0, 18, 5, 2,
        0, 1, 4, 12,
        0, 168, 3, 18,
        0, 1, 3, 16,
        0, 78, 6, 15,
        0, 175, 5, 13,
        0, 80, 5, 16,
        0, 27, 4, 9,
        0, 22, 2, 16,
        1, 105, 6, 5,
        1, 96, 3, 12,
        0, 131, 2, 3,
        1, 15, 7, 2,
        0, 9, 5, 13,
        0, 8, 3, 6,
        0, 100, 3, 14,
        0, 4, 3, 16,
        0, 151, 2, 16,
        0, 31, 3, 16,
        0, 125, 2, 11,
        0, 130, 5, 13,
        0, 112, 3, 16,
        0, 140, 5, 11,
        0, 93, 3, 16,
        0, 1, 3, 9,
        1, 52, 5, 6,
        0, 20, 6, 9,
        1, 91, 5, 12,
        1, 73, 5, 1,
        0, 35, 3, 13,
        0, 143, 9, 3,
        0, 61, 4, 1,
```

```
0, 97, 3, 16,
1, 139, 3, 10,
0, 136, 4, 15,
0, 131, 5, 13,
1, 121, 3, 3,
0, 177, 2, 14,
0, 68, 5, 10,
0, 9, 2, 17,
1, 139, 10, 6,
0, 2, 2, 17,
0, 140, 4, 15,
0, 72, 5, 15,
0, 2, 3, 13,
1, 120, 5, 8,
0, 51, 7, 9,
0, 102, 3, 13,
1, 130, 4, 1,
1, 114, 7, 8,
0, 81, 4, 1,
0, 118, 3, 16,
0, 118, 4, 16,
0, 17, 4, 10,
0, 195, 2, 17,
0, 159, 4, 13,
0, 18, 4, 11,
0, 15, 5, 16,
0, 158, 5, 14,
0, 127, 4, 12,
0, 87, 4, 16,
0, 206, 4, 10,
0, 11, 3, 15,
0, 178, 4, 15,
1, 157, 3, 13,
0, 26, 7, 13,
0, 120, 2, 13,
1, 42, 7, 6,
0, 36, 4, 13
};

float xy_test[10*4] =
{
    0, 71, 3, 5,
    1, 128, 4, 5,
    0, 1, 4, 15,
    0, 61, 6, 10,
    0, 113, 2, 16,
    1, 82, 5, 14,
    0, 148, 3, 16,
    0, 1, 4, 12,
    0, 1, 3, 16,
    0, 175, 5, 13
};

int n = 81;
int ncols = 4;
int response_col_idx = 0;
int method = 3;
int control[] = {5, 10, 10, 50, 10};
int var_type[] = {0, 2, 2, 2};
```

```

int n_test = 10;
int i, idx;

float *predictions;
float pred_err_ss;
const char* names[] = {"Age", "Number", "Start"};
const char* classNames[] = {"Absent", "Present"};
const char* responseName[] = {"Kyphosis"};
Imsls_f_decision_tree *tree = NULL;

tree = imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_METHOD, method,
    IMSLS_N_FOLDS, 1,
    IMSLS_CONTROL, control,
    IMSLS_TEST_DATA, n_test, xy_test,
    0);

predictions = imsls_f_decision_tree_predict(n_test, ncols, xy_test,
    var_type, tree,
    IMSLS_X_RESPONSE_COL, response_col_idx,
    IMSLS_ERROR_SS, &pred_err_ss,
    0);

printf("\nPredictions for test data:\n");
printf("%5s%8s%7s%10s\n", names[0], names[1], names[2],
    responseName[0]);
for(i=0; i<n_test; i++){
    printf("%5.0f%8.0f%7.0f",
        xy_test[i*ncols+1],
        xy_test[i*ncols+2],
        xy_test[i*ncols+3]);
    idx = (int)predictions[i];
    printf("%10s\n", classNames[idx]);
}
printf("\nMean squared prediction error: %f\n", pred_err_ss);

imsls_f_decision_tree_free(tree);
imsls_free(predictions);
}

```

Output

```

Predictions for test data:
Age Number Start Kyphosis
 71      3      5  Absent
128      4      5  Present
  1      4     15  Absent
 61      6     10  Absent
113      2     16  Absent
 82      5     14  Absent
148      3     16  Absent
  1      4     12  Absent
  1      3     16  Absent
175      5     13  Absent

```

```

Mean squared prediction error: 0.100000

```

Warning Errors

IMSLS_NO_SURROGATES

Use of surrogates is limited to method 1 (ALACART).

IMSLS_INVALID_PARAM

The value of # is out of range.

decision_tree_print

Prints a decision tree.

Synopsis

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

```
void imsls_f_decision_tree_print (Imsls_f_decision_tree *tree, ..., 0)
```

The type *double* function is `imsls_d_decision_tree_print`.

Required Arguments

imsls_f_decision_tree *tree (Input)
An estimated decision tree.

Synopsis with Optional Arguments

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

```
void imsls_f_decision_tree_print (Imsls_f_decision_tree *tree,  
    IMSLS_RESP_NAME, char *response_name,  
    IMSLS_VAR_NAMES, char *names[ ],  
    IMSLS_CLASS_NAMES, char *class_names[ ],  
    IMSLS_CATEG_NAMES, char *categ_names[ ],  
    IMSLS_PRINT_MAX,  
    0)
```

Optional Arguments

`IMSLS_RESP_NAME, char *response_name` (Input)
An array of length 1 containing a pointer to a character string representing the name of the response variable.
Default: `response_name[0] = "Y"`.

IMSLS_VAR_NAMES, *char *var_names[]* (Input)

An array of length `tree->npreds` containing pointers to character strings representing the names of the predictors.

Default: `var_names[0]="X0"`, `var_names[1]="X1"`, etc.

IMSLS_CLASS_NAMES, *char *class_names[]* (Input)

An array of length `tree->nclasses` containing pointers to character strings representing the names of the different classes in Y , assuming Y is of categorical type.

Default: `class_names[0]="0"`, `class_names[1]="1"`, etc.

IMSLS_CATEG_NAMES, *char *categ_names[]* (Input)

An array of length `tree->pred_nvalues[0] + tree->pred_nvalues[1] + ... + tree->pred_nvalues[tree->npreds-1]` containing pointers to character strings representing the names of the different category levels for each predictor of categorical type.

Default: `categ_names[0]="0"`, `categ_names[1]="1"`, etc.

IMSLS_PRINT_MAX, (Input)

If present, the maximal tree is printed despite any pruning information.

Default: Accounts for pruning.

Description

Function `imsls_f_decision_tree_print` provides a convenient way to quickly see the structure of the tree. More elaborate visualization methods or summaries can be written for the decision tree structure described in [Structure Definitions](#) for function `decision_tree`, and [Figure 22](#) in the *Overview* section].

Comments

1. The nodes are labeled as the tree was grown. In other words, the first child of the root node is labeled Node 1, the first child node of Node 1 is labeled Node 2, and so on, until the branch stops growing. The numbering continues with the most recent split one level up.
2. If the tree has fewer than five levels, each new level is indented. Otherwise, there is no indentation.

Example

This example operates on simulated categorical data.

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

int main()
```

```
{
float xy[30*3] =
{
    2, 0, 2,
    1, 0, 0,
    2, 1, 3,
    0, 1, 0,
    1, 2, 0,
    2, 2, 3,
    2, 2, 3,
    0, 1, 0,
    0, 0, 0,
    0, 1, 0,
    1, 2, 0,
    2, 0, 2,
    0, 2, 0,
    2, 0, 1,
    0, 0, 0,
    2, 0, 1,
    1, 0, 0,
    0, 2, 0,
    2, 0, 1,
    1, 2, 0,
    0, 2, 2,
    2, 1, 3,
    1, 1, 0,
    2, 2, 3,
    1, 2, 0,
    2, 2, 3,
    2, 0, 1,
    2, 1, 3,
    1, 2, 0,
    1, 1, 0
};

int n = 30;
int ncols = 3;
int response_col_idx= 2;
int var_type[] = {0, 0, 0};
int control[] = {5, 10, 10, 50, 10};

const char* names[] = {"Var1", "Var2"};
const char* class_names[] = {"c1", "c2", "c3", "c4"};
const char* response_name = "Response";
const char* var_levels[] = {"L1", "L2", "L3", "A", "B", "C"};
Imsls_f_decision_tree *tree = NULL;

tree = imsls_f_decision_tree(n, ncols, xy, response_col_idx, var_type,
    IMSLS_CONTROL, control,
    0);

printf("\nGenerated labels:\n");
imsls_f_decision_tree_print(tree,
    IMSLS_PRINT_MAX,
    0);
printf("\nCustom labels:\n");

imsls_f_decision_tree_print(tree,
    IMSLS_RESP_NAME, &response_name,
```

```

    IMSLS_VAR_NAMES, names,
    IMSLS_CATEG_NAMES, var_levels,
    IMSLS_CLASS_NAMES, class_names,
    IMSLS_PRINT_MAX,
    0);

    imsls_f_decision_tree_free(tree);
}

```

Output

Generated labels:

Decision Tree:

Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes: 1 2 3

P(Y=0)= 0.533

P(Y=1)= 0.133

P(Y=2)= 0.100

P(Y=3)= 0.233

Predicted Y: 0

Node 1: Cost = 0.033, N= 8, Level = 1

Rule: X0 in: { 0 }

P(Y=0)= 0.875

P(Y=1)= 0.000

P(Y=2)= 0.125

P(Y=3)= 0.000

Predicted Y: 0

Node 2: Cost = 0.000, N= 9, Level = 1

Rule: X0 in: { 1 }

P(Y=0)= 1.000

P(Y=1)= 0.000

P(Y=2)= 0.000

P(Y=3)= 0.000

Predicted Y: 0

Node 3: Cost = 0.200, N= 13, Level = 1

Rule: X0 in: { 2 }

P(Y=0)= 0.000

P(Y=1)= 0.308

P(Y=2)= 0.154

P(Y=3)= 0.538

Predicted Y: 3

Custom labels:

Decision Tree:

Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes: 1 2 3

P(Y=0)= 0.533

P(Y=1)= 0.133

P(Y=2)= 0.100

P(Y=3)= 0.233

Predicted Response c1

Node 1: Cost = 0.033, N= 8, Level = 1

Rule: Var1 in: { L1 }

P(Y=0)= 0.875

P(Y=1)= 0.000

P(Y=2)= 0.125

P(Y=3)= 0.000

Predicted Response c1

Node 2: Cost = 0.000, N= 9, Level = 1

Rule: Var1 in: { L2 }

P(Y=0)= 1.000

P(Y=1)= 0.000

P(Y=2)= 0.000

P(Y=3)= 0.000

Predicted Response c1

Node 3: Cost = 0.200, N= 13, Level = 1

Rule: Var1 in: { L3 }

P(Y=0)= 0.000

P(Y=1)= 0.308

P(Y=2)= 0.154

P(Y=3)= 0.538

Predicted Response c4

decision_tree_free

Frees the memory associated with a decision tree.

Synopsis

```
#include <ims1s.h>

void ims1s_f_decision_tree_free (ims1s_f_decision_tree *tree)
```

The type *double* function is `ims1s_d_decision_tree_free`.

Required Arguments

ims1s_f_decision_tree *tree (Input)
A decision tree structure.

Description

`ims1s_f_decision_tree_free` frees the memory associated with a decision tree structure.

Example

See `ims1s_f_decision_tree`, [Example 1](#).

bagged_trees_free

Frees the memory associated with an array of decision trees generated in the bagging procedure.

Synopsis

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

```
void imsls_f_bagged_trees_free(int ntrees, imsls_f_decision_tree **trees)
```

The type *double* function is `imsls_d_bagged_trees_free`.

Required Arguments

int `ntrees` (Input)

The number of decision tree structures.

imsls_f_decision_tree `**trees` (Input)

An array of size `ntrees` containing the decision tree structures.

Description

`imsls_f_bagged_trees_free` frees the memory associated with an array of decision tree structures.

Example

See `imsls_f_decision_tree`, [Example 7](#).

gradient_boosting

Performs stochastic gradient boosting of decision trees.

Synopsis

```
#include <imspls.h>
```

```
float impls_f_gradient_boosting (int n, int n_cols, float xy[ ], int response_col_idx,  
    int var_type[], ..., 0)
```

The type *double* function is `impls_d_gradient_boosting`.

Required Arguments

int **n** (Input)

The number of rows in **xy**.

int **n_cols** (Input)

The number of columns in **xy**.

float **xy[]** (Input)

Array of size **n** × **n_cols** containing the data.

int **response_col_idx** (Input)

The column index of **xy** containing the response variable.

int **var_type[]** (Input)

Array of length **n_cols** indicating the type of each variable.

var_type[i]	Description
0	Categorical
1	Ordered Discrete (e.g., Low, Med, High)
2	Quantitative or Continuous
3	Ignore this variable

Note: When the variable type is specified as Categorical (`var_type[i] = 0`), the numbering of the categories must begin at 0. For example, if there are three categories, they must be represented as 0, 1, and 2 in the `xy` array.

The number of classes for a categorical response variable is determined by the largest value discovered in the data. Note that a warning message is displayed if a class level in 0, 1, ..., `n_classes - 1` has a 0 count in the data.

Return Value

A pointer to an array of predicted values on the test data if test data is provided (see optional argument, [IMSLS_TEST_DATA](#)). If test data is not provided, the predicted values are the fitted values on the training data. If an error occurs, `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_gradient_boosting (int n, int n_cols, float xy[ ], int response_col_idx,
    int var_type[ ],

    IMSLS_TEST_DATA, int n_test, float xy_test[ ],
    IMSLS_TEST_DATA_WEIGHTS, float weights_test[ ],
    IMSLS_WEIGHTS, float weights[ ],
    IMSLS_N_SAMPLE, int sample_size,
    IMSLS_SAMPLE_PROPORTION, float sample_p,
    IMSLS_SHRINKAGE, float shrinkage,
    IMSLS_MAX_ITER, int max_iter,
    IMSLS_LOSS_FCN, int loss_fcn_type,
    IMSLS_ALPHA, float huber_alpha,
    IMSLS_CONTROL, int params[ ],
    IMSLS_RANDOM_SEED, int seed,
    IMSLS_PRINT, int print_level,
    IMSLS_LOSS_VALUE, float *loss_value,
```

```

IMSLS_TEST_LOSS_VALUE, float *test_loss_value,
IMSLS_FITTED_VALUES, float **fitted_values,
IMSLS_FITTED_VALUES_USER, float fitted_values[],
IMSLS_PROBABILITIES, float **probs,
IMSLS_PROBABILITIES_USER, float probs[],
IMSLS_FITTED_PROBABILITIES, float **fitted_probs,
IMSLS_FITTED_PROBABILITIES_USER, float fitted_probs[],
IMSLS_RETURN_TREES, Imsls_f_decision_tree ***bagged_trees,
IMSLS_RETURN_USER, float predictions[],
0)

```

Optional Arguments

IMSLS_TEST_DATA, *int* `n_test`, *float* `xy_test[]` (Input)

`xy_test` is an array of size `n_test × n_cols` containing test data for which predictions are requested. When this optional argument is present, the number of observations `n_test` must be greater than 0. The response variable may have missing values in `xy_test`, but it must be in the same column as it is in `xy` and the predictors must be in the same columns as they are in `xy`. If the test data is not provided but predictions are requested, then `xy` is used, and the predictions are the fitted values.

Default: `n_test = n`, `xy_test = xy`.

IMSLS_TEST_DATA_WEIGHTS, *float* `weights_test[]` (Input)

An array of size `n_test` containing the frequencies or weights for each observation in `xy_test`. This argument is ignored if **IMSLS_TEST_DATA** is not present.

Default: `weights_test[i] = 1.0`.

IMSLS_WEIGHTS, *float* `weights[]` (Input)

An array of length `n` containing frequencies or weights for each observation in `xy`.

Default: `weights[i] = 1.0`.

IMSLS_N_SAMPLE, *int* `sample_size` (Input)

The number of examples to be drawn randomly from the training data in each iteration.

Default: `sample_size = sample_p*n`.

`IMSLS_SAMPLE_PROPORTION`, *float* `sample_p` (Input)

The proportion of the training examples to be drawn randomly from the training data in each iteration.

Default: `sample_p` = 0.5.

`IMSLS_SHRINKAGE`, *float* `shrinkage` (Input)

The shrinkage parameter to be used in the boosting algorithm. The parameter must be in the interval [0,1] inclusive.

Default: `shrinkage` = 1.0 (no shrinkage).

`IMSLS_MAX_ITER`, *int* `max_iter` (Input)

The number of iterations. This value is equivalent to M in the boosting algorithm described below.

Default: `max_iter` = 50.

`IMSLS_LOSS_FCN`, *int* `loss_fcn_type` (Input)

An integer specifying the loss function to use in the algorithm for regression problems

(`loss_fcn_type` = 0, 1, 2) or binary classification problems (`loss_fcn_type` = 3, 4).

See the [Description](#) section for the loss function in the multinomial case (categorical response variables with more than two outcomes).

Default: `loss_fcn_type = 0`.

Name	loss_fcn_type	Definition
Least Squares	0	The loss function is the sum of squared error: $L = \sum_{i=1}^n (y_i - f(x_i))^2$
Least Absolute Deviation	1	The loss function is the sum of absolute errors: $L = \sum_{i=1}^n y_i - f(x_i) $
Huber M	2	The loss function is the weighted mixture of squared error and absolute error: $L = \sum_{i=1}^n \Psi(y_i, f(x_i))$ where $\Psi(y, z) = \begin{cases} 0.5(y - z)^2 & y - z \leq \delta \\ \delta(y - z - 0.5\delta) & y - z > \delta \end{cases}$ and where δ is the α empirical quantile of the errors, $(y_i - f(x_i)), i = 1, \dots, n$.
Adaboost	3	The loss function is the AdaBoost.M1 criterion: $L = \sum_{i=1}^n \exp(-(2y_i - 1)f(x_i))$
Bernoulli or binomial deviance	4	The loss function is the binomial or Bernoulli negative log-likelihood: $L = -2 \sum_{i=1}^n (y_i f(x_i) - \log(1 + \exp(f(x_i))))$

IMSLS_ALPHA, *float* `huber_alpha` (Input)

The quantile value for the Huber-M loss function.

Default: `huber_alpha = 0.05`.

IMSLS_CONTROL, *int* `params` [] (Input)

Array of length 5 containing parameters to control the size and other characteristics of the decision trees.

<code>params[i]</code>	Name	Action
0	<code>min_n_node</code>	Do not split a node if one of its child nodes will have fewer than <code>min_n_node</code> observations.
1	<code>min_split</code>	Do not split a node if the node has fewer than <code>min_split</code> observations.
2	<code>max_x_cats</code>	Allow for up to <code>max_x_cats</code> number of categories or levels for categorical variables.
3	<code>max_size</code>	Stop growing the tree once it has reached <code>max_size</code> number of nodes.
4	<code>max_depth</code>	Stop growing the tree once it has reached <code>max_depth</code> number of levels.

Default: `params` [] = {10, 21, 10, 4, 10}.

IMSLS_RANDOM_SEED, *int* `seed` (Input)

Sets the seed of the random number generator used in sampling. Using the same seed in repeated calls will result in the same output. If `seed` = 0, the random seed is set by the system clock and repeated calls result in slightly different results.

Default: `seed` = 0.

IMSLS_PRINT, *int* `print_level` (Input)

<code>print_level</code>	Action
0	No printing
1	Print final results only
2	Print intermediate and final results

Default: `print_level` = 0.

IMSLS_LOSS_VALUE, *float* *`loss_value` (Output)

The final value of the loss function after *M* iterations of the algorithm.

IMSLS_TEST_LOSS_VALUE, *float* *`test_loss_value` (Output)

The final value of the loss function after *M* iterations of the algorithm on the test data.

IMSLS_FITTED_VALUES, *float* **`fitted_values` (Output)

Address of a pointer to an array of length *n* containing the fitted values on the training data *xy* after *M* iterations of the algorithm.

`IMSLS_FITTED_VALUES_USER`, *float* `fitted_values[]` (Output)

Storage for the array of the fitted values for the training data is provided by the user.

`IMSLS_PROBABILITIES`, *float ***`probs` (Output)

Address of a pointer to an array of length `n*n_classes` containing the predicted class probabilities for each observation in the test data.

`IMSLS_PROBABILITIES_USER`, *float* `probs[]` (Output)

Storage for the array of the predicted class probabilities is provided by the user.

`IMSLS_FITTED_PROBABILITIES`, *float ***`fitted_probabilities` (Output)

Address of a pointer to an array of length `n*n_classes` containing the fitted class probabilities on the training data for classification problems.

`IMSLS_FITTED_PROBABILITIES_USER`, *float* `fitted_probabilities[]` (Output)

Storage for the array of the fitted class probabilities is provided by the user.

`IMSLS_RETURN_TREES`, *Imsls_f_decision_tree ****`bagged_trees` (Output)

Address of a pointer to an array of length `M` containing the collection of trees generated during the algorithm. To release this space, use [imsls_f_bagged_trees_free](#).

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

Storage for the array of the return value is provided by the user.

Description

Stochastic gradient boosting is an optimization algorithm for minimizing residual errors to improve the accuracy of predictions. This function implements the algorithm by [Friedman \(1999\)](#). For further discussion, see [Hastie, et al. \(2009\)](#).

In the following, x_i is the vector of predictor variable values, and y_i is the response variable value in the observation at row i . The function $f_m(x_i)$ evaluated at x_i is the predicted value in a particular iteration, m . This value is iteratively reweighted to minimize a loss function. Specifically, the algorithm is:

Initialize the predictor function to the constant

$$f_0(x) = \arg \min_{\gamma} \sum_{i=1}^n L(y_i, \gamma)$$

For each iteration $m = 1, 2, \dots, M$,

1. Calculate the pseudo-residuals

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right] \Big|_{f=f_{m-1}}$$

2. Fit a regression tree to the pseudo-residuals r_{im} and use the resulting models to predict the observations in the training data. The resulting terminal nodes define J_m terminal regions R_{jm} for the response. Compute

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

3. Update the prediction function for each observation, x_i ,

$$f_m(x_i) = f_{m-1}(x_i) + \lambda \sum_{j=1}^{J_m} \gamma_{jm} I(x_i \in R_{jm})$$

where $\lambda \in [0,1]$ is a shrinkage parameter ($\lambda = 1$ means no shrinking, whereas $\lambda = 0$ gives just $f_M = f_0$).

After M iterations, the function $f_M(\cdot)$ forms the basis of the predictions for the response variable.

Specifically

Response variable type	Definition
QUANTITATIVE_CONTINUOUS	For the regression problem, the predicted value at a new observation vector x_i is $\hat{y}_i = f_M(x_i)$

Response variable type	Definition
CATEGORICAL with 2 outcomes (binomial)	<p>For a classification problem with 2 outcomes, the predicted probability is</p> $p_i = \Pr[y_i = 1] = \frac{\exp(f_M(x_i))}{1 + \exp(f_M(x_i))}$ <p>Then the predicted value is</p> $\hat{y}_i = I\{p_i > 0.5\}$ <p>where $I\{\cdot\}$ is the indicator function.</p>
CATEGORICAL with 3 or more outcomes (multinomial)	<p>For a classification problem with $K \geq 3$ outcomes, the predicted probabilities for $k = 1, \dots, K$ are</p> $p_{ik} = \frac{\exp(f_{kM}(x_i))}{\sum_{j=1}^K \exp(f_{jM}(x_i))}$ <p>Then the predicted value is</p> $\hat{y}_i = \arg \max_k \{p_{ik}\}$

For regression problems, the algorithm uses the squared error loss by default. For classification problems with two categories, the Bernoulli or binomial loss function is the default (see optional argument `IMSLS_LOSS_FCN`). For a categorical response with three or more categories, the multinomial deviance (described below) is used.

For a categorical response with K categories, the loss function is the multinomial negative log-likelihood, or multinomial deviance:

$$L = -2 \sum_{i=1}^n \sum_{k=1}^K y_{ik} \log(p_{ik})$$

where

$$P_{ik} = \frac{\exp(f_k(x_i))}{\sum_{j=1}^K \exp(f_j(x_i))}$$

Examples

Example 1

This example uses stochastic gradient boosting to obtain fitted values for a regression variable on a small data set with six predictor variables.

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

#define ROW 61
#define COL 7

int main(){

    float XY[ROW][COL] = {
        { 4.45617685, 0.8587425048, 1.2705688183, 0.0, 0.0, 1.0, 0.836626959 },
        { 3.01895357, 0.8928761308, 1.3886538362, 2.0, 1.0, 2.0, 2.155131825 },
        { 5.16899757, 0.7385954093, 1.5773203815, 0.0, 4.0, 2.0, 0.075368922 },
        { -0.23062048, 0.6227398487, 0.0228797458, 3.0, 4.0, 2.0, 0.070793233 },
        { 2.43144968, 0.8519553537, 1.2141886768, 2.0, 4.0, 2.0, 0.762200702 },
        { 2.28255119, 0.5578103897, 0.9185446175, 2.0, 4.0, 2.0, 0.085492814 },
        { 4.51650903, 0.4178302658, 1.3686663737, 0.0, 0.0, 0.0, 2.573941051 },
        { 5.42996967, 0.9829705667, 0.7817731784, 0.0, 5.0, 1.0, 0.865016054 },
        { 0.99551212, 0.3859238869, 0.2746516233, 3.0, 4.0, 0.0, 1.908151819 },
        { 1.23525017, 0.4165328839, 1.3154437956, 3.0, 4.0, 2.0, 2.752358041 },
        { 1.51599306, 0.2008399745, 0.9003028921, 3.0, 0.0, 2.0, 1.437127559 },
        { 2.72854297, 0.2072261081, 1.2282209327, 2.0, 5.0, 2.0, 0.68596562 },
        { 3.06956138, 0.9067490781, 0.8283077031, 2.0, 0.0, 2.0, 2.862403627 },
        { 1.81659279, 0.4506153886, 1.2822537781, 3.0, 4.0, 2.0, 1.710525684 },
        { 3.75978142, 0.2638894715, 0.4995447062, 0.0, 1.0, 1.0, 1.077172402 },
        { 5.72383445, 0.7682430062, 1.4758595745, 0.0, 3.0, 1.0, 2.365233736 },
        { 3.78155015, 0.6888140934, 0.4809393724, 0.0, 0.0, 1.0, 1.061246069 },
        { 3.60023233, 0.8470419827, 1.6149122352, 1.0, 1.0, 0.0, 0.01120048 },
        { 4.30238917, 0.9484412405, 1.6122899544, 1.0, 4.0, 2.0, 0.782038861 },
        { -0.19206757, 0.7674867723, 0.01665624, 3.0, 5.0, 2.0, 2.924944949 },
        { 3.03246318, 0.8747456241, 1.6051767552, 2.0, 1.0, 0.0, 2.233971364 },
        { 1.56652306, 0.0947128241, 1.470864601, 3.0, 0.0, 1.0, 1.851705944 },
        { 2.77490671, 0.1347932827, 1.3693161067, 1.0, 2.0, 0.0, 0.795709459 },
        { 1.05042043, 0.258093959, 0.4679728113, 3.0, 5.0, 0.0, 2.897785557 },
        { 2.73366469, 0.152943752, 0.5244769375, 1.0, 4.0, 2.0, 2.712871963 },
        { 1.78996951, 0.7921472492, 0.4686144991, 2.0, 4.0, 1.0, 1.295327727 },
        { 1.10343272, 0.123231777, 0.563989053, 2.0, 4.0, 1.0, 0.510414582 },
        { 1.70883743, 0.1931027549, 1.8561577178, 3.0, 5.0, 1.0, 0.165721288 },
        { 2.17977731, 0.316932481, 1.3376214528, 2.0, 2.0, 0.0, 2.366607214 },
        { 2.46127675, 0.9601344266, 0.2090187217, 1.0, 3.0, 1.0, 0.846218965 },
        { 1.92249547, 0.1104206559, 1.739415036, 3.0, 0.0, 0.0, 0.652622544 },
        { 5.81907137, 0.7049566596, 1.6238740934, 0.0, 3.0, 0.0, 1.685337845 },
        { 2.04774497, 0.0480224835, 0.7510998738, 2.0, 5.0, 2.0, 1.400641323 },
```

```

    { 4.54023907, 0.0557708007, 1.0864350675, 0.0, 1.0, 1.0, 1.630408823 },
    { 3.66100874, 0.2939440177, 0.9709178614, 0.0, 1.0, 0.0, 0.06970193 },
    { 4.39253655, 0.0982369843, 1.2492676578, 0.0, 2.0, 2.0, 0.138188998 },
    { 3.23303353, 0.3775206071, 0.2937129182, 0.0, 0.0, 2.0, 1.070823081 },
    { 3.13800098, 0.7891691434, 1.90897633, 2.0, 3.0, 0.0, 1.240732062 },
    { 1.49034639, 0.2456938969, 0.9157859818, 3.0, 5.0, 0.0, 0.850803277 },
    { 0.09486277, 0.1240615626, 0.3891524528, 3.0, 5.0, 0.0, 2.532516038 },
    { 3.74460501, 0.0181218453, 1.4921644945, 1.0, 2.0, 1.0, 1.92839241 },
    { 3.24158796, 0.9203409508, 1.1644667462, 2.0, 3.0, 1.0, 1.956283022 },
    { 1.97796767, 0.5977597698, 0.5501609747, 2.0, 5.0, 2.0, 0.39384095 },
    { 4.15214037, 0.1433333508, 1.4292114358, 1.0, 0.0, 0.0, 1.114095218 },
    { 0.7799787, 0.8539819908, 0.7039108537, 3.0, 0.0, 1.0, 1.468978726 },
    { 2.01869009, 0.8919721926, 1.1436212659, 3.0, 4.0, 1.0, 2.09256257 },
    { 0.56311561, 0.0899261576, 0.7989077698, 3.0, 5.0, 0.0, 0.195650739 },
    { 4.74296429, 0.9625684835, 1.5732420743, 0.0, 3.0, 2.0, 2.685061853 },
    { 2.97981809, 0.5511086562, 1.6053283028, 2.0, 5.0, 2.0, 0.906810926 },
    { 2.82187135, 0.3869563073, 0.9321342241, 1.0, 5.0, 1.0, 0.756223386 },
    { 5.24390592, 0.3500950718, 1.7769328682, 0.0, 3.0, 2.0, 1.328165314 },
    { 3.17307157, 0.8798056154, 1.4647966106, 2.0, 5.0, 1.0, 0.561835038 },
    { 0.78246075, 0.1472158518, 0.4658273738, 2.0, 0.0, 0.0, 1.317240539 },
    { 1.57827027, 0.3415432149, 0.7513634153, 2.0, 2.0, 0.0, 1.502675544 },
    { 0.84104905, 0.1501226462, 0.9332020828, 3.0, 1.0, 2.0, 1.083374695 },
    { 2.63627352, 0.1707233109, 1.1676406977, 2.0, 3.0, 0.0, 2.236639737 },
    { 1.30863625, 0.2616807753, 0.8342161868, 3.0, 2.0, 2.0, 1.778402721 },
    { 2.7313073, 0.9616109401, 1.596915911, 3.0, 3.0, 1.0, 0.303127344 },
    { 3.56848173, 0.4072918599, 1.5345127448, 1.0, 2.0, 2.0, 1.47452504 },
    { 5.40152982, 0.7796053565, 1.3659530994, 0.0, 4.0, 1.0, 0.484531098 },
    { 3.94901823, 0.5052344366, 1.9319026601, 1.0, 2.0, 0.0, 2.504392843 },
};
int i;
int response_col_idx = 0;
int var_type[] = { 2, 2, 2, 0, 0, 2 };
float *fitted_values = NULL;
float loss_value;

fitted_values = imsls_f_gradient_boosting(ROW, COL, &XY[0][0], response_col_idx,
var_type,
IMSLS_RANDOM_SEED, 123457,
IMSLS_LOSS_VALUE, &loss_value,
0);

printf("Fitted values vs actuals:\n");
for (i = 0; i < ROW; i++){
    printf("\t\t%5.3f\t\t%5.3f\n", fitted_values[i], XY[i][response_col_idx]);
}

printf("\nLoss value: %5.5f\n", loss_value);
imsls_free(fitted_values);
}

```

Output

```

Fitted values vs actuals:
    4.956    4.456
    2.908    3.019
    5.105    5.169
    0.229   -0.231
    2.124    2.431

```

2.338	2.283
4.333	4.517
5.273	5.430
0.734	0.996
1.491	1.235
1.359	1.516
2.611	2.729
2.275	3.070
1.875	1.817
3.233	3.760
6.246	5.724
3.676	3.782
3.981	3.600
3.742	4.302
0.716	-0.192
3.367	3.032
1.975	1.567
3.105	2.775
0.471	1.050
2.386	2.734
1.307	1.790
1.370	1.103
1.709	1.709
2.371	2.180
3.386	2.461
1.404	1.922
5.822	5.819
2.207	2.048
4.028	4.540
3.831	3.661
4.824	4.393
3.451	3.233
3.451	3.138
2.285	1.490
0.471	0.095
3.728	3.745
3.204	3.242
1.474	1.978
4.291	4.152
0.937	0.780
1.709	2.019
0.534	0.563
5.608	4.743
3.376	2.980
2.749	2.822
5.656	5.244
3.192	3.173
1.659	0.782
1.851	1.578
0.435	0.841
2.880	2.636
1.869	1.309
2.201	2.731
3.517	3.568
5.263	5.402
3.584	3.949

Loss value: 0.08452

Example 2

This example uses stochastic gradient boosting to obtain probability estimates for a binary response variable and four predictor variables. An estimate of $P[Y = 0]$ is obtained for each example in the training data as well as a small test data set.

Probabilities ≤ 0.5 lead to a prediction of $Y = 0$, while probabilities > 0.5 lead to a prediction of $Y = 1.0$.

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

#define TROW 100
#define TCOL 5
#define TSTROW 10

int main(){
    float training_data[TROW][TCOL] = {
        { 0.0, 0.4223019897, 1.7540411302, 3.0, 0.763836258 },
        { 0.0, 0.0907259332, 0.8722643796, 2.0, 1.859006285 },
        { 0.0, 0.1384744535, 0.838324877, 1.0, 0.249729405 },
        { 1.0, 0.5435024537, 1.2359190206, 4.0, 0.831992314 },
        { 0.0, 0.8359154933, 1.8527500411, 1.0, 1.089201049 },
        { 1.0, 0.3577950741, 0.3652825342, 3.0, 2.204364955 },
        { 1.0, 0.6799094002, 0.6610595905, 3.0, 1.44730419 },
        { 0.0, 0.5821297709, 1.6180879478, 1.0, 2.957565282 },
        { 1.0, 0.8229457375, 1.0201675948, 3.0, 2.872570117 },
        { 0.0, 0.0633462721, 0.4140600134, 1.0, 0.63906323 },
        { 1.0, 0.1019134156, 0.0677204356, 3.0, 1.493447564 },
        { 0.0, 0.1551713238, 1.541201456, 3.0, 1.90219884 },
        { 1.0, 0.8273822817, 0.2114979578, 3.0, 2.855730173 },
        { 0.0, 0.7955570114, 1.8757067556, 2.0, 2.930132627 },
        { 0.0, 0.6537275917, 1.2139678737, 2.0, 1.535853243 },
        { 1.0, 0.1243124125, 1.5130919744, 4.0, 2.733670775 },
        { 0.0, 0.2163864174, 0.7051185896, 2.0, 2.755841087 },
        { 0.0, 0.2522670308, 1.2821007571, 2.0, 0.342119491 },
        { 0.0, 0.8677104027, 1.9003869346, 2.0, 2.454376481 },
        { 1.0, 0.8670932774, 0.7993045617, 4.0, 2.732812615 },
        { 0.0, 0.5384287981, 0.1856947718, 1.0, 1.838702635 },
        { 0.0, 0.7236269342, 0.4993310347, 1.0, 1.030699128 },
        { 0.0, 0.0789361731, 1.011216166, 1.0, 2.539607478 },
        { 1.0, 0.7631686032, 0.0536725423, 2.0, 1.401761686 },
        { 0.0, 0.1157020777, 0.0123261618, 1.0, 2.098372295 },
        { 1.0, 0.1451248352, 1.9153951635, 3.0, 0.492650534 },
        { 1.0, 0.8497178114, 1.80941298, 4.0, 2.653985489 },
        { 0.0, 0.8027864883, 1.2631045617, 3.0, 2.716214291 },
        { 0.0, 0.798560373, 0.6872106791, 2.0, 2.763023936 },
        { 1.0, 0.1816879204, 0.4323868025, 4.0, 0.098090197 },
        { 1.0, 0.6301239238, 0.3670980479, 3.0, 0.02313788 },
        { 1.0, 0.0411311248, 0.0173408454, 3.0, 1.994786958 },
        { 1.0, 0.0427366099, 0.8114635572, 3.0, 2.966069741 },
        { 1.0, 0.4107826762, 0.1929467283, 4.0, 0.573832348 },
        { 0.0, 0.9441903098, 0.0729898885, 1.0, 1.710992303 },
        { 1.0, 0.3597549822, 0.2799857073, 2.0, 0.969428934 },
        { 0.0, 0.3741368004, 1.6052779425, 2.0, 1.866030486 },
        { 0.0, 0.3515911719, 0.3383029872, 1.0, 2.639469598 },
        { 0.0, 0.9184092905, 1.7116801264, 1.0, 1.380178652 },
        { 1.0, 0.77803064, 1.9830028405, 3.0, 1.834021992 },
```

```
{ 0.0, 0.573786814, 0.0258851023, 1.0, 1.52130144 },
{ 1.0, 0.3279244492, 0.6977945678, 4.0, 1.322451157 },
{ 0.0, 0.7924819048, 0.3694838509, 1.0, 2.369654865 },
{ 0.0, 0.9787846403, 1.1470323382, 2.0, 0.037156113 },
{ 1.0, 0.6910662795, 0.1019420708, 2.0, 2.58588334 },
{ 0.0, 0.1367050812, 0.6635301332, 2.0, 0.368273583 },
{ 0.0, 0.2826360366, 1.4468787988, 1.0, 2.705811968 },
{ 0.0, 0.4524727969, 0.7885378413, 2.0, 0.851228449 },
{ 0.0, 0.5118664701, 1.061143666, 1.0, 0.249325278 },
{ 0.0, 0.9965170731, 0.2068265025, 2.0, 0.9210639 },
{ 1.0, 0.7801500652, 1.565742691, 4.0, 1.827419217 },
{ 0.0, 0.2906187973, 1.7036567871, 2.0, 2.842997725 },
{ 0.0, 0.1753704017, 0.7124397112, 2.0, 1.262811961 },
{ 1.0, 0.7796778064, 0.3478030777, 3.0, 0.90719801 },
{ 1.0, 0.3889356288, 1.1771452101, 4.0, 1.298438454 },
{ 0.0, 0.9374473374, 1.1879778663, 1.0, 1.854424331 },
{ 1.0, 0.1939157653, 0.093336341, 4.0, 0.166025681 },
{ 1.0, 0.2023756928, 0.0623724433, 3.0, 0.536441906 },
{ 0.0, 0.1691352043, 1.1587338657, 2.0, 2.15494096 },
{ 1.0, 0.0921523357, 0.2247394961, 3.0, 2.006995301 },
{ 0.0, 0.819186907, 0.0392292971, 1.0, 1.282159743 },
{ 0.0, 0.9458126165, 1.5268264762, 1.0, 1.960050194 },
{ 0.0, 0.1373939656, 1.8025095677, 2.0, 0.633624267 },
{ 0.0, 0.0555424779, 0.5022063241, 2.0, 0.639495004 },
{ 1.0, 0.3581428374, 1.4436954968, 3.0, 1.408938169 },
{ 1.0, 0.1189418568, 0.8011626904, 4.0, 0.210266769 },
{ 1.0, 0.5782070206, 1.58215921, 3.0, 2.648622607 },
{ 0.0, 0.460689794, 0.0704823257, 1.0, 1.45671379 },
{ 0.0, 0.6959878858, 0.2245675903, 2.0, 1.849515461 },
{ 0.0, 0.1930288749, 0.6296302159, 2.0, 2.597390946 },
{ 0.0, 0.4912149447, 0.0713489084, 1.0, 0.426487798 },
{ 0.0, 0.3496920248, 1.0135462089, 1.0, 2.962295362 },
{ 1.0, 0.7716284667, 0.5387295927, 4.0, 0.736709363 },
{ 1.0, 0.3463061263, 0.7819578522, 4.0, 1.597238498 },
{ 1.0, 0.6897138762, 1.2793166582, 4.0, 2.376281484 },
{ 0.0, 0.2818824656, 1.4379718141, 3.0, 2.627468417 },
{ 0.0, 0.5659798421, 1.6243568249, 1.0, 1.624809581 },
{ 0.0, 0.7965560518, 0.3933029529, 2.0, 0.415849269 },
{ 0.0, 0.9156922165, 1.0465683565, 1.0, 2.802914008 },
{ 0.0, 0.8299879942, 1.2237155279, 1.0, 2.611676934 },
{ 0.0, 0.0241912066, 1.9213823564, 1.0, 0.659596571 },
{ 0.0, 0.0948590154, 0.3609640412, 1.0, 1.287687748 },
{ 0.0, 0.230467916, 1.9421709292, 3.0, 2.290064565 },
{ 0.0, 0.2209760561, 0.4812708795, 1.0, 1.862393057 },
{ 0.0, 0.4704530933, 0.2644400774, 1.0, 1.960189529 },
{ 1.0, 0.1986645423, 0.48924731, 2.0, 0.333790415 },
{ 0.0, 0.9201823308, 1.4247304946, 1.0, 0.367654009 },
{ 1.0, 0.8118424334, 0.1017034058, 2.0, 2.001390385 },
{ 1.0, 0.1347265388, 0.1362061207, 3.0, 1.151431168 },
{ 0.0, 0.9884603191, 1.5700038988, 2.0, 0.717332943 },
{ 0.0, 0.1964012324, 0.4306495111, 1.0, 1.689056823 },
{ 1.0, 0.4031848807, 1.1251849262, 4.0, 1.977734922 },
{ 1.0, 0.0341882701, 0.3717348906, 4.0, 1.830587439 },
{ 0.0, 0.5073120815, 1.7860476542, 3.0, 0.142862822 },
{ 0.0, 0.6363195451, 0.6631249222, 2.0, 1.211148724 },
{ 1.0, 0.1642774614, 1.1963615627, 3.0, 0.843113448 },
{ 0.0, 0.0945515088, 1.8669327218, 1.0, 2.417198514 },
{ 0.0, 0.2364508687, 1.4035215094, 2.0, 2.964026097 },
{ 1.0, 0.7490112646, 0.1778408242, 4.0, 2.343119453 },
{ 1.0, 0.5193473259, 0.3090019161, 3.0, 1.300277323 }
```

```
};

float test_data[TSTROW][TCOL] = {
    { 0.0, 0.0093314846, 0.0315045565, 1.0, 2.043737003 },
    { 0.0, 0.0663379349, 0.0822378928, 2.0, 1.202557951 },
    { 1.0, 0.9728333529, 0.8778284262, 4.0, 0.205940753 },
    { 1.0, 0.7655418115, 0.3292853828, 4.0, 2.940793653 },
    { 1.0, 0.1610695978, 0.3832762009, 4.0, 1.96753633 },
    { 0.0, 0.0849463812, 1.4988451041, 2.0, 2.307902221 },
    { 0.0, 0.7932621511, 1.2098399368, 1.0, 0.886761862 },
    { 0.0, 0.1336030525, 0.2794256401, 2.0, 2.672175208 },
    { 0.0, 0.4758480834, 0.0441179522, 1.0, 0.399722717 },
    { 1.0, 0.1137434335, 0.922533263, 3.0, 1.927635631 }
};

int i;
int n_classes = 2;
int var_type[] = { 0, 2, 2, 0, 2 };
/* min_n_node, min_split, max_x_cats, max_size, max_depth*/
int tree_control_params[] = { 10, 21, 10, 4, 10 };
int response_col_idx = 0;
float *predicted_values = NULL;
float *fitted_values = NULL;
float *probabilities = NULL;
float *fitted_probabilities = NULL;
float loss_value;
float test_loss_value;

predicted_values = imsls_f_gradient_boosting(TROW, TCOL, &training_data[0][0],
    response_col_idx,
    var_type,
    IMSLS_SHRINKAGE, 0.05,
    IMSLS_RANDOM_SEED, 123457,
    IMSLS_LOSS_VALUE, &loss_value,
    IMSLS_TEST_LOSS_VALUE, &test_loss_value,
    IMSLS_CONTROL, tree_control_params,
    IMSLS_TEST_DATA, TSTROW, &test_data[0][0],
    IMSLS_FITTED_VALUES, &fitted_values,
    IMSLS_PROBABILITIES, &probabilities,
    IMSLS_FITTED_PROBABILITIES, &fitted_probabilities,
    0);

printf("Training data fitted prob[Y=0] and actuals:\n");
for (i = 0; i < TROW; i++){
    printf("\t%3.2f %3.0f\n", fitted_probabilities[i*n_classes],
        training_data[i][response_col_idx]);
}
printf("\nTraining data loss_value=%f\n\n", loss_value);

printf("Test data predicted prob[Y=0] and actuals:\n");
for (i = 0; i < TSTROW; i++){
    printf("\t%3.2f %3.0f\n", probabilities[i*n_classes],
        test_data[i][response_col_idx]);
}
printf("\nTest data loss value=%f\n", test_loss_value);

imsls_free(predicted_values);
imsls_free(fitted_values);
imsls_free(probabilities);
```

```
    imsls_free(fitted_probabilities);  
}
```

Output

Training data fitted prob[Y=0] and actuals:

0.35	0
0.82	0
0.87	0
0.25	1
0.90	0
0.24	1
0.26	1
0.90	0
0.30	1
0.84	0
0.23	1
0.35	0
0.24	1
0.85	0
0.84	0
0.26	1
0.82	0
0.85	0
0.85	0
0.22	1
0.83	0
0.85	0
0.87	0
0.75	1
0.83	0
0.35	1
0.26	1
0.35	0
0.81	0
0.18	1
0.24	1
0.23	1
0.30	1
0.17	1
0.83	0
0.76	1
0.85	0
0.83	0
0.90	0
0.35	1
0.83	0
0.21	1
0.84	0
0.83	0
0.75	1
0.81	0
0.90	0
0.82	0
0.87	0
0.76	0
0.26	1
0.85	0
0.82	0

0.24	1
0.24	1
0.89	0
0.16	1
0.23	1
0.83	0
0.24	1
0.83	0
0.90	0
0.85	0
0.78	0
0.35	1
0.22	1
0.35	1
0.83	0
0.76	0
0.78	0
0.83	0
0.87	0
0.18	1
0.22	1
0.26	1
0.35	0
0.90	0
0.77	0
0.87	0
0.89	0
0.90	0
0.83	0
0.35	0
0.84	0
0.83	0
0.77	1
0.90	0
0.75	1
0.23	1
0.85	0
0.84	0
0.22	1
0.18	1
0.35	0
0.81	0
0.32	1
0.90	0
0.85	0
0.16	1
0.24	1

Training data loss_value=0.650631

Test data predicted prob[Y=0] and actuals:

0.83	0
0.75	0
0.22	1
0.17	1
0.18	1
0.85	0
0.89	0
0.76	0
0.83	0

0.30 1

Test data loss value=0.440048

Example 3

This example uses the same data as in [Example 2](#), but switches the response variable to the 4th column of the training data. Because the response is categorical with more than two categories, the multinomial loss function is used.

Note: The response variable is considered to have five categorical levels because its largest value is 4, but the code assumes categorical variables start in '0'. Since '0' is not present in the data, a warning message is printed.

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

#define TROW 100
#define TCOL 5
#define TSTROW 10

int main(){

    float training_data[TROW][TCOL] = {
        { 0.0, 0.4223019897, 1.7540411302, 3.0, 0.763836258 },
        { 0.0, 0.0907259332, 0.8722643796, 2.0, 1.859006285 },
        { 0.0, 0.1384744535, 0.838324877, 1.0, 0.249729405 },
        { 1.0, 0.5435024537, 1.2359190206, 4.0, 0.831992314 },
        { 0.0, 0.8359154933, 1.8527500411, 1.0, 1.089201049 },
        { 1.0, 0.3577950741, 0.3652825342, 3.0, 2.204364955 },
        { 1.0, 0.6799094002, 0.6610595905, 3.0, 1.44730419 },
        { 0.0, 0.5821297709, 1.6180879478, 1.0, 2.957565282 },
        { 1.0, 0.8229457375, 1.0201675948, 3.0, 2.872570117 },
        { 0.0, 0.0633462721, 0.4140600134, 1.0, 0.63906323 },
        { 1.0, 0.1019134156, 0.0677204356, 3.0, 1.493447564 },
        { 0.0, 0.1551713238, 1.541201456, 3.0, 1.90219884 },
        { 1.0, 0.8273822817, 0.2114979578, 3.0, 2.855730173 },
        { 0.0, 0.7955570114, 1.8757067556, 2.0, 2.930132627 },
        { 0.0, 0.6537275917, 1.2139678737, 2.0, 1.535853243 },
        { 1.0, 0.1243124125, 1.5130919744, 4.0, 2.733670775 },
        { 0.0, 0.2163864174, 0.7051185896, 2.0, 2.755841087 },
        { 0.0, 0.2522670308, 1.2821007571, 2.0, 0.342119491 },
        { 0.0, 0.8677104027, 1.9003869346, 2.0, 2.454376481 },
        { 1.0, 0.8670932774, 0.7993045617, 4.0, 2.732812615 },
        { 0.0, 0.5384287981, 0.1856947718, 1.0, 1.838702635 },
        { 0.0, 0.7236269342, 0.4993310347, 1.0, 1.030699128 },
        { 0.0, 0.0789361731, 1.011216166, 1.0, 2.539607478 },
        { 1.0, 0.7631686032, 0.0536725423, 2.0, 1.401761686 },
        { 0.0, 0.1157020777, 0.0123261618, 1.0, 2.098372295 },
        { 1.0, 0.1451248352, 1.9153951635, 3.0, 0.492650534 },
        { 1.0, 0.8497178114, 1.80941298, 4.0, 2.653985489 },
        { 0.0, 0.8027864883, 1.2631045617, 3.0, 2.716214291 },
        { 0.0, 0.798560373, 0.6872106791, 2.0, 2.763023936 },
        { 1.0, 0.1816879204, 0.4323868025, 4.0, 0.098090197 },
        { 1.0, 0.6301239238, 0.3670980479, 3.0, 0.02313788 },
```

```
{ 1.0, 0.0411311248, 0.0173408454, 3.0, 1.994786958 },
{ 1.0, 0.0427366099, 0.8114635572, 3.0, 2.966069741 },
{ 1.0, 0.4107826762, 0.1929467283, 4.0, 0.573832348 },
{ 0.0, 0.9441903098, 0.0729898885, 1.0, 1.710992303 },
{ 1.0, 0.3597549822, 0.2799857073, 2.0, 0.969428934 },
{ 0.0, 0.3741368004, 1.6052779425, 2.0, 1.866030486 },
{ 0.0, 0.3515911719, 0.3383029872, 1.0, 2.639469598 },
{ 0.0, 0.9184092905, 1.7116801264, 1.0, 1.380178652 },
{ 1.0, 0.77803064, 1.9830028405, 3.0, 1.834021992 },
{ 0.0, 0.573786814, 0.0258851023, 1.0, 1.52130144 },
{ 1.0, 0.3279244492, 0.6977945678, 4.0, 1.322451157 },
{ 0.0, 0.7924819048, 0.3694838509, 1.0, 2.369654865 },
{ 0.0, 0.9787846403, 1.1470323382, 2.0, 0.037156113 },
{ 1.0, 0.6910662795, 0.1019420708, 2.0, 2.58588334 },
{ 0.0, 0.1367050812, 0.6635301332, 2.0, 0.368273583 },
{ 0.0, 0.2826360366, 1.4468787988, 1.0, 2.705811968 },
{ 0.0, 0.4524727969, 0.7885378413, 2.0, 0.851228449 },
{ 0.0, 0.5118664701, 1.061143666, 1.0, 0.249325278 },
{ 0.0, 0.9965170731, 0.2068265025, 2.0, 0.9210639 },
{ 1.0, 0.7801500652, 1.565742691, 4.0, 1.827419217 },
{ 0.0, 0.2906187973, 1.7036567871, 2.0, 2.842997725 },
{ 0.0, 0.1753704017, 0.7124397112, 2.0, 1.262811961 },
{ 1.0, 0.7796778064, 0.3478030777, 3.0, 0.90719801 },
{ 1.0, 0.3889356288, 1.1771452101, 4.0, 1.298438454 },
{ 0.0, 0.9374473374, 1.1879778663, 1.0, 1.854424331 },
{ 1.0, 0.1939157653, 0.093336341, 4.0, 0.166025681 },
{ 1.0, 0.2023756928, 0.0623724433, 3.0, 0.536441906 },
{ 0.0, 0.1691352043, 1.1587338657, 2.0, 2.15494096 },
{ 1.0, 0.0921523357, 0.2247394961, 3.0, 2.006995301 },
{ 0.0, 0.819186907, 0.0392292971, 1.0, 1.282159743 },
{ 0.0, 0.9458126165, 1.5268264762, 1.0, 1.960050194 },
{ 0.0, 0.1373939656, 1.8025095677, 2.0, 0.633624267 },
{ 0.0, 0.0555424779, 0.5022063241, 2.0, 0.639495004 },
{ 1.0, 0.3581428374, 1.4436954968, 3.0, 1.408938169 },
{ 1.0, 0.1189418568, 0.8011626904, 4.0, 0.210266769 },
{ 1.0, 0.5782070206, 1.58215921, 3.0, 2.648622607 },
{ 0.0, 0.460689794, 0.0704823257, 1.0, 1.45671379 },
{ 0.0, 0.6959878858, 0.2245675903, 2.0, 1.849515461 },
{ 0.0, 0.1930288749, 0.6296302159, 2.0, 2.597390946 },
{ 0.0, 0.4912149447, 0.0713489084, 1.0, 0.426487798 },
{ 0.0, 0.3496920248, 1.0135462089, 1.0, 2.962295362 },
{ 1.0, 0.7716284667, 0.5387295927, 4.0, 0.736709363 },
{ 1.0, 0.3463061263, 0.7819578522, 4.0, 1.597238498 },
{ 1.0, 0.6897138762, 1.2793166582, 4.0, 2.376281484 },
{ 0.0, 0.2818824656, 1.4379718141, 3.0, 2.627468417 },
{ 0.0, 0.5659798421, 1.6243568249, 1.0, 1.624809581 },
{ 0.0, 0.7965560518, 0.3933029529, 2.0, 0.415849269 },
{ 0.0, 0.9156922165, 1.0465683565, 1.0, 2.802914008 },
{ 0.0, 0.8299879942, 1.2237155279, 1.0, 2.611676934 },
{ 0.0, 0.0241912066, 1.9213823564, 1.0, 0.659596571 },
{ 0.0, 0.0948590154, 0.3609640412, 1.0, 1.287687748 },
{ 0.0, 0.230467916, 1.9421709292, 3.0, 2.290064565 },
{ 0.0, 0.2209760561, 0.4812708795, 1.0, 1.862393057 },
{ 0.0, 0.4704530933, 0.2644400774, 1.0, 1.960189529 },
{ 1.0, 0.1986645423, 0.48924731, 2.0, 0.333790415 },
{ 0.0, 0.9201823308, 1.4247304946, 1.0, 0.367654009 },
{ 1.0, 0.8118424334, 0.1017034058, 2.0, 2.001390385 },
{ 1.0, 0.1347265388, 0.1362061207, 3.0, 1.151431168 },
{ 0.0, 0.9884603191, 1.5700038988, 2.0, 0.717332943 },
{ 0.0, 0.1964012324, 0.4306495111, 1.0, 1.689056823 },
```

```

    { 1.0, 0.4031848807, 1.1251849262, 4.0, 1.977734922 },
    { 1.0, 0.0341882701, 0.3717348906, 4.0, 1.830587439 },
    { 0.0, 0.5073120815, 1.7860476542, 3.0, 0.142862822 },
    { 0.0, 0.6363195451, 0.6631249222, 2.0, 1.211148724 },
    { 1.0, 0.1642774614, 1.1963615627, 3.0, 0.843113448 },
    { 0.0, 0.0945515088, 1.8669327218, 1.0, 2.417198514 },
    { 0.0, 0.2364508687, 1.4035215094, 2.0, 2.964026097 },
    { 1.0, 0.7490112646, 0.1778408242, 4.0, 2.343119453 },
    { 1.0, 0.5193473259, 0.3090019161, 3.0, 1.300277323 }
};

float test_data[TSTROW][TCOL] = {
    { 0.0, 0.0093314846, 0.0315045565, 1.0, 2.043737003 },
    { 0.0, 0.0663379349, 0.0822378928, 2.0, 1.202557951 },
    { 1.0, 0.9728333529, 0.8778284262, 4.0, 0.205940753 },
    { 1.0, 0.7655418115, 0.3292853828, 4.0, 2.940793653 },
    { 1.0, 0.1610695978, 0.3832762009, 4.0, 1.96753633 },
    { 0.0, 0.0849463812, 1.4988451041, 2.0, 2.307902221 },
    { 0.0, 0.7932621511, 1.2098399368, 1.0, 0.886761862 },
    { 0.0, 0.1336030525, 0.2794256401, 2.0, 2.672175208 },
    { 0.0, 0.4758480834, 0.0441179522, 1.0, 0.399722717 },
    { 1.0, 0.1137434335, 0.922533263, 3.0, 1.927635631 }
};

int i, j;
int n_classes = 5;
int response_col_idx = 3;
int var_type[] = { 0, 2, 2, 0, 2 };
float *predicted_values = NULL;
float *fitted_values = NULL;
float *probabilities = NULL;
float *fitted_probabilities = NULL;
float loss_value;
float test_loss_value;
/* min_n_node, min_split, max_x_cats, max_size, max_depth*/
int tree_control_params[] = { 10, 21, 10, 4, 10 };

predicted_values = imsls_f_gradient_boosting(TROW, TCOL, &training_data[0][0],
    response_col_idx,
    var_type,
    IMSLS_SHRINKAGE, 0.05,
    IMSLS_RANDOM_SEED, 123457,
    IMSLS_LOSS_VALUE, &loss_value,
    IMSLS_CONTROL, tree_control_params,
    IMSLS_TEST_LOSS_VALUE, &test_loss_value,
    IMSLS_TEST_DATA, TSTROW, &test_data[0][0],
    IMSLS_FITTED_VALUES, &fitted_values,
    IMSLS_PROBABILITIES, &probabilities,
    IMSLS_FITTED_PROBABILITIES, &fitted_probabilities,
    0);

printf("Training data fitted probabilities and actuals:\n\n");
printf("Class: ");
for (j = 0; j < n_classes; j++){
    printf("\t %d ", j);
}
printf("\tActual\n");
for (i = 0; i < TROW; i++){

```

```

        for (j = 0; j < n_classes; j++){
            printf("\t%3.2f  ", fitted_probabilities[i*n_classes + j]);
        }
        printf(" %3.0f\n", training_data[i][response_col_idx]);
    }
    printf("\nTraining data loss value=%f\n\n", loss_value);

    printf("Test data predicted probabilities and actuals:\n\n");
    printf("Class: ");
    for (j = 0; j < n_classes; j++){
        printf("\t %d  ", j);
    }
    printf("\tActual\n");
    for (i = 0; i < TSTROW; i++){
        for (j = 0; j < n_classes; j++){
            printf("\t%3.2f  ", probabilities[i*n_classes + j]);
        }
        printf(" %3.0f\n", test_data[i][response_col_idx]);
    }
    printf("\nTest data loss value=%f\n\n", test_loss_value);

    imsls_free(predicted_values);
    imsls_free(fitted_values);
    imsls_free(probabilities);
    imsls_free(fitted_probabilities);

}

```

Output

```

*** WARNING Error IMSLS_EMPTY_CLASS_LEVEL from imsls_f_gradient_boosting.
***          The count of class level 0 in the training data is zero.

```

Training data fitted probabilities and actuals:

Class:	0	1	2	3	4	Actual
	0.02	0.39	0.35	0.17	0.06	3
	0.02	0.45	0.33	0.14	0.06	2
	0.02	0.39	0.40	0.13	0.06	1
	0.02	0.06	0.16	0.36	0.40	4
	0.02	0.44	0.34	0.15	0.05	1
	0.02	0.10	0.17	0.40	0.32	3
	0.02	0.07	0.19	0.32	0.39	3
	0.02	0.45	0.29	0.17	0.06	1
	0.02	0.08	0.19	0.32	0.39	3
	0.02	0.46	0.34	0.12	0.05	1
	0.02	0.09	0.16	0.43	0.29	3
	0.02	0.42	0.30	0.20	0.06	3
	0.02	0.11	0.19	0.37	0.31	3
	0.02	0.43	0.33	0.16	0.05	2
	0.02	0.43	0.36	0.13	0.05	2
	0.02	0.06	0.13	0.46	0.34	4
	0.02	0.44	0.35	0.13	0.06	2
	0.02	0.37	0.39	0.15	0.07	2
	0.02	0.46	0.32	0.15	0.05	2
	0.02	0.08	0.19	0.31	0.39	4
	0.02	0.55	0.28	0.12	0.04	1

0.02	0.46	0.38	0.10	0.05	1
0.02	0.45	0.33	0.14	0.06	1
0.02	0.10	0.20	0.37	0.31	2
0.02	0.52	0.28	0.14	0.04	1
0.02	0.05	0.15	0.43	0.36	3
0.02	0.07	0.16	0.40	0.35	4
0.02	0.44	0.35	0.14	0.05	3
0.02	0.45	0.37	0.11	0.05	2
0.02	0.08	0.18	0.32	0.40	4
0.02	0.09	0.20	0.35	0.34	3
0.02	0.09	0.16	0.43	0.30	3
0.02	0.07	0.16	0.37	0.38	3
0.02	0.08	0.19	0.37	0.34	4
0.02	0.55	0.30	0.10	0.03	1
0.02	0.09	0.19	0.39	0.31	2
0.02	0.44	0.31	0.18	0.06	2
0.02	0.52	0.29	0.13	0.04	1
0.02	0.47	0.32	0.15	0.05	1
0.02	0.07	0.16	0.41	0.35	3
0.02	0.55	0.28	0.12	0.04	1
0.02	0.07	0.17	0.34	0.40	4
0.02	0.53	0.31	0.10	0.04	1
0.02	0.41	0.41	0.10	0.06	2
0.02	0.10	0.19	0.37	0.31	2
0.02	0.39	0.40	0.13	0.06	2
0.02	0.43	0.31	0.18	0.06	1
0.02	0.41	0.39	0.12	0.06	2
0.02	0.41	0.38	0.12	0.06	1
0.02	0.50	0.35	0.10	0.03	2
0.02	0.07	0.16	0.41	0.35	4
0.02	0.43	0.31	0.18	0.06	2
0.02	0.44	0.35	0.13	0.06	2
0.02	0.09	0.22	0.36	0.31	3
0.02	0.07	0.16	0.34	0.40	4
0.02	0.48	0.35	0.11	0.05	1
0.02	0.08	0.19	0.37	0.34	4
0.02	0.08	0.19	0.37	0.34	3
0.02	0.44	0.34	0.14	0.06	2
0.02	0.09	0.16	0.43	0.30	3
0.02	0.54	0.30	0.10	0.03	1
0.02	0.47	0.32	0.14	0.05	1
0.02	0.37	0.36	0.19	0.06	2
0.02	0.41	0.38	0.13	0.05	2
0.02	0.06	0.14	0.43	0.36	3
0.02	0.06	0.18	0.34	0.40	4
0.02	0.06	0.13	0.43	0.36	3
0.02	0.55	0.28	0.12	0.04	1
0.02	0.52	0.32	0.11	0.04	2
0.02	0.44	0.34	0.13	0.06	2
0.02	0.48	0.33	0.12	0.04	1
0.02	0.45	0.34	0.13	0.06	1
0.02	0.07	0.21	0.29	0.41	4
0.02	0.07	0.17	0.34	0.40	4
0.02	0.07	0.17	0.37	0.37	4
0.02	0.42	0.31	0.18	0.06	3
0.02	0.46	0.29	0.17	0.06	1
0.02	0.46	0.38	0.09	0.05	2
0.02	0.48	0.34	0.11	0.05	1
0.02	0.47	0.34	0.12	0.05	1
0.02	0.38	0.35	0.19	0.06	1

0.02	0.52	0.29	0.14	0.04	1	
0.02	0.43	0.31	0.18	0.06	3	
0.02	0.51	0.30	0.12	0.05	1	
0.02	0.55	0.27	0.12	0.04	1	
0.02	0.07	0.19	0.32	0.40	2	
0.02	0.41	0.38	0.14	0.06	1	
0.02	0.11	0.19	0.37	0.31	2	
0.02	0.09	0.17	0.43	0.29	3	
0.02	0.41	0.37	0.14	0.06	2	
0.02	0.52	0.30	0.12	0.05	1	
0.02	0.07	0.16	0.35	0.40	4	
0.02	0.09	0.16	0.40	0.33	4	
0.02	0.40	0.35	0.17	0.06	3	
0.02	0.42	0.39	0.11	0.05	2	
0.02	0.05	0.17	0.40	0.36	3	
0.02	0.42	0.30	0.20	0.06	1	
0.02	0.43	0.33	0.16	0.06	2	
0.02	0.10	0.19	0.37	0.31	4	
0.02	0.10	0.16	0.39	0.32	3	
Training data loss value=0.992967						
Test data predicted probabilities and actuals:						
Class:	0	1	2	3	4	Actual
	0.02	0.52	0.28	0.14	0.04	1
	0.02	0.50	0.31	0.14	0.04	2
	0.02	0.07	0.22	0.28	0.41	4
	0.02	0.10	0.19	0.37	0.32	4
	0.02	0.09	0.16	0.39	0.34	4
	0.02	0.43	0.30	0.19	0.06	2
	0.02	0.39	0.41	0.12	0.05	1
	0.02	0.51	0.29	0.14	0.04	2
	0.02	0.48	0.33	0.12	0.04	1
	0.02	0.07	0.16	0.37	0.38	3
Test data loss value=1.006980						

Warning Errors

IMSLS_NO_PREDICTORS

The model has no predictors.

IMSLS_INVALID_LOSS_FCN

The loss function type # is invalid for a response variable of type #. Resetting to loss function type # = "#".

IMSLS_EMPTY_CLASS_LEVEL

The count of class level # in the training data is zero.

Genetic Algorithms – An Overview

Genetic algorithms are increasingly popular for solving optimization, search and machine learning problems. The analog between optimizing a fitness function and biological processes of natural selection and genetics is generally attributed to John H. Holland and his students at the University of Michigan. His landmark publication “Adaptation in Natural and Artificial Systems” (Holland, 1975) sparked wide ranging investigations into his approach in a variety of areas ranging from science and engineering to business.

This genetic algorithm implementation supports Holland’s basic algorithm with most popular variations. This is achieved by supporting:

1. User defined population size and selection method including roulette, remainder, tournament and stochastic universal sampling both with and without replacement,
2. Random or user defined initial populations,
3. Any combination of four different data types: nominal, binary, integer and real,
4. Base 2 and Gray encoding and decoding of integer and real data types,
5. Automatic encoding and decoding of chromosome information into phenotypes,
6. User specified number of crossover points and three different options for crossover: standard, inversion and partially matched crossover,
7. Elitism to ensure fittest individuals are retained between generations,
8. User supplied fitness functions with or without additional function parameters,
9. User defined crossover and mutation probabilities,
10. Linear and sigma fitness scaling,
11. Customized and predetermined stopping criteria,
12. Measures of algorithm convergence and efficiency – velocity, on-line and off-line fitness.

Data Structures

Alleles

The genetic encoding of a real or artificial organism is contained within their chromosomes. Each chromosome consists of a large number of genes, each uniquely located on the chromosome. Each gene in turn is composed of several alleles. In artificial organisms, i.e. genetic algorithms, an allele is encoded with discrete values.

The original simple genetic algorithm encoded alleles as either zero or one, represented by a single computer bit. This algorithm uses the same encoding for binary, integer and real phenotype values. In addition, users can specify nominal phenotypes which can use any non-negative value. This expands the basic genetic algorithm to include search domains with any number of symbols encoded as nominal phenotypes.

Each nominal phenotype is encoded into a single non-negative integer. Integer phenotypes, on the other hand, are encoded into a binary representation using either Base-2 or Gray encoding.

The crossover operation in `imsls_f_genetic_algorithm` handles a wide variety of allele encoding. Users define their allele encoding using single or multiple bits or a combination. In `imsls_f_genetic_algorithm` nominal, binary, integer and real phenotypes can be defined with any number of crossover points. The crossover and mutation probabilities can be specified. In addition, inversion can be specified for any phenotype and partially matched crossover can be automatically invoked for nominal phenotypes.

This large variety of data types, encoding and crossover options allows users to solve a wide range of search and optimization problems using `imsls_f_genetic_algorithm`.

Chromosomes

In natural systems, chromosomes consist of thousands of genes, each encoded using alleles. In artificial systems, chromosomes are strings of alleles. The relationship between phenotype values and the chromosome allele data structure is created using `imsls_f_ga_chromosome`.

The chromosome data structure for an individual consists of an integer array representing the alleles, and additional information needed for encoding and decoding nominal, integer and real phenotype values into the allele. This information is used for implementing automatic Base-2 and Gray encoding and differentiating between nominal phenotypes requiring partially matched crossover and other classes of nominal phenotypes.

A detailed description of the `imsls_f_chromosome` data structure is given in [Table 46](#). The data structure not only contains the chromosome information encoded as an integer array of alleles, it also contains phenotype values. By default, information in the allele array is automatically decoded into phenotypes. This behavior can be suppressed using the `IMSL_NO_DECODE` option in the `imsls_f_genetic_algorithm` function.

Table 46: The `imsls_f_chromosome` Data Structure

Parameter		Description
<code>total_length</code>	<i>int</i>	Total number of bytes allocated to the data structure.
<code>c_length</code>	<i>int</i>	The length of the <code>allele</code> array.
<code>allele</code>	<i>int[]</i>	An array of of length <code>c_length</code> containing the allele values (bits) for the chromosome.
<code>n_binary</code>	<i>int</i>	The number of binary phenotypes.
<code>n_nominal</code>	<i>int</i>	The number of nominal phenotypes.

Table 46: The `Imsls_f_chromosome` Data Structure

Parameter		Description
<code>n_integer</code>	<i>int</i>	The number of integer phenotypes.
<code>n_real</code>	<i>int</i>	The number of real phenotypes.
<code>n_intBits</code>	<i>int</i>	The total number of bits in <code>allele</code> used to represent the integer phenotypes.
<code>n_realBits</code>	<i>int</i>	The total number of bits in <code>allele</code> used to represent the real phenotypes.
<code>binaryIndex</code>	<i>int</i>	The index of the first bit in <code>allele</code> used to represent the binary phenotypes.
<code>nominalIndex</code>	<i>int</i>	The index of the first bit in <code>allele</code> used to represent the nominal phenotypes.
<code>integerIndex</code>	<i>int</i>	The index of the first bit in <code>allele</code> used to represent the integer phenotypes.
<code>realIndex</code>	<i>int</i>	The index of the first bit in <code>allele</code> used to represent the real phenotypes.
<code>n_categories</code>	<i>int[]</i>	An array of length <code>n_nominal</code> containing the maximum number of categories for each nominal phenotype.
<code>i_intervals</code>	<i>int[]</i>	An array of length <code>n_integer</code> containing the number of discrete intervals used to represent each integer phenotype.
<code>i_bits</code>	<i>int[]</i>	An array of length <code>n_integer</code> containing the number of bits in the allele array assigned to each integer phenotype.
<code>i_bounds</code>	<i>int[]</i>	An array of size <code>n_integer</code> by 2 containing rows of lower and upper limits for each integer phenotype.
<code>r_intervals</code>	<i>int[]</i>	An array of length <code>n_real</code> containing the number of discrete intervals used to represent each real phenotype.
<code>r_bits</code>	<i>int[]</i>	An array of length <code>n_real</code> containing the number of bits in the allele array assigned to each real phenotype.
<code>r_bounds</code>	<i>float[]</i>	An array of size <code>n_real</code> by 2 containing rows of lower and upper limits for each real phenotype.

Individuals

An individual consists of an expressed chromosome for the individual. By default the data structure for individuals also contains decoded values for all phenotypes. This allows users to program their fitness function to use phenotype values instead of their encoded allele representation.

A phenotype is the expression of a collection of genes. In organisms, this expression includes physical characteristics, such as eye color, and behavior. In artificial systems, a phenotype is generally thought of as an attribute. For function optimization problems phenotypes might be floating points or integer values. Phenotypes in a search problem might include nominal or binary encoded information about the search space.

Phenotypes are encoded into the chromosome allele as groups of bits. Later, when the fitness function is evaluated, the algorithm decodes the bits in these groups into their phenotype values. By default this is Base-2 encoding, but Gray encoding can be declared in the `imsls_f_ga_individual`, `imsls_f_ga_random_population` and `imsls_f_genetic_algorithm` functions. Support is provided for mapping integer and real values into allele encoding using discretization and either Base-2 or Gray encoding.

Traditional Base-2 encoding of integer and floating point phenotypes can produce binary representations with widely different representations for phenotypes with similar values. Adjacent integral values encoded using Gray's mapping differ by only one bit. For example, in binary, the numbers 15 and 16 have very different representations: 15="01111" and 16="10000". The Gray encoded values for this number are closer, differing by only a single bit: 15="01000" and 16="11000".

Although the majority of applications discretize integer and real phenotypes and then encode them using either Base-2 or Gray encoding, other encoding methods can be implemented by incorporating phenotype encoding and decoding into the fitness function.

Decoding of chromosome information into its associated phenotypes can be suppressed using the `IMSLS_NO_DECODING` argument in `imsls_f_genetic_algorithm`. In that case the phenotype values in the `Imsls_f_individual` data structure will not be updated with every crossover. They are only decoded for the final generation. Encoding can be either Base-2 or Gray. Base-2 is the default, but Gray encoding can be invoked using the `IMSLS_GRAY` argument in `imsls_f_ga_individual`, `imsls_f_ga_random_population` or `imsls_f_genetic_algorithm`.

Table 47 describes the contents of the `Imsls_f_individual` data structure.

Table 47: The `Imsls_f_individual` Data Structure

Parameter	Data Type	Description
<code>encoding</code>	<code>int</code>	Controls encoding of real and integer phenotypes. Encoding is either Base-2, the default, or Gray invoked using the optional <code>IMSLS_GRAY</code> argument with <code>imsls_f_genetics_algorithm</code> .
<code>total_length</code>	<code>int</code>	Total number of bytes allocated to the data structure.
<code>nominalPhenotype</code>	<code>int[]</code>	An array of integers of length <code>chromosome->n_nominal</code> containing the values of the nominal phenotypes.
<code>binaryPhenotype</code>	<code>int[]</code>	An array of integers of length <code>chromosome->n_binary</code> containing the values of the binary phenotypes.
<code>integerPhenotype</code>	<code>int[]</code>	An array of integers of length <code>chromosome->n_integer</code> containing the values of the integer phenotypes.

Table 47: The `Imsls_f_individual` Data Structure

Parameter	Data Type	Description
<code>realPhenotype</code>	<code>float[]</code>	An array of floating point values of length <code>chromosome->n_real</code> containing the values of the real phenotypes.
<code>chromosome</code>	<code>Imsls_f_chromosome*</code>	A pointer to the chromosome data structure for this individual.

Population

A population is a collection of individuals. A genetic algorithm operates on a population, transforming it from one generation to the next using rules including selection, reproduction, crossover and mutation. A population is described by the chromosome and individual data structures and the number of its members.

The initial population can be created randomly using `imsls_f_ga_random_population`, or it can be created from a user specified set of individuals using `imsls_f_ga_population`. Both of these functions return an `Imsls_f_population` data structure, which is required input to `imsls_f_genetic_algorithm`.

Table 48 describes the `Imsls_f_population` data structure.

Table 48: The `Imsls_f_population` data structure

Parameter	Data Type	Description
<code>n</code>	<code>int</code>	The number of individuals in the population.
<code>indexFittest</code>	<code>int</code>	The index in <code>individual</code> of the fittest individual within the population.
<code>indexWeakest</code>	<code>int</code>	The index in <code>individual</code> of the weakest individual within the population.
<code>avgFitness</code>	<code>float</code>	The average fitness for the population.
<code>stdFitness</code>	<code>float</code>	The standard deviation of the fitness for the population.
<code>maxFitness</code>	<code>float</code>	The maximum fitness of the population.
<code>minFitness</code>	<code>float</code>	The minimum fitness of the population.
<code>fitness</code>	<code>float*</code>	An array of the fitness values for each individual in the population.
<code>chromosome</code>	<code>Imsls_f_chromosome*</code>	A pointer to the chromosome data structure for the individuals in the population.
<code>individual</code>	<code>Imsls_f_individual**</code>	A pointer to an array of size <code>n</code> containing the individuals of the population.

Note that the fitness values in this data structure are only initialized if the fitness function is passed to the [imsls_f_ga_population](#) or [imsls_f_ga_random_population](#). Upon completion, [imsls_f_genetic_algorithm](#) updates these parameters to the values associated with the last generation.

Fitness and Penalty Functions

The genetic algorithm is designed to find the phenotype that maximizes the fitness function. This is a user supplied function that describes the fitness of a particular phenotype. With each succeeding generation, the genetic algorithm transforms a population into better performing individuals as defined by the fitness function.

The fitness function is a required argument to [imsls_f_genetic_algorithm](#). Phenotype restrictions other than simple lower and upper value boundaries are handled by incorporating a penalty function into the fitness calculation.

The optional argument `IMSLS_FITNESS_FCN_WITH_PARMS` allows users to have the algorithm pass individuals and parameters to the fitness function. This provides the flexibility to program a single fitness function that can be applied to a wider variety of applications.

The Genetic Algorithm

There are many variations of the original simple genetic algorithm described by Holland (1975). Many of these were developed for particular applications or data types. [imsls_f_genetic_algorithm](#) implements both the simple algorithm as well as more advanced variations. It has also been designed to provide advanced users the flexibility to provide their own initial populations, stopping criteria, and phenotype encoding and decoding.

Once an initial population is constructed, the genetic algorithm finds a solution to the search or optimization problem using five basic operations to evolve the population from one generation to the next: selection, reproduction, crossover, mutation and fitness.

Selection

Selection is the process used to select individuals for reproduction to create the next generation. This is driven by a fitness function that makes higher fitness individuals more likely to be selected for creating the next generation.

Optimum selection of individuals for reproduction is important to the efficiency and convergence of a genetics algorithm. Many methods have been researched. [imsls_f_genetic_algorithm](#) implements the following variations: deterministic selection, roulette wheel selection with and without replacement, remainder selection with and without replacement, SUS selection, rank selection and two forms of tournament selection. Each of these can be employed with fitness scaling and elitism.

Fitness scaling is not required, but there are two options available: linear scaling and sigma scaling. See [IMSLS_LINEAR_SCALING](#) and [IMSLS_SIGMA_SCALING](#).

Reproduction and Crossover

After individuals are selected, reproduction involves crossing the individual's chromosomes to produce their offspring's chromosome. In the simple case, this involves exchanging genetic information by swapping bits within the parent's chromosome.

Crossover is a random process. It is controlled by the optional arguments `IMSLS_CROSSTERS` and `IMSLS_CROSSOVER_PROB` in `imsls_f_genetic_algorithm`. Not all parents selected for reproduction are mated. Most genetic algorithms use a crossover probability in the range of 0.6 to 0.9. The `IMSLS_CROSSOVER_PROB` argument allows users to select any crossover probability between 0 and 1.

Traditionally chromosomes are crossed at a single point. However, some problems benefit from using more crossover points. The `IMSLS_CROSSTERS` argument allows users to select any number of crossover points.

Once two parents are selected for crossover and their crossover points are defined, a genetic algorithm proceeds to develop a new offspring by alternately mapping alleles from the two chromosomes, swapping the source of the alleles at each crossover point.

For most applications, this creates a new offspring with a non-zero fitness value. However, for some applications, such as the traveling salesman problem, the offspring produced by this simple crossover operation will likely be infeasible. For these problems partially matched crossover and inversion crossover have been developed to ensure that the resulting offspring is a feasible solution.

Partially matched and inversion crossover are invoked using the `IMSLS_PMX_CROSSOVER` and `IMSLS_INVERT_CROSSOVER` optional arguments in `imsls_f_genetic_algorithm`.

Mutation

Mutation stochastically switches allele settings using the mutation probability set with `IMSLS_MUTATION_PROB` in `imsls_f_genetic_algorithm`. Most applications set the mutation probability to a value in the range 0.01 to 0.001. The `IMSLS_MUTATION_PROB` argument accepts any probability between 0 and 1. However, high mutation rates cause the genetic algorithm to perform similar to a random search.

For users who prefer to replace `imsls_f_genetic_algorithm` with their own algorithm, the function `imsls_f_ga_mutate` can be used for mutation. Decoding of the resulting chromosome into phenotype values can be achieved using `imsls_f_ga_decode`.

This traditional mutation operator can produce infeasible solutions for some problems. In those cases, swap mutation is used. That is, instead of inverting a single allele value, two alleles are randomly swapped within the nominal portion of the chromosome. This allows mutation to proceed with search problems such as the traveling salesman problem.

Fitness and Phenotype Constraints

The fitness function is a required argument to `imsls_f_genetic_algorithm`. The genetic algorithm function applies the fitness function to each new individual. It must be scaled to return a non-negative value.

Higher fitness values represent higher performing individuals. Constraints on integer and real phenotypes can be handled by setting lower and upper bounds. Additional constraints for these phenotypes and others should be incorporated using a penalty calculation in the fitness function.

Artificial Populations

A critical step in applying genetic algorithms to a search or optimization problem is creating a population of artificial organisms and their fitness function.

Mapping Phenotypes into Chromosomes

Most applications of genetic algorithms for search and optimization involve binary, nominal, integer or real phenotypes. Most introductions to genetic algorithms describe applications involving the use of simple binary phenotypes, making it easier to focus on the algorithm operations. Binary phenotypes make it possible to implement basic applications and allow users to develop their own phenotype encoding when default encodings are insufficient.

In most applications, integer and real phenotypes are encoded into chromosome bits by mapping their values into a discrete representation. Users specify upper and lower bounds for these phenotypes as well as the number of discrete intervals used for their encoding.

Nominal phenotypes are treated differently from integer phenotypes. Integer phenotypes use chromosome bits as alleles. Nominal phenotypes use groups of bits as alleles. This allows symbolic chromosome representations other than binary. Search problems such as the traveling salesman problem are best represented using nominal phenotypes with partially mixed crossover rather than binary or integer phenotypes.

Information about the nature of the phenotypes and their chromosome encoding is encapsulated in the `imsls_f_chromosome` data structure created by `imsls_f_ga_chromosome`.

Describing Individuals and the Population

An individual is described by their expressed chromosome, phenotypes and parentage information. Chromosome information is encapsulated into an `imsls_f_chromosome` data structure. Individuals are represented by the `imsls_f_individual` data structure, which can be automatically created using `imsls_f_ga_random_population` or systematically specified using `imsls_f_ga_individual` and `imsls_f_ga_population`.

Typically users create an initial population of 20 to 100 individuals, depending on the length of the chromosome.

Selection

Genetic algorithms support a large variety of methods for selecting population individuals for reproduction. The initial population is either randomly selected or systematically specified using `imsls_f_ga_random_population` or `imsls_f_ga_population` with `imsls_f_ga_individual`, respectively.

Selection between generations can be done using a variety of approaches based upon individual fitness. The most common approach is stochastic selection with replacement based upon the individual's fitness. Holland (1975) also referred to this as roulette wheel selection with replacement. Under this approach, individuals with higher fitness have a higher probability of selection. The roulette wheel selection works well when the distribution of fitness across the population is not dominated by the high fitness of a few individuals.

If the population includes a few high fitness individuals, then stochastic selection without replacement can work better than selection with replacement. When selection without replacement is used, an individual cannot be selected more than once per generation for reproduction. Effectively, this ensures that the individuals in the next generation are not generated from just a few, high fitness parents.

In addition to stochastic selection with and without replacement, `imsls_f_genetic_algorithm` also supports deterministic, remainder, tournament and stochastic universal sampling.

Reproduction and Crossover

Reproduction involves selection and crossover using a selection and crossover model. Standard, partially matched and inversion crossover can be selected.

Mutation

Mutation is the stochastic process applied to chromosome bits after crossover. Standard mutation examines each bit and determines whether it should be changed. The probability that a bit is changed is controlled by the mutation probability set using the optional argument `IMSLS_MUTATION_PROB` with `imsls_f_genetic_algorithm`.

When partially matched crossover (PMX) is used with nominal phenotypes, the standard mutation algorithm can result in infeasible offspring. When PMX is employed the mutation algorithm is automatically changed. Instead of switching individual bits, two are randomly selected and swapped. The probability they are swapped is controlled by the mutation probability.

Since the mutation probability is generally in the range 0.001 to 0.01, mutation occurs infrequently. Still it plays a key role in halting premature convergence due to early domination by a few fit individuals resulting in a loss of diversity.

ga_chromosome

Creates an *imsls_f_chromosome* data structure containing unencoded and encoded phenotype information.

Synopsis

```
#include <imsls.h>

imsls_f_chromosome *imsls_f_ga_chromosome (... , 0)
```

The type *double* function is `imsls_d_ga_chromosome`.

Return Value

The function `imsls_f_ga_chromosome` returns an *imsls_f_chromosome* data structure, which is required input to `imsls_f_ga_individual`, `imsls_f_ga_population`, and `imsls_f_ga_random_population`. The memory allocated to this data structure can be released using [imsls_free](#).

Synopsis with Optional Arguments

```
#include <imsls.h>

imsls_f_chromosome *imsls_f_ga_chromosome (
    IMSLS_PRINT,
    IMSLS_BINARY, int n_binary,
    IMSLS_NOMINAL, int n_nominal, int n_categories[],
    IMSLS_INTEGER, int n_integer, int i_intervals[], int i_bounds[],
    IMSLS_REAL, int n_real, int r_intervals[], float r_bounds[],
    0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

By default, summary results are not printed. This option causes the function to print summary results describing the chromosome structure.

IMSL5_BINARY, *int* n_binary (Input)

The number of binary phenotypes.

Default: n_binary = 0.

IMSL5_NOMINAL, *int* n_nominal, *int* n_categories[] (Input)

The first parameter n_nominal is equal to the number of nominal phenotypes. n_categories is an array of length n_nominal containing the number of nominal categories for each nominal phenotype. Each value of this array must be at least 2 or greater. If partially matched crossover is used during the genetic algorithm search then the array n_categories is ignored since all of its values are assumed equal to n_nominal.

Default: n_nominal = 0.

IMSL5_INTEGER, *int* n_integer, *int* i_intervals[], *int* i_bounds[] (Input)

The first parameter n_integer is equal to the number of integer phenotypes. The second parameter in this argument, i_intervals, is a one-dimensional array of length n_integer containing the number of discrete intervals used to map each integer into the chromosome loci. For efficiency, each value in this array should be a power of 2 such as 2, 4, 8, 16, etc. The third parameter, i_bounds is an array of size n_integer by 2 containing the lower and upper bounds for each integer phenotype. The lower and upper bounds for the *i*-th integer phenotype are equal to i_bounds[2*i] and i_bounds[2 * i+1] respectively. Each integer value submitted to [imsls_f_genetic_algorithm](#) for the *i*-th integer phenotype, ω must conform to the inequality:

$$i_bounds[2i] \leq w_i \leq i_bounds[2i+1]$$

Default: n_integer = 0.

IMSL5_REAL, *int* n_real, *int* r_intervals[], *float* r_bounds[] (Input)

The first parameter n_real is equal to the number of real phenotypes. Point values are mapped into chromosome loci using discretization. The second parameter in this argument, r_intervals, is a one-dimensional array of length n_real containing the number of discrete intervals used to map each real value into the chromosome loci. For efficiency, each value in this array should be a power of 2 such as 2, 4, 8, 16, etc. The third parameter, r_bounds is an array of size n_real by 2 containing the lower and upper bounds for each integer phenotype. The lower and upper bounds for the *i*-th real phenotype are equal to r_bounds[2*i] and r_bounds[2*i+1] respectively. Hence, r_bounds[2*i+1] must be greater than r_bounds[2*i]. Each real value submitted to [imsls_f_genetic_algorithm](#) for the *i*-th real phenotype, ω_i must conform to the inequality:

$$r_bounds[2i] \leq w_i < r_bounds[2i+1]$$

Default: n_real = 0.

Description

The genetic algorithm requires a chromosome representation of phenotypes. Most textbook applications of genetic algorithms use phenotypes that have a natural binary encoding. Real world problems often have non-binary phenotypes. Phenotypes are parameters used by the fitness function. Those can include any data type. This function allows for easy encoding of binary, nominal, integer and real phenotypes.

Binary phenotypes are mapped directly into the chromosome as binary bits. Each binary phenotype is treated as a single allele. If the user specifies `n_binary > 0`, then the first `n_binary` bits in the chromosome are allocated for encoding this information. When the fitness function is called during genetic optimization, these bits are translated into zeros and ones and then sent to the fitness function as an integer array of length `n_binary`.

Nominal phenotypes are mapped into the chromosome following the binary phenotypes. The number of bits assigned to each nominal phenotype is determined from the number of categories for each nominal phenotype. The value `n_categories[i]` is equal to the number of categories for the i -th nominal phenotype. The number of bits assigned to this category is the smallest value of k such that $2^k \geq n_categories[i]$, i.e., $k = \lceil \log_2(n_categories[i]) \rceil$, where $\lceil x \rceil$ is the ceiling of x (least integer of x). A binary nominal phenotype would be assigned one bit, and one bit would constitute a single allele. A trinary nominal phenotype would be assigned two bits since $2^2 = 4 \geq 3$, and these bits would be treated as a single allele.

The mapping of multiple bits to a single allele is a key difference between nominal phenotypes and other phenotypes. Alleles for binary, integer and real phenotypes are represented as single bits in the chromosome. The alleles for nominal phenotypes consist of multiple bits. Since crossover occurs between alleles, crossover for nominal phenotypes is treated differently. This ensures that only viable values for nominal phenotypes result from crossover.

It also means that Gray encoding of individual bits has no effect on nominal phenotypes. For many problems Gray encoding is used instead of standard Base-2 encoding to reduce large changes of encoded phenotype values after crossover. As a result, Gray encoding is never applied to nominal phenotypes.

In addition, partially mixed crossover is only an option for nominal phenotypes. Nominal phenotypes combined with partially mixed crossover make it easy to implement search problems similar to the traveling salesman problem.

Both integer and real phenotypes are discretized. Although this is the most common approach to encoding these phenotypes, some problems may benefit from other forms of encoding. If so, users should provide their own encoding, translating the phenotype into a bit representation that can be mapped into binary phenotypes.

Discretization is controlled by two arrays. For integer phenotypes, the array `i_intervals` contains the number of discrete intervals used to represent each integer. The number of chromosome bits assigned to the i -th integer is determined by the values in this array. If `i_intervals[i] = k = 2m` then the i -th integer phenotype is assigned m bits. For example, if `i_intervals[i] = 4`, then this phenotype is assigned two bits.

The array `i_bounds` contains the upper and lower bounds for each integer phenotype. The lower bound for the i -th integer phenotype is equal to $lb = i_bounds[2i]$, and the upper bound is equal to $ub = i_bounds[2i+1]$. The values for the i -th phenotype, w , must satisfy the inequality $lb \leq w < ub$. w is discretized to w' using the formula:

$$w' = \left\lfloor k \frac{(w - lb)}{(ub - lb)} \right\rfloor$$

Where $\lfloor x \rfloor$ is the floor of x (greatest integer of x). This results in mapping the i -th integer phenotype, w , into one of the integers $0, 1, \dots, k-1$.

Real phenotypes are handled in the same fashion as integer phenotypes using the values in `r_intervals` and `r_bounds`.

The number of chromosome bits assigned to each phenotype are described in the following table:

PHENOTYPE	NUMBER of BITS
Binary	$bbits = n_binary$
Nominal	$nbits = \sum_{i=0}^{n_nominal-1} \lceil \log_2(n_categories[i]) \rceil$
Integer	$ibits = \sum_{i=0}^{n_integer-1} \log_2(i_intervals[i])$
Real	$rbits = \sum_{i=0}^{n_real-1} \log_2(r_intervals[i])$

See [Table 46](#) for a description of the allele values (bits) for the chromosome. Chromosome bits are ordered first by binary phenotypes in bits 0 through $bbits - 1$, then nominals, integers and reals in that order.

The memory allocated to this data structure can be released using `imsls_free`.

Example

This example creates a chromosome with 1 binary, 2 nominal, 3 integer and 2 real phenotypes. The `IMSL_PRINT` argument is used to print a description of the chromosome structure.

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

```

int main(){
    int n_binary=1, n_nominal=2, n_integer=3, n_real=2;

    /* number of categories for nominal phenotypes */
    int n_categories[] = {2, 3};

    /* number of intervals and boundaries for integer */
    /* phenotypes */
    int i_intervals[] = {16, 16, 16};
    int i_bounds[]    = {0, 1000, -10, 10, -20, 0};

    /* number of intervals and boundaries for real */
    /* phenotypes */
    int r_intervals[] = {512, 1024};
    float r_bounds[]  = {0.0, 20.0, -20.0, 20.0};

    /* Chromosome Data Types */
    Imsls_f_chromosome* chromosome;

    chromosome = imsls_f_ga_chromosome(
        IMSLS_BINARY, n_binary,
        IMSLS_NOMINAL, n_nominal, n_categories,
        IMSLS_INTEGER, n_integer, i_intervals, i_bounds,
        IMSLS_REAL, n_real, r_intervals, r_bounds,
        IMSLS_PRINT, 0);

    imsls_free(chromosome);
}

```

Output

The `IMSLS_PRINT` option produced the following description of the chromosome. The data structure uses 304 bytes. The chromosome has 34 alleles. The first bit is used to represent the binary phenotype.

The next two alleles are assigned to the nominal phenotypes. The first phenotype will be encoded in allele 1 with the integers zero and one since it has only two categories. The second nominal phenotype has 3 categories. It will be encoded with the integers zero, one, and two.

The integer phenotypes are each assigned 4 binary bits. Since the number of intervals is the same for each integer, 16, 4 bits will be used to encode the integers 0-15. If Base-2 encoding is used, the 16th interval will be encoded as $15 = \{1111\}$.

The first real phenotype uses 512 intervals to discretize its value. This is encoded using 9 alleles. The second real phenotype uses 1024 intervals to discretize its value. This requires 10 alleles to properly represent the values 0 to 1023.

```

*****
**** CHROMOSOME STRUCTURE ****
Data Structure length: 304 Bytes
Chromosome length:      34 Bits

*****ALLELE ASSIGNMENTS*****
Binary:    0 - 0 n_binary = 1

```

```
Nominal: 1 - 2 n_nominal= 2
Integer: 3 - 14 n_integer= 3
Real: 15 - 33 n_real = 2
*****
NOMINAL CATEGORIES*****
Variable 0: 2 categories
Variable 1: 3 categories
*****
INTEGER BOUNDS*****
Variable 0: [ 0, 1000]
Variable 1: [-10, 10]
Variable 2: [-20, 0]
*****
INTEGER BITS*****
Variable 0: 4 bits
Variable 1: 4 bits
Variable 2: 4 bits
*****
INTEGER DISCRETE INTERVALS*****
Variable 0: 16 intervals
Variable 1: 16 intervals
Variable 2: 16 intervals
*****
REAL BOUNDS*****
Variable 0: [0,20]
Variable 1: [-20,20]
*****
REAL BITS*****
Variable 0: 9 bits
Variable 1: 10 bits
*****
REAL DISCRETE INTERVALS*****
Variable 0: 512 intervals
Variable 1: 1024 intervals
*****
```

ga_copy_chromosome

Copies the contents of one chromosome into another chromosome.

Synopsis

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

```
void imsls_f_ga_copy_chromosome (imsls_f_chromosome *chromosomeIn,  
    imsls_f_chromosome *chromosomeOut)
```

The type *double* function is `imsls_d_ga_copy_chromosome`.

Required Arguments

imsls_f_chromosome ***chromosomeIn** (Input)

An existing chromosome to be copied into **chromosomeOut**.

imsls_f_chromosome ***chromosomeOut** (Input/Output)

The contents of **chromosomeOut** are replaced with **chromosomeIn**. The memory for **chromosomeOut** must already be allocated.

Description

This function copies the contents of **chromosomeIn** into **chromosomeOut**. Both chromosomes must have been previously created with identical structures. The memory for both data structures must have already been allocated. If memory is not allocated, use [imsls_f_ga_clone_chromosome](#) to create an entirely new copy of an existing chromosome. Although the structures can have different encoding boundaries, they must have the same number of binary, nominal, integer and real phenotypes.

ga_clone_chromosome

Clones an existing chromosome.

Synopsis

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

```
Imsls_f_chromosome *imsls_f_ga_clone_chromosome (Imsls_f_chromosome *chromosomeIn,  
..., 0)
```

The type *double* function is `imsls_d_ga_clone_chromosome`.

Required Arguments

Imsls_f_chromosome *chromosomeIn (Input)

An existing chromosome to be copied and returned as a new *Imsls_f_chromosome* data structure.

Return Value

The function `imsls_f_ga_clone_chromosome` returns a copy of an *Imsls_f_chromosome* data structure. The memory is allocated for the new data structure and the contents of `chromosomeIn` are copied into that structure. The memory allocated for this data structure can be released using `imsls_free`.

Synopsis with Optional Arguments

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

```
Imsls_f_chromosome *imsls_f_ga_clone_chromosome (Imsls_f_chromosome *chromosomeIn,  
IMSLS_PRINT,  
0)
```

Optional Arguments

`IMSLS_PRINT` (Input)

By default, results are not printed. This option turns on printing of summary information for the cloned chromosome.

Description

Function `ga_clone_chromosome` returns an *imsls_f_chromosome* data structure containing the contents of `chromosomeIn`.

ga_individual

Creates an *ImSls_f_individual* data structure from user supplied phenotypes.

Synopsis

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

```
ImSls_f_individual *imsls_f_ga_individual (ImSls_f_chromosome *chromosome, ..., 0)
```

The type *double* function is `imsls_d_ga_individual`.

Required Arguments

ImSls_f_chromosome *chromosome (Input)

A chromosome data structure created by `imsls_f_ga_chromosome`. This structure is cloned and stored in the *ImSls_f_individual* data structure.

Return Value

The function `imsls_f_ga_individual` returns an *ImSls_f_individual* data structure, which is required input to `imsls_f_ga_population`. The memory allocated to this data structure can be freed using `imsls_f_ga_free_individual`.

Synopsis with Optional Arguments

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

```
ImSls_f_individual *imsls_f_ga_individual (ImSls_f_chromosome *chromosome,
```

```
    IMSLS_PRINT,
```

```
    IMSLS_GRAY_ENCODING,
```

```
    IMSLS_BINARY, int binaryPhenotype[ ],
```

```
    IMSLS_NOMINAL, int nominalPhenotype[ ],
```

```
    IMSLS_INTEGER, int intPhenotype[ ],
```

```

    IMSLS_REAL, float realPhenotype[ ],
    0)

```

Optional Arguments

IMLSL_PRINT, (Input)

By default, intermediate results are not printed. This option turns on printing of intermediate results.

IMLSL_GRAY_ENCODING, (Input)

Specifies whether alleles are encoded using Base-2 or Gray encoding for integer and real phenotypes.

Default: Base-2 encoding.

IMLSL_BINARY, *int* binaryPhenotype[] (Input)

An array of length `chromosome->n_binary` containing the integer values for any binary phenotypes. This is a required argument when `chromosome->n_binary > 0`.

IMLSL_NOMINAL, *int* nominalPhenotype[] (Input)

An array of length `chromosome->n_nominal` containing the integer values for any nominal phenotypes. This is a required argument when `chromosome->n_nominal` is greater than zero. The value of `nominalPhenotype[i]` must be one of the integers 0, 1, ..., `chromosome > n_categories[i]-1`.

IMLSL_INTEGER, *int* integerPhenotype[] (Input)

An array of length `chromosome->n_integer` containing the integer values for any integer phenotypes. This is a required argument when `chromosome->n_integer > 0`. The value of `integerPhenotype[i]` must conform to the inequality:

$$\text{chromosome->i_bounds}[2*i] \leq \text{integerPhenotype}[i] \leq \text{chromosome->i_bounds}[2*i+1]$$

IMLSL_REAL, *float* realPhenotype[] (Input)

An array of length `chromosome->n_real` containing the floating point values for any real phenotypes. This is a required argument when `chromosome->n_real` is greater than zero. The value of `realPhenotype[i]` must conform to the inequality:

$$\text{chromosome->r_bounds}[2*i] \leq \text{realPhenotype}[i] < \text{chromosome->r_bounds}[2*i+1]$$

Description

The `imsls_f_genetic_algorithm` operates on a population of individuals. Individuals can be created automatically using `ga_random_population` or systematically using `imsls_f_ga_population`. If the initial population is created using randomly selected individuals, then this function is not needed. However, if the initial population is to be constructed systematically, then the individuals for that population must first be created using this function.

This function takes the phenotype values in the optional arguments and creates an `Imsls_f_individual` data structure. This structure contains a chromosome created by encoding the phenotypes into their respective allele representations using the chromosome map described in `Imsls_f_chromosome`.

It also allows for incorporating parentage information for the individual, although this is typically not done for the individuals in the initial population.

Memory allocated for this data structure is released using `imsls_f_ga_free_individual`. The chromosome data structure passed to this function is copied into the individual and left unaltered. Hence, releasing memory using `imsls_f_ga_free_individual` does not release memory allocated to the original chromosome. The original chromosome can be released using `imsls_free`.

Example

This example creates an individual using a chromosome with 1 binary, 2 nominals, 3 integers and 2 real phenotypes. The `IMSLS_PRINT` argument is used to print a description of the data structure. By default, Base-2 encoding is used for encoding integer and real phenotypes.

Note that `imsls_f_ga_free_individual` frees the `Imsls_f_chromosome` data structure within the individual.

```
#include <imsls.h>
int main(){
    int n_binary=1, n_nominal=2, n_integer=3, n_real=2;
    /* binary phenotype */
    int binaryPhenotype[] = {1};
    /* number of categories for nominal phenotypes */
    int n_categories[] = {2, 3};
    /* nominal phenotype values */
    int nominalPhenotype[] = {1, 2};
    /* number of intervals and boundaries for integer */
    /* phenotypes */
    int i_intervals[] = {16, 16, 16};
    int i_bounds[] = {0, 1000, -10, 10, -20, 0};
    /* integer phenotype values */
    int integerPhenotype[] = {200, -5, -5};
    /* number of intervals and boundaries for real */
    /* phenotypes */
    int r_intervals[] = {512, 1024};
    float r_bounds[] = {0.0, 20.0, -20.0, 20.0};
    /* real phenotype values */
}
```

```

float realPhenotype[] = {19.9, 19.9};
/* Chromosome Data Structure */
Imsls_f_chromosome* chromosome;
/* Individual Data Structure */
Imsls_f_individual* individual;

chromosome = imsls_f_ga_chromosome(
    IMSLS_BINARY, n_binary,
    IMSLS_NOMINAL, n_nominal, n_categories,
    IMSLS_INTEGER, n_integer, i_intervals, i_bounds,
    IMSLS_REAL, n_real, r_intervals, r_bounds, 0);

/* Create individual data structure */
individual = imsls_f_ga_individual(chromosome,
    IMSLS_BINARY, binaryPhenotype,
    IMSLS_NOMINAL, nominalPhenotype,
    IMSLS_INTEGER, integerPhenotype,
    IMSLS_REAL, realPhenotype,
    IMSLS_PRINT, 0);
imsls_free(chromosome);
imsls_f_ga_free_individual(individual);
}

```

Output

The `IMLSL_PRINT` option produced the following description of the individual. Summary starts with a detailed description of the chromosome. It consists of 34 alleles split among the phenotypes. The actual encoding of the phenotypes into alleles is shown below.

Bits assigned to binary phenotypes are not encoded. They are mapped directly into the first `n_binary` bits of the chromosome. In this case there is only one binary phenotype. It gets mapped into bit zero.

Following the binary phenotype are the nominal phenotypes. Each of these is also mapped into a single allele. However, unlike binary phenotypes, the alleles can assume values other than zero and one.

The integer and real phenotypes are discretized into sixteen interval values. These are then encoded into 4 bit Base-2 representations of the integers 0-15.

```

*****
**** INDIVIDUAL STRUCTURE ****
  Number of Parents: 2
  Encoding: BASE-2
*****
**** CHROMOSOME STRUCTURE ****

Chromosome length:      34 Bits

*****BIT ASSIGNMENTS*****
Binary:    0 -  0 n_binary = 1
Nominal:   1 -  2 n_nominal= 2
Integer:   3 - 14 n_integer= 3
Real:     15 - 33 n_real   = 2
*****

```

```

*****PHENOTYPES*****
BINARY*****
Variable 0: 1
*****
NOMINAL*****
Variable 0: 1
Variable 1: 2
*****
INTEGER*****
Variable 0: 200
Variable 1: -5
Variable 2: -5
*****
REAL*****
Variable 0: 19.9
Variable 1: 19.9
*****

*****CHROMOSOME*****
BINARY BITS: 1

NOMINAL ALLELES: 1 2

INTEGER BITS: 0 0 1 1 0 1 0 0 1 1 0 0

REAL BITS: 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1
*****

```

ga_copy_individual

Copies the contents of one individual into another individual.

Synopsis

```
#include <imsls.h>

void imsls_f_ga_copy_individual(Imsls_f_individual *individualIn,
                               Imsls_f_individual *individualOut)
```

The type *double* function is `imsls_d_ga_copy_individual`.

Required Arguments

Imsls_f_individual *individualIn (Input)
An existing individual to be copied into `individualOut`.

Imsls_f_individual *individualOut (Input/Output)
The contents of `individualOut` are replaced with `individualIn`. `individualOut` must have been previously created with a structure which is identical to `individualIn`.

Description

Function `imsls_f_ga_copy_individual` copies the contents of `individualIn` into `individualOut`. Both individuals must have been created previously with identical structures. Although they can have different encoding boundaries, they must have the same number of binary, nominal, integer and real phenotypes.

ga_clone_individual

Clones an existing individual.

Synopsis

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

```
Imsls_f_individual *imsls_f_ga_clone_individual (Imsls_f_individual *individualIn, ..., 0)
```

The type *double* function is `imsls_d_ga_clone_individual`.

Required Arguments

Imsls_f_individual *individualIn (Input)

An existing individual to be copied.

Return Value

Returns a pointer to an *Imsls_f_individual* data structure containing a copy of `individualIn`. This data structure can be released using the `imsls_f_ga_free_individual` function.

Synopsis with Optional Arguments

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

```
Imsls_f_individual *imsls_f_ga_clone_individual (Imsls_f_individual *individualIn,  
IMSLS_PRINT,  
0)
```

Optional Arguments

IMSLS_PRINT, (Input)

By default, results are not printed. This option turns on printing of summary information for the cloned individual.

Description

Function `imsls_f_ga_clone_individual` returns an *Imsls_f_individual* data structure containing the contents of `individualIn`. Memory is allocated for the returned data structure. It can be released using the `imsls_f_ga_free_individual` function.

ga_mutate

Performs the mutation operation on an individual's chromosome.

Synopsis

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

```
void imsls_f_ga_mutate (float p, imsls_f_individual *individual, ..., 0)
```

The type *double* function is `imsls_d_ga_mutate`.

Required Arguments

float **p** (Input)

The mutation probability. **p** can be any value between 0 and 1. Most applications set the probability to a value in the range 0.01 to 0.001.

imsls_f_individual ***individual** (Input/Output)

An existing individual that will undergo mutation.

Synopsis with Optional Arguments

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

```
void imsls_f_ga_mutate (float p, imsls_f_individual *individual,
```

```
    IMSLS_PRINT,
```

```
    IMSLS_SWAP_MUTATION,
```

```
    0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

By default, results are not printed. This option turns on printing of summary information for the individual showing the chromosome before and after mutation.

IMSLS_SWAP_MUTATION, (Input)

By default, swap encoding of nominal phenotype values is not used. This option turns on swap mutation for nominal phenotypes which ensures that after mutation the new individual contains the same nominal phenotype values with at most two of them having swapped positions. Swap mutation has no effect on the mutation of binary, integer or real phenotypes.

Description

Function `imsls_f_ga_mutate` performs the genetic algorithm mutation operation on the chromosome of an `imsls_f_individual` data structure. Each bit assigned to binary, integer and real phenotypes undergoes the mutation operation using probability p . If the bit is zero it is switched to one and vice versa.

Mutation for nominal phenotypes is handled differently since these are not encoded as binary bits. Each bit assigned to a nominal phenotype can take on a range of values from 0, 1, ... `n_categories[i]-1`. Any mutation must preserve this encoding. If the nominal phenotypes are further encoded using PMX crossover, then mutation must preserve that encoding as well.

For non-PMX encoded nominal phenotypes, each nominal phenotype value is mutated with probability p . If it is selected for mutation, its new value is randomly selected from the uniform discrete distribution of values from 0, 1, ... `n_categories[i]-1`.

If PMX encoding is being used, the optional argument `IMSLS_SWAP_MUTATION` should be employed. This argument invokes swap mutation for the nominal phenotypes. Two nominal phenotypes are randomly selected and then their values are swapped with probability p . This ensures that the resulting new individual has the same nominal values, with at most two of them having switched positions.

ga_decode

Decodes an individual's chromosome into its binary, nominal, integer and real phenotypes.

Synopsis

```
#include <imsls.h>

void imsls_f_ga_decode(Imsls_f_individual *individual)
```

The type *double* function is `imsls_d_ga_decode`.

Required Arguments

Imsls_f_individual *individual (Input/Output)
An existing individual that will have its chromosome information decoded.

Description

Normally decoding is required after crossover or mutation. By default, if `imsls_f_genetic_algorithm` is used, mutation, crossover and decoding is done automatically. If a custom genetic algorithm is being written to replace `imsls_f_genetic_algorithm`, then `imsls_f_ga_decode` can be used within the fitness function prior to fitness calculations.

Binary and nominal phenotypes are copied directly from their chromosome values. Integer and real phenotypes are encoded as binary bits using either Gray or Base-2 encoding. This function decodes those bits into their integer or real representations using the Gray or Base-2 encoding specification contained in the individual's chromosome.

ga_encode

Encodes an individual's binary, nominal, integer and real phenotypes into its chromosome.

Synopsis

```
#include <ims1s.h>

void ims1s_f_ga_encode(Im1s1s_f_individual *individual)
```

The type *double* function is `ims1s_d_ga_encode`.

Required Arguments

*Im1s1s_f_individual *individual* (Input/Output)
An existing individual whose phenotypes get encoded into its chromosome.

Description

Normally encoding is required after changing phenotype values. By default, if either `ims1s_f_ga_population` or `ims1s_f_ga_random_population` are used to build an initial population, phenotype values are automatically encoded into the chromosome. Normally this makes it unnecessary to decode the chromosome within the fitness function. However, if individual phenotype values are changed `ims1s_f_ga_encode` can be used to encode these values into the individual's chromosome.

Binary and nominal phenotypes are copied directly from their phenotype values into the individual's chromosome. Integer and real phenotypes are encoded as binary bits using either Gray or Base-2 encoding. This function encodes those bits into their integer or real representations using the Gray or Base-2 encoding specification in the individual's chromosome.

ga_free_individual

Frees memory allocated to an existing individual.

Synopsis

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

```
void imsls_f_ga_free_individual(Imsls_f_individual *individual)
```

The type *double* function is `imsls_d_ga_free_individual`.

Required Arguments

Imsls_f_individual *individual (Input)
The individual whose memory is released.

Description

Function `imsls_f_ga_free_individual` frees memory allocated to an *Imsls_f_individual* data structure.

Example

See the [Example](#) section of function `imsls_f_ga_individual`.

ga_population

Creates an *Imsls_f_population* data structure from user supplied individuals.

Synopsis

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

```
Imsls_f_population *imsls_f_ga_population, (int n, Imsls_f_chromosome *chromosome,  
      Imsls_f_individual *individual[], ..., 0)
```

The type *double* function is `imsls_d_ga_population`.

Required Arguments

int n (Input)

The number of individuals in the population.

Imsls_f_chromosome *chromosome (Input)

A chromosome data structure created by `imsls_f_ga_chromosome` describing the chromosome encoding for individuals.

Imsls_f_individual *individual[] (Input)

An array of pointers to n individuals.

Return Value

Function `imsls_f_ga_population` returns an *Imsls_f_population* data structure, which is required input to [imsls_f_genetic_algorithm](#). The memory allocated to this data structure can be released using [imsls_f_ga_free_population](#).

Synopsis with Optional Arguments

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

```
Imsls_f_population *imsls_f_ga_population (int n, Imsls_f_chromosome *chromosome,  
      Imsls_f_individual *individual[],
```

```
      IMSLS_PRINT,
```

```
IMSLS_GRAY_ENCODING,  
IMSLS_FITNESS, float fitness[ ],  
IMSLS_FITNESS_FCN, float fitness( ),  
IMSLS_FITNESS_FCN_WITH_PARAMS, float fitness( ), void *parms,  
0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

By default, intermediate results are not printed. This option turns on printing of intermediate results.

`IMSLS_GRAY_ENCODING`, (Input)

Specifies whether alleles are encoded using Base-2 or Gray encoding for integer and real phenotypes.

Default: Base-2 encoding.

`IMSLS_FITNESS`, *float fitness[]* (Input)

An array of length `n` containing the fitness values for the individuals in the population. `fitness[i]` is the fitness for the *i*-th individual.

`IMSLS_FITNESS_FCN`, *float fitness (Imsls_f_individual *individual)* (Input)

User-supplied function to calculate fitness for `individual`. If this is supplied, fitness values are calculated for each individual and included within the expanded population data structure. Otherwise they are set to zero.

`IMSLS_FITNESS_FCN_WITH_PARAMS`, *float fitness (Imsls_f_individual *individual, void *parms), void *parms* (Input)

User-supplied function to calculate fitness for `individual`. If this is supplied, fitness values are calculated for each individual and included in the expanded population data structure. The parameters in `parms` are passed to the function.

Description

The `imsls_f_genetic_algorithm` operates on a population of individuals. `ga_population` allows users to systematically create an initial population by adding individuals to that population. It takes the individuals created using `imsls_f_ga_individual` and encapsulates them into an `Imsls_f_population` data structure.

Example

This example creates a population of 40 individuals each with one binary, two nominal, three integer and two real phenotypes. The `IMSLS_PRINT` argument is used to print a description of the population. A simple fitness function calculation is used to illustrate how fitness values can be used to initialize a population with the `IMSLS_FITNESS` argument. If fitness is not initialized, the fitness array in the data structure is set to `NULL`. It can be initialized using an optional argument with `imsls_f_genetic_algorithm`.

```
#include <imsls.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    int i;
                                /* number of phenotypes by category          */
    int n_binary=1, n_nominal=2, n_integer=3, n_real=2;
    int n = 40;                    /* population size                    */
    int irandom[1];                /* temporary working storage          */
    float rrandom[1];             /* temporary working storage          */
    int binaryPhenotype[] = {1}; /* binary phenotype                   */
                                /* number of categories for nominal phenotypes */
    int n_categories[] = {2, 3};
                                /* nominal phenotype values          */
    int nominalPhenotype[] = {1, 2};
                                /* number of intervals and boundaries for integer */
                                /* phenotypes                          */
    int i_intervals[] = {16, 16, 16};
    int i_bounds[] = {0, 1000, -10, 10, -20, 0};
                                /* integer phenotype values          */
    int integerPhenotype[] = {200, -5, -5};
                                /* number of intervals and boundaries for real */
                                /* phenotypes                          */
    int r_intervals[] = {512, 1024};
    float r_bounds[] = {0.0, 20.0, -20.0, 20.0};
                                /* real phenotype values            */
    float realPhenotype[] = {19.9, 19.9};
                                /* fitness array for individuals     */
    float fitness[40];
                                /* Chromosome Data Structure          */
    Imsls_f_chromosome* chromosome=NULL;
                                /* Individual Data Structure          */
    Imsls_f_individual* individuals[40];
                                /* Population Data Structure          */
    Imsls_f_population* population=NULL;

    chromosome = imsls_f_ga_chromosome(
        IMSLS_BINARY, n_binary,
        IMSLS_NOMINAL, n_nominal, n_categories,
        IMSLS_INTEGER, n_integer, i_intervals, i_bounds,
        IMSLS_REAL, n_real, r_intervals, r_bounds, 0);

    imsls_random_seed_set(12345);
                                /* Create individuals                  */
    printf("Creating %d Individuals\n", n);
    for(i=0; i<n; i++){
                                /* generate random values for phenotypes */
        imsls_f_random_binomial(1, 1, 0.5,
```

```

    IMSLS_RETURN_USER, binaryPhenotype, 0);
    imsls_f_random_uniform_discrete(1, n_categories[0],
    IMSLS_RETURN_USER, irandom, 0);
    nominalPhenotype[0] = irandom[0]-1;
    imsls_f_random_uniform_discrete(1, n_categories[1],
    IMSLS_RETURN_USER, irandom, 0);
    nominalPhenotype[1] = irandom[0]-1;
    imsls_f_random_uniform_discrete(1, i_bounds[1]-i_bounds[0],
    IMSLS_RETURN_USER, irandom, 0);
    integerPhenotype[0] = irandom[0]-1;
    imsls_f_random_uniform_discrete(1, i_bounds[3]-i_bounds[2],
    IMSLS_RETURN_USER, irandom, 0);
    integerPhenotype[1] = irandom[0]-1+i_bounds[2];
    imsls_f_random_uniform_discrete(1, i_bounds[5]-i_bounds[4],
    IMSLS_RETURN_USER, irandom, 0);
    integerPhenotype[2] = irandom[0]-1+i_bounds[4];
    imsls_f_random_uniform(1, IMSLS_RETURN_USER, rrandom, 0);
    realPhenotype[0] =
        rrandom[0] * (r_bounds[1]-r_bounds[0]) + r_bounds[0];
    imsls_f_random_uniform(1, IMSLS_RETURN_USER, rrandom, 0);
    realPhenotype[1] =
        rrandom[0]*(r_bounds[3]-r_bounds[2]) + r_bounds[2];
        /* create individual from these phenotypes */
    individuals[i] = imsls_f_ga_individual(chromosome,
    IMSLS_BINARY, binaryPhenotype,
    IMSLS_NOMINAL, nominalPhenotype,
    IMSLS_INTEGER, integerPhenotype,
    IMSLS_REAL, realPhenotype, 0);
        /* calculate fitness for this individual */
    fitness[i] = 100.0 + 10*binaryPhenotype[0];
    fitness[i] += 2*nominalPhenotype[1] - 4*nominalPhenotype[0];
    fitness[i] += 0.0001*integerPhenotype[0] +
        abs(integerPhenotype[1]+integerPhenotype[2])*0.1;
    fitness[i] += 0.1*realPhenotype[0];
    if (realPhenotype[1]>0) fitness[i] += 0.2*realPhenotype[1];
    else fitness[i] -= 0.2*realPhenotype[1];
}

printf("Creating Population from %d Individuals\n", n);

population = imsls_f_ga_population(n, chromosome, individuals,
    IMSLS_FITNESS, fitness, IMSLS_PRINT, 0);

    imsls_free(chromosome);
    for(i=0; i<n; i++) imsls_f_ga_free_individual(individuals[i]);
    imsls_f_ga_free_population(population);
}

```

Output

The `IMLSL_PRINT` option produced the following description of the population. A summary of the population chromosome structure and fitness are printed followed by detailed information for the first 5 individuals in the population.

This example also illustrates the bit ordering within chromosomes. Nominal phenotypes are placed in the first bits followed by binary and encoded integers and real phenotypes.

```
Creating 40 Individuals
Creating Population from 40 Individuals
Population Size: 40
```

```
Average Fitness: 109.472527
Std. Dev. Fitness: 5.927261
Maximum Fitness: 120.244392
Minimum Fitness: 98.221916
```

```
Chromosome:
```

```
*****
**** CHROMOSOME STRUCTURE ****
```

```
Chromosome length:      34 Bits
```

```
*****BIT ASSIGNMENTS*****
```

```
Binary:   2 - 2 n_binary = 1
Nominal:  1 - 2 n_nominal= 2
Integer:  3 - 14 n_integer= 3
Real:     15 - 33 n_real = 2
*****
```

```
NOMINAL CATEGORIES*****
```

```
Variable 0:  2 categories
Variable 1:  3 categories
*****
```

```
INTEGER BOUNDS*****
```

```
Variable 0: [  0, 1000]
Variable 1: [-10,  10]
Variable 2: [-20,  0]
*****
```

```
INTEGER BITS*****
```

```
Variable 0:  4 bits
Variable 1:  4 bits
Variable 2:  4 bits
*****
```

```
INTEGER DISCRETE INTERVALS*****
```

```
Variable 0: 16 intervals
Variable 1: 16 intervals
Variable 2: 16 intervals
*****
```

```
REAL BOUNDS*****
```

```
Variable 0: [0,20]
Variable 1: [-20,20]
*****
```

```
REAL BITS*****
```

```
Variable 0:  9 bits
Variable 1: 10 bits
*****
```

```
REAL DISCRETE INTERVALS*****
```

```
Variable 0: 512 intervals
Variable 1: 1024 intervals
*****
```

```

First 5 Individuals
***** INDIVIDUAL 0 *****
  Number of Parents: 2
  Encoding: BASE-2
  Fitness: 105.114510

      PHENOTYPES
*****BINARY*****
  Variable 0: 0
*****NOMINAL*****
  Variable 0: 1
  Variable 1: 2
*****INTEGER*****
  Variable 0: 35
  Variable 1: -10
  Variable 2: -19
*****REAL*****
  Variable 0: 15.3157
  Variable 1: 3.39719

*****CHROMOSOME*****
1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 1 1 0
*****

***** INDIVIDUAL 1 *****
  Number of Parents: 2
  Encoding: BASE-2
  Fitness: 111.796173

      PHENOTYPES
*****BINARY*****
  Variable 0: 1
*****NOMINAL*****
  Variable 0: 1
  Variable 1: 0
*****INTEGER*****
  Variable 0: 195
  Variable 1: -5
  Variable 2: -5
*****REAL*****
  Variable 0: 19.6777
  Variable 1: -14.0445

*****CHROMOSOME*****
1 0 1 0 0 1 1 0 1 0 0 1 1 0 0 1 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 0 0
*****

***** INDIVIDUAL 2 *****
  Number of Parents: 2
  Encoding: BASE-2
  Fitness: 104.841797

      PHENOTYPES
*****BINARY*****
  Variable 0: 0
*****NOMINAL*****
  Variable 0: 0
  Variable 1: 0
*****INTEGER*****
  Variable 0: 167

```

```
Variable 1: 7
Variable 2: -16
*****REAL*****
Variable 0: 18.3331
Variable 1: -10.4589

*****CHROMOSOME*****
0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1 1 0 1 0 0
*****

***** INDIVIDUAL 3 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 110.905807

PHENOTYPES
*****BINARY*****
Variable 0: 1
*****NOMINAL*****
Variable 0: 1
Variable 1: 0
*****INTEGER*****
Variable 0: 629
Variable 1: 0
Variable 2: -17
*****REAL*****
Variable 0: 18.213
Variable 1: -6.608

*****CHROMOSOME*****
1 0 1 1 0 1 0 1 0 0 0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 1 0 1 0 1 0 1 1 0
*****

***** INDIVIDUAL 4 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 114.371025

PHENOTYPES
*****BINARY*****
Variable 0: 1
*****NOMINAL*****
Variable 0: 1
Variable 1: 2
*****INTEGER*****
Variable 0: 51
Variable 1: 8
Variable 2: -3
*****REAL*****
Variable 0: 7.13049
Variable 1: -15.7644

*****CHROMOSOME*****
1 2 1 0 0 0 0 1 1 1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0
*****
```

ga_random_population

Creates an *ImSls_f_population* data structure from randomly generated individuals.

Synopsis

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

```
ImSls_f_population *imsls_f_ga_random_population(int n, ImSls_f_chromosome *chromosome,  
..., 0)
```

The type *double* function is `imsls_d_ga_random_population`.

Required Arguments

int **n** (Input)

The number of individuals to be randomly generated for the population.

ImSls_f_chromosome ***chromosome** (Input)

A chromosome data structure created by `imsls_f_ga_chromosome` describing the chromosome encoding for individuals.

Return Value

Function `imsls_f_ga_random_population` returns a pointer to an *ImSls_f_population* data structure, which is required input to `imsls_f_genetic_algorithm`. The memory allocated to this data structure can be released using `imsls_f_ga_free_population`.

Synopsis with Optional Arguments

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

```
ImSls_f_population *imsls_f_ga_random_population(int n, ImSls_f_chromosome *chromosome,  
IMSL_S_PRINT,  
IMSL_S_GRAY_ENCODING,  
IMSL_S_PMX_CROSSOVER,
```

```

IMSL_S_FITNESS_FCN, float fitness(),
IMSL_S_FITNESS_FCN_WITH_PARMS, float fitness(), void *parms,
IMSL_S_BINARY_SELECTION_PROB, float binary_prob[],
IMSL_S_NOMINAL_SELECTION_PROB, float nominal_prob[],
IMSL_S_INTEGER_SELECTION_MODEL, int int_s_model[], float i_parms[],
IMSL_S_REAL_SELECTION_MODEL, int real_s_model[], float r_parms[],
0)

```

Optional Arguments

IMSL_S_PRINT, (Input)

By default, intermediate results are not printed. This option turns on printing of intermediate results.

IMSL_S_GRAY_ENCODING, (Input)

Specifies whether alleles are encoded using Base-2 or Gray encoding for integer and real phenotypes.

Default: Base-2 encoding.

IMSL_S_PMX_CROSSOVER, (Input)

This optional argument applies partially matched crossover to the nominal portion of the chromosome. Although `imsls_f_ga_random_population` does not perform crossover in the population, this option signals that the nominal phenotypes are sequential with values consisting of an arrangement of the integers 0, 1, ..., `n_nominal-1`.

Default: Standard crossover. Each nominal phenotype can independently have values from 0 to `n_nominal-1`.

IMSL_S_FITNESS_FCN, *float fitness(imsls_f_individual *individual)* (Input)

User-supplied function to calculate fitness for `individual`. If this is supplied, fitness values are calculated for each individual and included within the expanded population data structure. Otherwise they are set to zero.

IMSL_S_FITNESS_FCN_WITH_PARMS, *float fitness(imsls_f_individual *individual, void *parms), void *parms* (Input)

User-supplied function to calculate fitness for `individual`. If this is supplied, fitness values are calculated for each individual and included in the expanded population data structure. The parameters in `parms` are passed to the function.

IMSLS_BINARY_SELECTION_PROB, *float* **binary_prob**[], (Input)

The random selection model for randomly generating values for the binary phenotypes. By default binary phenotype values are selected with equal probability, i.e. $p(0) = p(1) = 0.5$.

However, **binary_prob** can be used to specify any Bernoulli distribution as the random selection model for individual binary phenotypes. **binary_prob** is a one-dimensional array of length **n_binary**. **binary_prob**[*i*] is equal to the probability that the *i*-th binary phenotype equals zero. Hence the probability it equals one is $1 - \text{binary_prob}[i]$.

IMSLS_NOMINAL_SELECTION_PROB, *float* **nominal_prob**[] (Input)

The random selection model for randomly generating values for the nominal phenotypes are described by an array of length n_cats , where

$$n_cats = \sum_{i=0}^{n_nominal-1} n_categories[i]$$

By default all integer values between zero and **n_categories**[*i*]-1 are selected with equal probability. However, **nominal_prob** can be used to specify any multinomial distribution as the random selection model for individual nominal phenotypes. **nominal_prob** is a jagged two dimensional array. The values in the *i*-th row of this array contain the probability of selecting 0, 1, ..., **n_categories**[*i*]-1 for the *i*-th nominal attribute. These must be valid probabilities scaled between 0 and 1, and they must sum to 1.0. The number of values in the *i*-th row is equal to **n_categories**[*i*]. See [imsls_f_ga_chromosome](#) for a description of **n_categories**.

IMSLS_INTEGER_SELECTION_MODEL, *int* **int_s_model**[], *float* **i_parms**[] (Input/Output)

The random selection model for randomly generating values for the integer phenotypes. **int_s_model**[*i*] declares the random selection model for the *i*-th integer phenotype. If **int_s_model**[*i*]= 0, all integer values between the upper and lower limits specified in **chromosome**, **i_bounds**[2*i*] and **i_bounds**[2*i*+1], are selected with equal probability for integer *i*, *i* = 0, ..., **n_integer**- 1. This is the default selection method for integer phenotypes. If **int_s_model**[*i*]= 0, the contents of **i_parms**[2*i*] and **i_parms**[2*i*+1] are replaced with the lower limit of the interval and its width, respectively.

If **int_s_model**[*i*]=1, the Poisson random selection model is used. The Poisson distribution models a population of non-negative integers. If this model is selected, then all values for the *i*-th integer phenotype must be non-negative. The user supplied value of **i_parms**[2*i*] is used as the mean for the Poisson distribution and the value of **i_parms**[2*i*+1] is ignored.

`IMSLS_REAL_SELECTION_MODEL`, *int* `real_s_model[]`, *float* `r_parms[]` (Input/Output)

The random selection model for randomly generating values for the real phenotypes.

`real_s_model[i]` can be used to specify the random selection model for the *i*-th real phenotype. If `real_s_model[i] = 0`, all real values between `i_bounds[2i]` and `i_bounds[2i+1]` are selected with equal probability using the uniform distribution. This is the default selection method for real phenotypes.

If `real_s_model[i] = 1`, then the Gaussian distribution is used. In this case, the value of `r_parms[2i]` should be set to the mean of this distribution and `r_parms[2i+1]` should equal its variance.

Description

The `imsls_f_genetic_algorithm` operates on a population of individuals.

`imsls_f_ga_random_population` creates an initial population of `n` randomly selected individuals.

`imsls_f_ga_random_population` takes the chromosome structure described by the `chromosome` argument and randomly generates values for each phenotype. These are then encoded into `imsls_f_individual` data structures and placed into the population.

Binary phenotypes are randomly generated Bernoulli random variables with $p(0) = p(1) = 0.5$.

Values for nominal phenotypes are generated with equal probability. That is the probability of sampling each of the `n_categories[i]` values for the *i*-th nominal phenotype is $1/(\text{chromosome} \rightarrow \text{n_categories}[i])$.

By default, random values for the integer phenotypes are generated using the discrete uniform distribution. All values between `i_bounds[2i]` and `i_bounds[2i+1]` are sampled with equal probability. This default can be changed using the optional argument `IMSLS_INTEGER_SELECTION_MODEL`.

Likewise, random values for real phenotypes are generated using the continuous uniform random distribution. All values between `r_bounds[2i]` and `r_bounds[2i+1]` are sampled with equal probability. This default can be changed using the optional `IMSLS_REAL_SELECTION_MODEL` argument.

Example

This example creates a population of 40 individuals each with 1 binary, 2 nominals, 3 integers and 2 real phenotypes. The `IMSLS_PRINT` argument is used to print a description of the population. A simple fitness function calculation is used to illustrate how fitness values can be used to initialize a population with the `IMSLS_FITNESS` argument. If fitness is not initialized, the fitness array in the data structure is set to `NULL`. It can be initialized using an optional argument with `imsls_f_genetic_algorithm`.

```

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

int main(){
    /* number of phenotypes by category */
    int n_binary=1, n_nominal=2, n_integer=3, n_real=2;
    int n = 40; /* population size */
    /* number of categories for nominal phenotypes */
    int n_categories[] = {2, 3};
    /* number of intervals and boundaries for integer */
    /* phenotypes */
    int i_intervals[] = {16, 16, 16};
    int i_bounds[] = {0, 1000, -10, 10, -20, 0};
    /* number of intervals and boundaries for real */
    /* phenotypes */
    int r_intervals[] = {512, 1024};
    float r_bounds[] = {0.0, 20.0, -20.0, 20.0};
    /* Fitness Function */
    static float fitness(Imsls_f_individual* individual);
    /* Chromosome Data Structure */
    Imsls_f_chromosome* chromosome;
    /* Population Data Structure */
    Imsls_f_population* population;

    /******
    /* In this example the user function is thread */
    /* safe. Let CNL know it is safe, which allows */
    /* genetic algorithm to run in parallel, if that */
    /* capability exists on the user computer. */
    imsls_omp_options(
        IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    chromosome = imsls_f_ga_chromosome(
        IMSLS_BINARY, n_binary,
        IMSLS_NOMINAL, n_nominal, n_categories,
        IMSLS_INTEGER, n_integer, i_intervals, i_bounds,
        IMSLS_REAL, n_real, r_intervals, r_bounds,
        0);

    /* Create individuals */
    imsls_random_seed_set(12345);
    printf("Creating Population with %d Individuals\n", n);
    population = imsls_f_ga_random_population(n, chromosome,
        IMSLS_FITNESS_FCN, fitness,
        IMSLS_PRINT,
        0);

    printf("Releasing Allocated Memory\n");
    imsls_free(chromosome);
    imsls_f_ga_free_population(population);
    return 0;
}

static float fitness(Imsls_f_individual* individual){
    float f;

    /* calculate fitness for this individual */

```

```

    f = 100.0 + 10*individual->binaryPhenotype[0];
    f += 2*individual->nominalPhenotype[1] -
        4*individual->nominalPhenotype[0];
    f += 0.0001*individual->integerPhenotype[0] +
        abs(individual->integerPhenotype[1]+
            individual->integerPhenotype[2])*0.1;
    f += 0.1*individual->realPhenotype[0];

    if(individual->realPhenotype[1]>0)
        f += 0.2*individual->realPhenotype[1];
    else
        f += -0.2*individual->realPhenotype[1];

    return f;
}

```

Output

The `IMSLI_PRINT` option produced the following description of the population. A summary of the population chromosome structure and fitness are printed followed by detailed information for the first 5 individuals in the population.

This example also illustrates the bit ordering within chromosomes. Nominal phenotypes are placed in the first bits followed by binary and encoded integer and real phenotypes. Note that this output is identical to the example for `imsls_f_ga_population` because the fitness function is identical and the random phenotype generation uses the same random seed.

```

Creating Population with 40 Individuals
Population Size: 40

```

```

Average Fitness: 109.400070
Std. Dev. Fitness: 5.923696
Maximum Fitness: 120.044495
Minimum Fitness: 98.022011

```

```

Chromosome:
*****
**** CHROMOSOME STRUCTURE ****

```

```

Chromosome length:      34 Bits

```

```

*****BIT ASSIGNMENTS*****
Binary:   2 - 2 n_binary = 1
Nominal:  1 - 2 n_nominal= 2
Integer:  3 - 14 n_integer= 3
Real:     15 - 33 n_real = 2
*****

```

```

NOMINAL CATEGORIES*****
Variable 0:  2 categories
Variable 1:  3 categories
*****

```

```

INTEGER BOUNDS*****
Variable 0: [  0, 1000]
Variable 1: [-10,  10]

```

```

Variable 2: [ -20,  0]
*****

INTEGER BITS*****
Variable 0:  4 bits
Variable 1:  4 bits
Variable 2:  4 bits
*****

INTEGER DISCRETE INTERVALS*****
Variable 0: 16 intervals
Variable 1: 16 intervals
Variable 2: 16 intervals
*****

REAL BOUNDS*****
Variable 0: [0,20]
Variable 1: [-20,20]
*****

REAL BITS*****
Variable 0:  9 bits
Variable 1: 10 bits
*****

REAL DISCRETE INTERVALS*****
Variable 0: 512 intervals
Variable 1: 1024 intervals
*****

First 5 Individuals
***** INDIVIDUAL 0 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 105.114510

      PHENOTYPES
*****BINARY*****
Variable 0: 0
*****NOMINAL*****
Variable 0: 1
Variable 1: 2
*****INTEGER*****
Variable 0: 35
Variable 1: -10
Variable 2: -19
*****REAL*****
Variable 0: 15.3157
Variable 1: 3.39719

*****CHROMOSOME*****
1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 1 1 0
*****

***** INDIVIDUAL 1 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 111.696175

```

```

          PHENOTYPES
*****BINARY*****
Variable 0: 1
*****NOMINAL*****
Variable 0: 1
Variable 1: 0
*****INTEGER*****
Variable 0: 195
Variable 1: -5
Variable 2: -4
*****REAL*****
Variable 0: 19.6777
Variable 1: -14.0445

*****CHROMOSOME*****
1 0 1 0 0 1 1 0 1 0 0 1 1 0 0 1 1 1 1 0 1 1 1 0 0 1 0 0 1 1 0 0 0
*****

***** INDIVIDUAL 2 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 104.741791

          PHENOTYPES
*****BINARY*****
Variable 0: 0
*****NOMINAL*****
Variable 0: 0
Variable 1: 0
*****INTEGER*****
Variable 0: 167
Variable 1: 8
Variable 2: -16
*****REAL*****
Variable 0: 18.3331
Variable 1: -10.4589

*****CHROMOSOME*****
0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1 1 0 1 0 0
*****

***** INDIVIDUAL 3 *****
Number of Parents: 2
Encoding: BASE-2
Fitness: 110.805908

          PHENOTYPES
*****BINARY*****
Variable 0: 1
*****NOMINAL*****
Variable 0: 1
Variable 1: 0
*****INTEGER*****
Variable 0: 630
Variable 1: 0
Variable 2: -16
*****REAL*****
Variable 0: 18.213
Variable 1: -6.608

```

```
*****CHROMOSOME*****  
1 0 1 1 0 1 0 1 0 0 0 0 1 1 1 1 1 0 1 0 0 1 0 0 1 0 1 0 1 0 1 1 0  
*****
```

```
***** INDIVIDUAL 4 *****  
Number of Parents: 2  
Encoding: BASE-2  
Fitness: 114.571030
```

```
PHENOTYPES  
*****BINARY*****  
Variable 0: 1  
*****NOMINAL*****  
Variable 0: 1  
Variable 1: 2  
*****INTEGER*****  
Variable 0: 51  
Variable 1: 9  
Variable 2: -2  
*****REAL*****  
Variable 0: 7.13049  
Variable 1: -15.7644
```

```
*****CHROMOSOME*****  
1 2 1 0 0 0 1 1 1 1 1 1 0 0 1 0 1 1 0 1 1 0 0 0 0 1 1 0 1 1 0 0  
*****
```

Releasing Allocated Memory

ga_copy_population

Copies the contents of one population into another population.

Synopsis

```
#include <imsls.h>

void imsls_f_ga_copy_population(Imsls_f_population *populationIn,
    Imsls_f_population *populationOut)
```

The type *double* function is `imsls_d_ga_copy_population`.

Required Arguments

Imsls_f_population ***populationIn** (Input)
An existing population to be copied into **populationOut**.

Imsls_f_population ***populationOut** (Input/Output)
The contents of **populationOut** are replaced with **populationIn**.

Description

Function `imsls_f_ga_copy_population` copies the contents of **populationIn** into **populationOut**. Both populations must have been previously created with identical structures. Although they can have different encoding boundaries, they must have the same number of binary, nominal, integer and real phenotypes and the same number of individuals.

ga_clone_population

Clones an existing population.

Synopsis

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

```
Imsls_f_population *imsls_f_ga_clone_population (Imsls_f_population *populationIn, ..., 0)
```

The type *double* function is `imsls_d_ga_clone_population`.

Required Arguments

Imsls_f_population *populationIn (Input)

A pointer to an existing population that gets cloned and returned as a pointer to a copy of `populationIn`.

Return Value

Function `imsls_f_ga_clone_population` creates a copy of an *Imsls_f_population* data structure. The memory is allocated for the new data structure and the contents of `populationIn` are copied into that structure. Memory allocated for this data structure can be released using the `imsls_f_ga_free_population` function.

Synopsis with Optional Arguments

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

```
Imsls_f_population *imsls_f_ga_clone_population (Imsls_f_population *populationIn,  
        IMSLS_PRINT,  
        0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

By default, summary information is not printed. This option turns on printing of summary information for the cloned population.

Description

Function `imsls_f_ga_clone_population` returns a pointer to an *imsls_f_population* data structure containing the contents of `populationIn`.

ga_grow_population

Adds individuals to an existing population.

Synopsis

```
#include <imsls.h>

void imsls_f_ga_grow_population(int n, imsls_f_individual *individual[],
    imsls_f_population *population, ..., 0)
```

The type *double* function is `imsls_d_ga_grow_population`.

Required Arguments

int **n** (Input)
The number of individuals to add to the population.

*imsls_f_individual ****individual**[] (Input)
An array of pointers to **n** individuals.

*imsls_f_population ****population** (Input/Output)
An existing population.

Synopsis with Optional Arguments

```
#include <imsls.h>

imsls_f_population *imsls_f_ga_grow_population(int n, imsls_f_individual *individual[],
    imsls_f_population *population,

    IMSLS_PRINT,

    IMSLS_FITNESS, float fitness[],

    IMSLS_FITNESS_FCN, float fitness(),

    IMSLS_FITNESS_FCN_WITH_PARMS, float fitness(), void *parms,

    0)
```

Optional Arguments

IMSLS_PRINT, (Input)

By default, summary statistics are not printed. This option turns on printing of the summary statistics for the new population.

IMSLS_FITNESS, *float fitness[]* (Input)

An array of length **n** containing the fitness values for the individuals added to the population. **fitness[i]** is the fitness for the *i*-th individual.

IMSLS_FITNESS_FCN, *float fitness(lmsls_f_individual *individual)* (Input)

The fitness function calculated for **individual**. If this is supplied, fitness values are calculated for each individual and included within the expanded population data structure. Otherwise they are set to zero.

IMSLS_FITNESS_FCN_WITH_PARMS, *float fitness(lmsls_f_individual *individual, void *parms), void *parms* (Input)

The fitness function calculated for **individual**. If this is supplied, fitness values are calculated for each individual and included in the expanded population data structure. The parameters in **parms** are passed to the function.

Description

Function **lmsls_f_ga_grow_population** grows an existing population by adding new individuals. The chromosome data structure of the individuals and the population must be identical. Fitness values for the new population are set to zero unless the fitness function is supplied using the optional arguments

IMSLS_FITNESS_FCN or **IMSLS_FITNESS_FCN_WITH_PARMS**.

Fitness values for the new individuals can also be supplied using optional argument **IMSLS_FITNESS**.

ga_merge_population

Creates a new population by merging two populations with identical chromosome structures.

Synopsis

```
#include <imsls.h>

Imsls_f_population *imsls_f_ga_merge_population (Imsls_f_population *population1,
                                                Imsls_f_population *population2, ..., 0)
```

The type *double* function is `imsls_d_ga_merge_population`.

Required Arguments

```
Imsls_f_population *population1 (Input)
    An existing population with chromosome data structure and encoding identical to population2.

Imsls_f_population *population2 (Input)
    An existing population with chromosome data structure and encoding identical to population1.
```

Return Value

Function `imsls_f_ga_merge_population` returns a pointer to a new population consisting of the individuals in `population1` and `population2`. `population1` and `population2` must have the same chromosome data structure and encoding, although they can have different population sizes. Memory allocated to this data structure can be released using `imsls_f_ga_free_population`.

Synopsis with Optional Arguments

```
#include <imsls.h>

Imsls_f_population *imsls_f_ga_merge_population (Imsls_f_population *population1,
                                                Imsls_f_population *population2,
                                                IMSLS_PRINT,
                                                0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

By default, statistics are not printed. This option turns on printing of the summary statistics for the new population.

Description

Function `imsls_f_ga_merge_population` creates a new population by combining the individuals and chromosome information from two populations. The chromosome data structure of the populations must be identical.

ga_free_population

Frees memory allocated to an existing population.

Synopsis

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

```
void imsls_f_ga_free_population(Imsls_f_population *population)
```

The type *double* function is `imsls_d_ga_free_population`.

Required Arguments

Imsls_f_population *population (Input)

The population for which allocated memory is to be released.

Description

Function `imsls_f_ga_free_population` frees memory allocated to an *Imsls_f_population* data structure.

Example

See the [Example](#) section of function `imsls_f_ga_population`.

genetic_algorithm



[more...](#)

Optimizes a user defined fitness function using a tailored genetic algorithm.

Synopsis

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

```
Imsls_f_individual *imsls_f_genetic_algorithm(float fitness(),  
Imsls_f_population *initial_population, ..., 0)
```

The type *double* function is `imsls_d_genetic_algorithm`.

Required Arguments

*float fitness(Imsls_f_individual *individual)* (Input)

The fitness function. Given the data structure for an individual within the population, `fitness` returns the fitness of that individual. The fitness function must return non-negative values.

*Imsls_f_population *initial_population* (Input)

A pointer to the initial population.

Return Value

Function `imsls_f_genetic_algorithm` optimizes a user defined fitness function by evolving an initial population using a tailored genetic algorithm that searches for the fittest individual. It returns a pointer to a clone of the fittest individual in the last generation. Memory allocated to this data structure can be released using [imsls_f_ga_free_individual](#).

Synopsis with Optional Arguments

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

```
lmsls_f_individual *lmsls_f_genetic_algorithm(float fitness(),  
lmsls_f_population *initial_population,  
  
    IMSLS_GRAY_ENCODING,  
  
    IMSLS_NO_ELITISM,  
  
    IMSLS_NO_DECODE,  
  
    IMSLS_PRINT_LEVEL, int level,  
  
    IMSLS_MAX_GENERATIONS, int max_generations,  
  
    IMSLS_MAX_FITNESS, float max_fitness,  
  
    IMSLS_LINEAR_SCALING, float c,  
  
    IMSLS_SIGMA_SCALING,  
  
    IMSLS_GENERATION_GAP, float p_gap,  
  
    IMSLS_MUTATION_PROB, float p_mutation,  
  
    IMSLS_CROSSOVER_PROB, float p_xover,  
  
    IMSLS_CROSSEVERS, int n_xover,  
  
    IMSLS_PMX_CROSSOVER,  
  
    IMSLS_INVERT_CROSSOVER,  
  
    IMSLS_SELECTION_MODEL, int selection_model,  
  
    IMSLS_FITNESS_FCN_WITH_PARAMS, float fitness(), void *parms,  
  
    IMSLS_N_GENERATIONS, int *n_generations,  
  
    IMSLS_ON_LINE_PERFORMANCE, float **on_line_performance,  
  
    IMSLS_OFF_LINE_PERFORMANCE, float **off_line_performance,  
  
    IMSLS_VELOCITY, float **velocity,  
  
    IMSLS_GENERATION_STATS, float **gen_statistics,  
  
    IMSLS_LAST_GENERATION, lmsls_f_population **last_generation,  
  
    0)
```

Optional Arguments

IMSLS_GRAY_ENCODING, (Input)

By default, alleles for integer and real phenotypes are encoded using Base-2 encoding. This argument changes that default to Gray encoding for integer and real phenotypes.

IMSLS_NO_ELITISM, (Input)

By default, elitism is used to preserve the fittest individual from one generation to the next. This argument disables elitism.

IMSLS_NO_DECODE, (Input)

By default, chromosome information is decoded into the individual's phenotypes before every call to the user's fitness function. This argument disables automatic decoding between generations. Decoding is only applied to the last generation, including the fittest individual.

IMSLS_PRINT_LEVEL, *int level* (Input)

By default, no printing of intermediate and final results occur from this function. The **IMSLS_PRINT_LEVEL** argument accepts the following values for *level*:

level	Enumeration	Description
0	IMSLS_NONE	Suppresses printing of any results.
1	IMSLS_FINAL	Prints summary of final results.
2	IMSLS_TRACE_GEN	Prints summary of final results plus generation statistics.
3	IMSLS_TRACE_ALL	Prints summary of final results, generation statistics and individual crossovers.

IMSLS_MAX_GENERATIONS, *int max_generations* (Input)

The maximum number of generations. Optimization is halted when the number of generations exceeds **max_generations**.

Default: **max_generations**=100.

IMSLS_MAX_FITNESS, *float max_fitness* (Input)

The optimization is halted if the maximum fitness is greater than this value.

Default: **max_fitness**=**imsls_f_machine(7)**, i.e., optimization is not halted by large fitness values. Optimization only stops when the number of generations exceeds **max_generations**.

IMSLS_LINEAR_SCALING, *float c* (Input)

Specifies an upper limit for the linear fitness scaling constant. Set **c** = 1 for no scaling. A check is made to ensure that the minimum scaled fitness is non-negative. If it falls below zero, then the scaling constant is automatically reduced to make the minimum scaled fitness equal to zero. For linear scaling the scaling constant is typically between one and two.

Default: `c =1` , no linear fitness scaling.

IMSLS_SIGMA_SCALING, (Input)

By default, sigma scaling is not used for fitness scaling. This argument enables sigma scaling.

IMSLS_GENERATION_GAP, *float* `p_gap` (Input)

The proportion of weakest individuals replaced between generations. If `p_gap=1`, all of the individuals are replaced.

Default: `p_gap=1`.

IMSLS_MUTATION_PROB, *float* `p_mutation` (Input)

The probability of mutation. Although most applications set this to a value between 0.005 and 0.1, any value between 0 and 1 is allowed.

Default: `p_mutation=0.005`.

IMSLS_CROSSOVER_PROB, *float* `p_xover` (Input)

The probability of crossover. `p_xover` can be any value between 0 and 1. Most genetic algorithms use a probability between 0.6 and 0.9.

Default: `p_xover= 0.6`.

IMSLS_CROSSEVERS, *int* `n_xover` (Input)

The number of crossover points. De Jong's (1975) generalized crossover model R6 is implemented. If `n_xover` is odd, then the chromosome is treated as a string with a default crossover at the beginning of the chromosome. If `n_xover` is even, then the chromosome is treated as a ring with no beginning or end, and crossovers are selected using the uniform distribution on a circle. Crossing points occur at the odd crossover points. If the **IMSLS_PMX_CROSSOVER** optional argument is used, there are always two crossover points within the nominal portion of the chromosome. For partially matched crossovers, this argument is only used to define the number of crossovers within the binary, integer and real portion of the chromosome.

Default: `n_xover=1`.

IMSLS_PMX_CROSSOVER, (Input)

By default this optional argument applies partially matched crossover to the nominal portion of the chromosome. Crossovers for other phenotypes are still applied using standard crossover and inversion crossover if requested. With partially matched crossover, the number of crossovers for nominal phenotypes is set to 2, and partially matched crossover is applied only to the nominal phenotype. The number of crossovers for non-nominal phenotypes is still controlled by the value of `n_xover`. However, if this optional argument is used, crossover points are randomly selected separately for nominal and non-nominal alleles.

IMSLS_INVERT_CROSSOVER, (Input)

This option augments standard or partially matched crossover with inversion. Inversion crossover inverts the values of the alleles in every other crossover segment. If this is applied with partially matched crossover, inversion is applied within the matched segment of the alleles for the nominal phenotypes and then within every other segment of any non-nominal phenotype.

IMSLS_SELECTION_MODEL, *int selection_model* (Input)

The model used for selecting individuals for reproduction. Selection models are described in the following table:

selection_model	Description
IMSLS_DETERMINISTIC	Individuals with highest fitness are selected for reproduction using their expected sampling frequency. See Goldberg (1989)
IMSLS_ROULETTE_WITH	Original fitness-proportional selected described by Holland(1975). Sampling is done with replacement.
IMSLS_ROULETTE_WITHOUT	The original fitness-proportional selected except that sampling is done without replacement. This is also referred to as De Jong's (1975) R3 model.
IMSLS_REMAINDER_WITH	Remainder selection with replacement.
IMSLS_REMAINDER_WITH	Remainder selection with replacement.
IMSLS_REMAINDER_WITHOUT	Remainder selection without replacement
IMSLS_SUS_SELECTION	Stochastic Universal Sampling as described by Baker (1987).
IMSLS_RANK_SELECTION	Rank selection. The individuals with the highest fitness are selected once for reproduction.
IMSLS_TOURNAMENT_1	Tournament selection as described by Wetzel (1983). Only the fittest individual in a pair is selected.
IMSLS_TOURNAMENT_2	Tournament selection as described by Goldberg and Deb (1991). The fittest individual in a pair is selected with probability 0.75. Otherwise the weaker individual is selected.

Default: The original selection method described by Holland (1975),
`selection_model=IMSLS_ROULETTE_WITH.`

IMSLS_FITNESS_FCN_WITH_PARMS, *float fitness(Imsls_f_individual *individual, void *parms), void *parms* (Input)

The fitness function calculated for **individual**. If this is supplied, fitness values are calculated for each individual and included in the expanded population data structure. The parameters in **parms** are passed to the function.

IMSLS_N_GENERATIONS, *int *n_generations* (Output)

The number of generations used to find the fittest individual.

IMSLS_ON_LINE_PERFORMANCE, *float **on_line_performance* (Output)

An array of length **max_generations** containing on-line performance statistics for each generation.

IMSLS_OFF_LINE_PERFORMANCE, *float **off_line_performance* (Output)

An array of length **max_generations** containing off-line performance statistics for each generation.

IMSLS_VELOCITY, *float **velocity* (Output)

An array of length **max_generations** containing velocity statistics for each generation. The velocity for the i -th generation is equal to $\frac{1}{2}\ln(f_i \times f_0)$ where f_i is the maximum fitness for the i -th generation.

IMSLS_GENERATION_STATS, *float **gen_statistics* (Output)

An array of size **max_generations** \times 4 containing the maximum fitness, minimum fitness, average fitness and standard deviation of the fitness for each generation. The i -th row of **gen_statistics** contains the statistics for the i -th generation. When **n_generations** < **max_generations**, rows greater than **n_generations** - 1 are filled with NaN values. The four columns contain the following statistics calculated for each generation:

	Description
1	Maximum Fitness
2	Minimum Fitness
3	Fitness Average
4	Fitness Standard Deviation

IMSLS_LAST_GENERATION, *Imsls_f_population **last_generation* (Output)

The last generation produced by the genetic algorithm. Memory allocated to this data structure can be released using **imsls_f_ga_free_population**.

Description

Genetic algorithms search for the optimum individual in a population. This is defined as the individual with the highest fitness. Function `imsls_f_genetic_algorithm` returns the fittest individual in the last generation. Mathematically, this is equivalent to finding the values of the phenotypes that maximize a user provided fitness function. Although there are no requirements that the fitness function be non-negative, in general, convergence to optimum fitness is faster when values of the fitness function are non-negative. Constraints can be applied by incorporating a penalty function within the fitness calculation. Phenotypes can consist of any combination of nominal, binary, integer and real values. Integer and real values must be encoded into a binary representation. This procedure provides for either Base-2 or Gray encoding. However, users can supply other encodings within the fitness function.

The function `imsls_f_genetic_algorithm` uses the population data structure and fitness with simulated genetic processes of reproduction to search for the optimum individual, i.e. settings of phenotype values. Genetic algorithms have been successfully applied to a wide variety of optimization and search problems, see Holland (1975) and Goldberg (1985).

There are many refinements to the basic genetic algorithm originally described by Holland (1975). His basic algorithm begins with an initial population of n individuals, a fitness function, and probabilities for crossover and mutation of `p_xover` and `p_mutation` respectively. The initial population is transformed from one generation to the next using the following steps:

1. Select n individuals from the current population to generate a mating pool.
2. Apply crossover with probability `p_xover` to pairs of the selected individuals within the mating pool to produce two offspring.
3. Apply mutation with probability `p_mutation` to the offspring to generate the next generation.
4. Check stopping criteria. If they are met, stop and report the fittest individual within the last generation.

By default Holland's approach to these steps are used. However, many variations of these can be selected using optional arguments.

The initial population can be generated automatically using `imsls_f_ga_random_population` or it can be created by first creating individuals using `imsls_f_ga_individual` and then a population for those individuals using `imsls_f_ga_population`.

By default Holland's roulette wheel with replacement is used for selecting the mating pool. The optional argument `IMSL5_SELECTION_MODEL` allows users to select alternate selection methods including remainder, tournament and stochastic universal selection. Default crossover and mutation probabilities are `p_xover`= 0.8 and `p_mutation`= 0.005. These defaults can be changed using the optional arguments `IMSL5_CROSSOVER_PROB` and `IMSL5_MUTATION_PROB`.

In the original algorithm only a single crossover point was randomly selected. The optional argument `IMSLS_CROSSEVERS` allows users to designate any number of crossover points.

Standard crossover proceeds by combining the genes from both parents in the order found in those parents. Inversion crossover inverts this order for one of the parents. Inversion crossover is selected using the optional argument `IMSLS_INVERT_CROSSOVER`.

For certain problems, such as the traveling salesman problem, standard crossover can produce infeasible individuals. One approach is to assign zero fitness to those solutions, but this can be very inefficient. Partially matched crossover is an approach that ensures individuals are feasible for a certain class of problems. If the problem is best represented using nominal phenotypes with values

`0, 1, ..., n_nominal-1` where all values must appear once and only once in the chromosome, then partially matched crossover preserves that condition. Partially matched crossover is selected using the optional argument `IMSLS_PMX_CROSSOVER`.

One issue with some applications of genetic algorithms is premature convergence or convergence to false local solutions. This can occur when dominant individuals within early generations take over the population prematurely reducing population diversity. One approach to this problem is fitness scaling. This implementation allows users to use either linear or sigma fitness scaling. By default, no scaling is used. However, the optional arguments `IMSLS_LINEAR_SCALING` and `IMSLS_SIGMA_SCALING` allow users to have fitness values automatically scaled before selection.

The genetic algorithm is stopped when any one of the stopping criteria is met. The algorithm is stopped when the number of generations exceeds `max_generations` or when the maximum fitness exceeds `max_fitness`. By default `max_generations`= 100; this can be changed using `IMSLS_MAX_GENERATIONS`. By default `max_fitness` is `imsls_f_machine(7)`; this can be changed using `IMSLS_MAX_FITNESS`.

On some platforms, `imsls_f_genetic_algorithm` can evaluate the user-supplied function `fitness` in parallel. This is done only if the function `imsls_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

Examples

Example 1

De Jong (1975) examined the performance of a genetic algorithm for finding the maximum of a multivariate function. This is an example of optimizing a variation of De Jong's R2 function:

$$f(x_1, x_2) = 4000 - 100(x_1^2 - x_2)^2 - (1 - x_1)^2$$

where

$$-2.048 \leq x_1 \leq 2.048 \text{ and } -2.048 \leq x_2 \leq 2.048$$

Since there were only two real phenotypes and the function is easily calculated, the phenotypes were encoded using discretization with 65,536 values over the interval [-2.048, 2.048]. By default, encoding and decoding is done within `imsls_f_genetic_algorithm`. This allows the fitness function to calculate individual fitness using the real phenotypes instead of the chromosome. Both the chromosome and its phenotype representation are available within the `Imsls_f_individual` data structure argument.

The default selection algorithm `IMSLS_ROULETTE_WITH` was used, but the number of crossover probability was set to 0.6. The genetic algorithm was more efficient using a lower crossover probability and Gray encoding instead of the defaults 0.7 and Base-2 encoding. Each generation consisted of 40 individuals.

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

static float deJongR2(Imsls_f_individual* individual);

int main(){
    int i, j;                /* index variables */
    int n = 40;              /* population size */
    int n_generations = 0;   /* final number of generations*/
    int n_real = 2;         /* number of real phenotypes */
    int r_intervals[2] = {65536, 65536};
    float r_bounds[4] = {-2.048, 2.048, -2.048, 2.048};
    float* genStats;        /* generation statistics */
    Imsls_f_chromosome* chromosome; /* chromosome data structure */
    Imsls_f_individual* best_individual; /* optimum */
    Imsls_f_population* population; /* population data structure */
    Imsls_f_population* last_generation; /* last generation */
    /******
    /* In this example the user function is thread safe. Let CNL
    /* know it is safe, which allows genetic algorithm to run in
    /* parallel, if that capability exists on the user computer.
    imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
    imsls_random_seed_set(12345);
    chromosome = imsls_f_ga_chromosome(
        IMSLS_REAL, n_real, r_intervals, r_bounds, 0);
    population = imsls_f_ga_random_population(n, chromosome,
        IMSLS_GRAY_ENCODING,
        IMSLS_FITNESS_FCN, deJongR2, 0);

    best_individual = imsls_f_genetic_algorithm(deJongR2, population,
        IMSLS_PRINT_LEVEL, IMSLS_FINAL,
        IMSLS_MAX_FITNESS, 3999.999,
        IMSLS_CROSSOVER_PROB, 0.6,
        IMSLS_GRAY_ENCODING,
        IMSLS_MAX_GENERATIONS, 1000,
        IMSLS_N_GENERATIONS, &n_generations,
        IMSLS_GENERATION_STATS, &genStats,
        IMSLS_LAST_GENERATION, &last_generation, 0);
```

```

printf("\n*****GENERATION STATISTICS*****\n");
printf("Generation Max. Fit. Avg. Fit. Min. Fit. CV\n");
printf("*****\n");
for(i=0; i<=n_generations; i++){
    printf("Gen. %3d: %11.5f %10.2f %10.2f %9.2f \n",
        i, genStats[4*i], genStats[4*i+2], genStats[4*i+1],
        100*genStats[4*i+3]/genStats[4*i+2]);
}
printf("\nLAST GENERATION\n");
printf("*****\n");
printf("\nIndv Fitness          Chromosome          X1    X2\n");
for(i=0; i<last_generation->n; i++){
    printf(" %2d %6.2f ", i, last_generation->fitness[i]);
    for(j=0; j<last_generation->chromosome->c_length; j++)
        printf("%d",
            last_generation->individual[i]->chromosome->allele[j]);
    printf("%7.3f %6.3f\n",
        last_generation->individual[i]->realPhenotype[0],
        last_generation->individual[i]->realPhenotype[1]);
}
printf("\nMaximum:   %6.2f for Individual %d\n",
        last_generation->maxFitness,
        last_generation->indexFittest);
printf("Minimum:   %6.2f for Individual %d\n",
        last_generation->minFitness,
        last_generation->indexWeakest);
printf("Average:   %6.2f\n", last_generation->avgFitness);
printf("Std. Dev:   %6.2f\n\n", last_generation->stdFitness);
printf("*****\n");
}
/*****
/* De Jong's R2 Function          */
*****/
static float deJongR2(Imsls_f_individual* individual)
{
    float f, x1, x2;
    x1 = individual->realPhenotype[0];
    x2 = individual->realPhenotype[1];
    f = 100*(x1*x1-x2)*(x1*x1-x2) + (1.0-x1)*(1.0-x1);
    f = 4000 - f;
    return f;
}

```

Output

In this example, the print level is set to `IMSLS_FINAL` in order to print the optimum solution. The generation statistics are requested using the `IMSLS_GENERATIONS_STATS` option, and the last population is requested using the `IMSLS_LAST_GENERATION` option.

Although the maximum number of generations is set to 100 using the `IMSLS_GENERATIONS` option, the genetic algorithm halted after 26 generations when the maximum population fitness exceeded 3999.999.

OPTIMUM SOLUTION

```

Fitness: 3999.999512
Phenotypes:
Real:    2

```

Function Calculations: 1080
 Population Size: 40
 Number of Generations: 26

Real Phenotype(s):

1.023594 1.047844

Chromosome (Gray Encoded):

1110000000001011010000111000011

*****GENERATION STATISTICS*****

Generation	Max. Fit.	Avg. Fit.	Min. Fit.	CV
Gen. 0:	3996.12915	3244.16	578.91	28.62
Gen. 1:	3996.25269	3725.90	2770.41	8.24
Gen. 2:	3999.22974	3699.14	1917.26	10.28
Gen. 3:	3999.22974	3683.41	2551.70	9.87
Gen. 4:	3999.73779	3778.83	2551.70	8.35
Gen. 5:	3999.73779	3823.50	3187.72	5.22
Gen. 6:	3999.73779	3796.59	3187.72	5.76
Gen. 7:	3999.73779	3850.94	3302.26	4.28
Gen. 8:	3999.73779	3860.17	3358.92	3.93
Gen. 9:	3999.73779	3886.33	3138.96	4.23
Gen. 10:	3999.74683	3896.13	3292.64	3.85
Gen. 11:	3999.74683	3900.24	3638.17	2.93
Gen. 12:	3999.74683	3899.95	3376.35	3.17
Gen. 13:	3999.74683	3900.57	3476.12	3.19
Gen. 14:	3999.74683	3897.88	3408.28	3.36
Gen. 15:	3999.74683	3908.28	2331.26	3.36
Gen. 16:	3999.99585	3897.28	3301.18	3.94
Gen. 17:	3999.99585	3910.99	3236.92	3.31
Gen. 18:	3999.99585	3953.46	3429.17	1.31
Gen. 19:	3999.99585	3944.98	3764.08	1.41
Gen. 20:	3999.99585	3945.62	3751.01	1.41
Gen. 21:	3999.99585	3934.07	3751.10	1.81
Gen. 22:	3999.99585	3947.08	3739.52	1.62
Gen. 23:	3999.99609	3943.99	3652.86	1.95
Gen. 24:	3999.99609	3942.42	3652.86	1.99
Gen. 25:	3999.99927	3970.69	3845.42	0.92
Gen. 26:	3999.99951	3970.61	3845.42	0.90

LAST GENERATION

Indv	Fitness	Chromosome	X1	X2
0	3916.58	11100000000011001100110001010011	1.023	0.134
1	3996.09	0101000000001011100110001010011	-0.512	0.134
2	3965.22	11010000000011101110111110011011	0.511	0.849
3	3993.47	01100101000001011010000001100011	-0.928	1.028
4	3941.02	1101000000001011010000001100011	0.512	1.028
5	3998.11	1111000000001011100110001110011	0.512	0.134
6	3951.04	01100100000101011100110001100011	-0.898	0.132
7	3919.76	0100000000001011110110001110011	-0.000	0.890
8	3965.35	1111000000001011110111110011011	0.512	0.849
9	3913.56	11010000000011101010111101110011	0.511	1.190
10	3948.39	11010000000011101110001110110011	0.511	0.978
11	3996.04	0101000000001011100110001100011	-0.512	0.132

12	3997.44	11100000100111001110111101101001	1.009	0.859
13	3992.45	11110000000001110100000001000011	0.512	-0.008
14	3958.07	01010000000101011110110001110011	-0.510	0.890
15	3951.33	01100100000001011100110001100011	-0.896	0.132
16	3948.39	11010000000011101110001110110011	0.511	0.978
17	4000.00	11100000000001011010000111010001	1.024	1.046
18	3999.96	11100000000001011010000001100011	1.024	1.028
19	3925.31	01010110010001011010000111010001	-0.440	1.046
20	3999.37	11010000000001011101111001101111	0.512	0.325
21	3845.42	01100101010110110101110001100011	-0.921	-0.380
22	3999.64	11010101000011101100111001000011	0.415	0.184
23	3992.37	011000000000011101110111101101001	-1.023	0.859
24	3990.55	01010000000001110100000001110011	-0.512	-0.006
25	3993.26	111100000000011101100000001000011	0.513	0.008
26	3964.15	110100000000011101110111101110011	0.511	0.858
27	3998.36	110100000000011101100110010011011	0.511	0.143
28	3950.63	01100101000011101100111001000011	-0.927	0.184
29	3996.08	01010000000001011100110001110011	-0.512	0.134
30	3997.89	01010110000011101100111001100111	-0.447	0.188
31	3998.11	11110000000001011100110001110011	0.512	0.134
32	3993.47	01100101000001011010000001100011	-0.928	1.028
33	3992.45	11110000000001110100000001000011	0.512	-0.008
34	3916.61	111000000000011011100110001010011	1.023	0.134
35	3929.38	01010110000001011010000001100011	-0.448	1.028
36	3997.89	01010110000011101100111001100011	-0.447	0.188
37	4000.00	11100000000001011010000111010001	1.024	1.046
38	4000.00	11100000000001011010000111000011	1.024	1.048
39	3991.28	01100100000001011010000001110001	-0.896	1.030
Maximum:	4000.00	for Individual 38		
Minimum:	3845.42	for Individual 21		
Average:	3970.61			
Std. Dev:	35.71			

Example 2

The traveling salesman problem creates a problem for traditional crossover. In this problem, the objective is to find the shortest route while traveling to each city once. In this example, there are eight cities, labeled using the letters a-h, with distances ranging from 17 to 113 miles.

Traditional crossover would create unfeasible routes; that is some routes after crossover would not visit every city once. Some would not be visited and others would be visited more than once.

Partially matched crossover (PMX) preserves the feasibility of a route. In the general sense, PMX assumes that the nominal phenotypes consists of a string of numbers from zero to `n_nominal-1`, with each number appearing once and only once in that string. Partially matched crossover uses two crossover points within the nominal portion of the chromosome and swaps the middle segment between the parents. The first and third segments are manipulated to ensure the resulting offspring is feasible.

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

int main(){
    int i, j, k;                /* index variables          */
```

```

int n = 50; /* population size */
int n_generations; /* number of generations */
int n_nominal = 8; /* number of nominal phenotypes*/
int n_categories[8] = {
    8, 8, 8, 8,
    8, 8, 8, 8 /* nominal category limits */
};
float x1; /* temporary storage */
float avg; /* average fitness */
float* genStats; /* generation statistics */
static float pmxFitness(Imsls_f_individual* individual);
Imsls_f_chromosome* chromosome; /* chromosome data structure */
Imsls_f_individual* best_individual; /* optimum */
Imsls_f_population* population; /* population data structure */
Imsls_f_population* last_generation; /* last generation */
char *cities[8] = {"a", "b", "c", "d", /* Cities Label a-h */
    "e", "f", "g", "h"};
/*****/
/* In this example the user function is thread safe. Let CNL */
/* know it is safe, which allows genetic algorithm to run in */
/* parallel, if that capability exists on the user computer. */
imsls_omp_options(
    IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1,
    0);
imsls_random_seed_set(12345);

chromosome = imsls_f_ga_chromosome(
    IMSLS_NOMINAL, n_nominal, n_categories,
    0);

population = imsls_f_ga_random_population(n, chromosome,
    IMSLS_PMX_CROSSOVER,
    IMSLS_FITNESS_FCN, pmxFitness,
    0);

best_individual = imsls_f_genetic_algorithm(pmxFitness, population,
    IMSLS_PRINT_LEVEL, IMSLS_FINAL,
    IMSLS_PMX_CROSSOVER,
    IMSLS_INVERT_CROSSOVER,
    IMSLS_CROSSOVER_PROB, 0.8,
    IMSLS_MAX_GENERATIONS, 10,
    IMSLS_GENERATION_STATS, &genStats,
    IMSLS_N_GENERATIONS, &n_generations,
    IMSLS_LAST_GENERATION, &last_generation,
    0);

printf("GENERATION STATISTICS\n");
printf("Total Number of Generations: %d\n\n", n_generations);
printf("Generation Max. Fit. Min. Fit. Avg. Fit. CV\n");

for(i=0; i<=n_generations; i++){
    printf("Gen. %3d: %8.0f %8.0f %12.2f %8.2f\n", i,
        genStats[4 * i], genStats[4 * i + 1], genStats[4 * i + 2],
        100 * genStats[4 * i + 3] / genStats[4 * i + 2]
    );
}

printf("\n\n          LAST GENERATION\n");
printf("*****\n");
printf("Individual Fitness Phenotype Values \n");

```

```

    avg = last_generation->avgFitness;

    for(i=0; i<last_generation->n; i++){
        x1 = last_generation->fitness[i];

        printf("    %2d    %6.0f    ", i, x1);

        for(j=0; j<last_generation->chromosome->c_length; j++) {
            k = last_generation->individual[i]->nominalPhenotype[j];
            printf("%s ", cities[k]);
        }
        if(x1 == last_generation->maxFitness){
            printf("***\n");
        }else{
            printf("\n");
        }
    }

    printf("Average Fitness: %6.1f\n\n", avg);
    printf("*****\n");
    printf("OPTIMUM SOLUTION:\n\nFitness:%4.0f\n\nChromosome:      ",
           pmxFitness(best_individual));

    for(i=0; i<best_individual->chromosome->c_length; i++)
        printf("%3d", best_individual->chromosome->allele[i]);
    printf("\n\nPhenotype Values: ");

    for(i=0; i<n_nominal; i++) {
        k = best_individual->nominalPhenotype[i];
        printf("->%s", cities[k]);
    }

    printf("\n\n");
    printf("freeing best individual\n");
    imsls_f_ga_free_individual(best_individual);
    printf("freeing last generation\n");
    imsls_f_ga_free_population(last_generation);
    printf("freeing chromosome\n");
    imsls_free(chromosome);
    printf("freeing population\n");
    imsls_f_ga_free_population(population);
}

/*****
/* FITNESS FUNCTION
*****/
static float pmxFitness(Imsls_f_individual* individual)
{
    int i=0, k=0, i1, i2; /* Index variables */
    int n_nominal = 8; /* number of nominal phenotypes */
    float f = 0.0; /* fitness value */
    float distances[64] = {
        /* cities:
        a  b  c  d  e  f  g  h      */
        0, 17, 27, 73, 61, 57, 51, 23, /* a */
        17, 0, 37, 73, 72, 74, 66, 40, /* b */
        27, 37, 0, 48, 35, 49, 65, 50, /* c */
        73, 73, 48, 0, 47, 82, 113, 95, /* d */
        61, 72, 35, 47, 0, 38, 80, 78, /* e */

```

```

57, 74, 49, 82, 38, 0, 48, 65, /* f */
51, 66, 65, 113, 80, 48, 0, 40, /* g */
23, 40, 50, 95, 78, 65, 40, 0 /* h */
};
n_nominal = individual->chromosome->n_nominal;
k = individual->chromosome->nominalIndex+1;
f = 0.0;

for(i=k; i<k+n_nominal-1; i++){
    i1 = individual->nominalPhenotype[i-1];
    i2 = individual->nominalPhenotype[i ];
    f += distances[i1*n_nominal + i2];
}
return 516-f;
}

```

Output

This program produced the following output. Since the print level was set to `IMSLS_FINAL`, the optimum solution was printed. The generation statistics were requested using the `IMSLS_GENERATIONS_STATS` option, and the last population was requested using the `IMSLS_LAST_GENERATION` option.

The maximum number of generations was set to 10. The genetic algorithm found the optimum route after evaluating the fitness of 550 routes in 10 generations. Generation zero is the initial generation provided to the algorithm and is not counted towards the maximum generation count.

OPTIMUM SOLUTION

```

Fitness: 269.000000
Phenotypes:
  Nominal: 8
Function Calculations: 550
Population Size:      50
Number of Generations: 10

```

Nominal Phenotype(s):

```
3 4 2 1 0 7 6 5
```

Chromosome (Base-2 Encoded):

```
3 4 2 1 0 7 6 5
```

GENERATION STATISTICS

```
Total Number of Generations: 10
```

Generation	Max. Fit.	Min. Fit.	Avg. Fit.	CV
Gen. 0:	194	29	113.64	33.55
Gen. 1:	251	24	119.10	35.08
Gen. 2:	251	37	131.16	35.99
Gen. 3:	251	38	128.96	33.68
Gen. 4:	251	29	135.08	33.25
Gen. 5:	255	38	136.18	33.44
Gen. 6:	255	28	142.38	33.23
Gen. 7:	255	56	150.56	29.31
Gen. 8:	269	55	155.06	28.88

Gen. 9: 269 56 148.62 26.68
 Gen. 10: 269 48 146.32 29.59

LAST GENERATION

Individual Fitness Phenotype Values

0	106	a c d g b h e f
1	269	d e c b a h g f ***
2	222	d e c h g f a b
3	154	e a g h b d c f
4	215	e d c a b f g h
5	215	e d c a b f g h
6	158	a h b c d f g e
7	113	c a d b f e h g
8	78	f a d b h c g e
9	125	e d h g b c f a
10	118	c b e g f h a d
11	209	d e c b h g a f
12	161	h b a d c f g e
13	167	g d b a h c e f
14	169	a f h g b c e d
15	137	f g a e d h b c
16	127	h d a b c f e g
17	90	b d h g c e a f
18	216	d h b a c e f g
19	167	g d e c f h a b
20	120	e h f c d a b g
21	86	f a h d b c g e
22	171	a b d f g h c e
23	156	d h g a c e f b
24	146	c e a d b h g f
25	116	f d g e c h a b
26	118	b c a h e f d g
27	95	d g a b f e h c
28	143	d h b a f c e g
29	157	b f c g h a e d
30	175	b e a h g f c d
31	146	c e a d b h g f
32	111	c b e h g f a d
33	140	b e a f g h c d
34	164	b c e d g h a f
35	166	f e b c a h g d
36	199	b a g h f c d e
37	171	e d f a b c g h
38	97	f a c d h b e g
39	222	d e c h g f a b
40	128	d h c g a b e f
41	99	e g d c b a f h
42	112	d h f c b a e g
43	107	h d a b e f c g
44	143	c h a b f d e g
45	144	g a c e d b f h
46	64	a e c g h d f b
47	103	b d f h g c a e
48	129	d a g b f e c h
49	172	f h b g a c d e

Average Fitness: 146.3

OPTIMUM SOLUTION:	
Fitness:	269
Chromosome:	3 4 2 1 0 7 6 5
Phenotype Values:	->d->e->c->b->a->h->g->f
freeing best individual	
freeing last generation	
freeing chromosome	
freeing population	

Example 3

This example uses the N-Queens problem to illustrate the use of a fitness function with parameters in implementing a genetic algorithm. The N-Queens problem is derived from chess. The genetic algorithm provides an efficient search for a valid solution. For this problem the chess board consists of N rows and N columns. The objective is to place N queens on this board with no conflicts. A conflict occurs when one queen can move and capture another. Since queens can move diagonally, vertically and horizontally this problem is challenging when N becomes large.

One solution for N = 4 is displayed in the following table. A valid solution must place every queen in a different row and column. The problem is to ensure the queens are not in conflict because of lying on the same diagonal.

Row/Col	0	1	2	3
0			Q	
1	Q			
2				Q
3		Q		

Similar to the traveling salesman problem, the N-Queens problem can be expressed using N nominal phenotypes with values 0, 1, ..., N-1. The value of the i -th phenotype represents the row number for the queen in the i -th column. This ensures that any arrangement of the phenotype values represents a board with N queens, each placed in a different row and column.

The solution for N queens displayed above can be represented by the phenotype values {1, 3, 0, 2}. The search looks for arrangements that also do not place queens on the same diagonal. Two queens fall on the same diagonal if the absolute value of the difference between their row numbers equals the absolute value of the difference between their column numbers.

This example uses this representation with a fitness function equal to $(N - C)$ where C equals the number of conflict among the queens. With this fitness, a solution to the N-Queens has a fitness of N.

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

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

typedef struct
{
    int n_queens;
} inputArgs;

int main(){
    int i, j, k;          /* index variables          */
    int n = 500;         /* population size        */
    int n_generations;  /* number of generations  */
    int n_queens = 25; /* number of nominal phenotypes*/
    int n_categories[25];
    float maxFit;       /* maximum fitness hurdle */
    float* genStats;   /* generation statistics   */
    static float queensFitness(
        Imsls_f_individual* individual, inputArgs* input);
    inputArgs* parameters;
    Imsls_f_chromosome* chromosome; /* chromosome */
    Imsls_f_individual* best_individual; /* optimum */
    Imsls_f_population* population; /* population */
    /******
    /* In this example the user function is thread */
    /* safe. Let CNL know it is safe, which allows */
    /* genetic algorithm to run in parallel, if that */
    /* capability exists on the user computer. */
    imsls_omp_options(
        IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    imsls_random_seed_set(12345);
    maxFit = n_queens - 0.5;

    for(i=0; i<n_queens; i++)
        n_categories[i] = n_queens;

    chromosome = imsls_f_ga_chromosome(
        IMSLS_NOMINAL, n_queens, n_categories,
        0);

    parameters = (inputArgs*) malloc(sizeof(inputArgs));
    parameters->n_queens = n_queens;

    population = imsls_f_ga_random_population(n, chromosome,
        IMSLS_PMX_CROSSOVER,
        IMSLS_FITNESS_FCN_WITH_PARMS,
        queensFitness, parameters,
        0);

    best_individual = imsls_f_genetic_algorithm(NULL, population,
        IMSLS_FITNESS_FCN_WITH_PARMS, queensFitness, parameters,
        IMSLS_PRINT_LEVEL, IMSLS_FINAL,
        IMSLS_PMX_CROSSOVER,
        IMSLS_LINEAR_SCALING, 2.0,
        IMSLS_CROSSOVER_PROB, 0.7,
        IMSLS_MUTATION_PROB, 0.01,
        IMSLS_MAX_GENERATIONS, 10000,
        IMSLS_MAX_FITNESS, maxFit,
        IMSLS_GENERATION_STATS, &genStats,
```

```

        IMSLS_N_GENERATIONS, &n_generations,
        0);

printf("GENERATION STATISTICS\n");
printf("Total Number of Generations: %d\n\n", n_generations);
printf("Generation Max. Fit. Min. Fit. Avg. Fit. CV\n");
printf("*****\n");

for(i = 0; i <= n_generations; i += 25){
    printf("Gen. %3d: %8.0f %8.0f %12.2f %8.2f\n",
        i, genStats[4 * i], genStats[4 * i + 1], genStats[4 * i + 2],
        100 * genStats[4 * i + 3] / genStats[4 * i + 2]);
}

i = n_generations;
printf("Gen. %3d: %8.0f %8.0f %12.2f %8.2f\n",
    i, genStats[4 * i], genStats[4 * i + 1], genStats[4 * i + 2],
    100 * genStats[4 * i + 3] / genStats[4 * i + 2]);
printf("*****\n");
printf("OPTIMUM SOLUTION:\n\nFitness:%4.0f\n\nChromosome:      ",
    queensFitness(best_individual, parameters));

for(i = 0; i < best_individual->chromosome->c_length; i++)
    printf("%3d", best_individual->chromosome->allele[i]);

if(n_queens<100){
    printf("\n\nBoard Positions: \n\n");

    for(i=0; i<n_queens; i++)
        printf("--");

    printf("-\n");

    for(i=0; i<n_queens; i++) {
        for(j=0; j<n_queens; j++){
            k = best_individual->nominalPhenotype[j];
            if(i==k)
                printf("|Q");
            else
                printf("| ");
        }

        printf("|\n");

        for(k=0; k<n_queens; k++)
            printf("--");

        printf("-\n");
    }
    printf("\n\n");
}

static float queensFitness(Imsls_f_individual* individual, inputArgs* input)
{
    int i=0, j=0, k=0; /* Index variables */
    int n_queens;
    float f = 0.0; /* Fitness value */

    n_queens = input->n_queens;

```

```

    f = 0;
    for(i=0; i<n_queens-1; i++){
        for(j=i+1; j<n_queens; j++){
            k = individual->chromosome->allele[i] -
                individual->chromosome->allele[j];
            k = abs(k);
            if (abs(i - j) == k) f++;
        }
    }
    f = n_queens - f;
    return f;
}

```

Output

This program produced the following solution to the N-Queens problem with N=50 queens. Notice that some of the minimum fitness values are negative. This alters the random selection of the fittest parents, but if these values are few and small, then the effect is not enough to halt the genetic algorithm.

For 50 queens there are over 1064 ways of placing queens ensuring each is in its own row and column. An exhaustive search of these possible solutions to find a arrangement without diagonal conflicts would be time consuming. The genetic algorithm search found a solution in only 568 generations, requiring 284,500 function evaluations.

OPTIMUM SOLUTION

```

Fitness: 25.000000
Phenotypes:
  Nominal: 25
Function Calculations: 284500
Population Size:      500
Number of Generations: 568
Nominal Phenotype(s):

```

```

23 12 16  9 13  2 18  1 21 10  6 19
 3 20  0  7 15  4  8 17 22 14  5 11

```

```
24
```

```
Chromosome (Base-2 Encoded):
```

```

23 12 16 9 13 2 18 1 21 10 6 19 3 20 0 7 15 4
 8 17 22 14 5 11 24

```

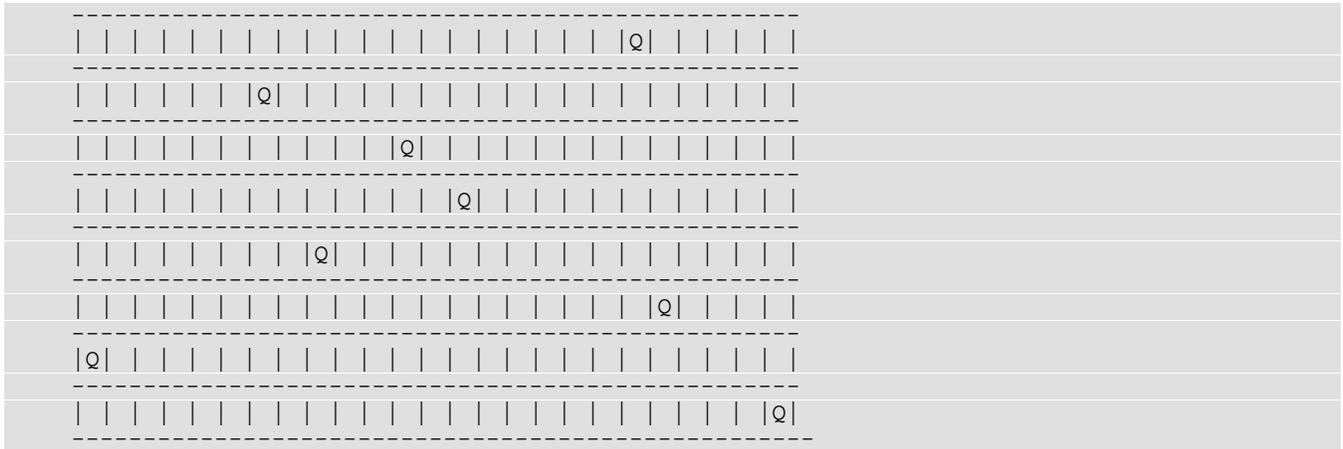
GENERATION STATISTICS

```
Total Number of Generations: 568
```

```

Generation Max. Fit. Min. Fit. Avg. Fit.  CV
*****
Gen.  0:      19     -10      8.11  52.98
Gen. 25:      21      -2     11.02  30.33
Gen. 50:      21      -4     11.49  29.48
Gen. 75:      21      -1     11.15  29.51
Gen. 100:     21      -3     11.42  33.10
Gen. 125:     21      -3     11.17  31.50
Gen. 150:     21      -4     10.64  34.01

```

Fatal Errors

IMSLS_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

Naive Bayes – An Overview

Classification problems are characterized by a need to classify unknown patterns or data into one of m categories based upon the values of k attributes x_1, x_2, \dots, x_k . There are many algorithms for solving classification problems including discriminant analysis, neural networks and Naive Bayes. Each algorithm has its strengths and weaknesses. Discriminant analysis is robust but it requires x_1, x_2, \dots, x_k to be continuous, and since it uses a simple linear equation for the discriminant function, its error rate can be higher than the other algorithms. See [imsls_f_discriminant_analysis](#).

Neural Networks provides a linear or non-linear classification algorithm that accepts both nominal and continuous input attributes. However, network training can be unacceptably slow for problems with a larger number of attributes, typically when $k > 1000$. Naive Bayes, on the other hand, is a simple algorithm that is very fast. A Naive Bayes classifier can be trained to classify patterns involving thousands of attributes and applied to thousands of patterns. As a result, Naive Bayes is a preferred algorithm for text mining and other large classification problems. However, its computational efficiency comes at a price. The error rate for a Naive Bayes classifier is typically higher than the equivalent Neural Network classifier, although it is usually low enough for many applications such as text mining.

If C is the classification attribute and $X^T = \{x_1, x_2, \dots, x_k\}$ is the vector valued array of input attributes, the classification problem simplifies to estimating the conditional probability $P(c | X)$ from a set of training patterns. The Bayes rule states that this probability can be expressed as the ratio:

$$P(C = c | X = \{x_1, x_2, \dots, x_k\}) = \frac{P(C = c)P(X = \{x_1, x_2, \dots, x_k\} | C = c)}{P(X = \{x_1, x_2, \dots, x_k\})}$$

where c is equal to one of the target classes $0, 1, \dots, \text{n_classes} - 1$. In practice, the denominator of this expression is constant across all target classes since it is only a function of the given values of X . As a result, the Naive Bayes algorithm does not expend computational time estimating $P(X = \{x_1, x_2, \dots, x_k\})$ for every pattern.

Instead, a Naive Bayes classifier calculates the numerator $P(C = c)P(X = \{x_1, x_2, \dots, x_k\} | C = c)$ for each target class and then classifies X to the target class with the largest value, i.e.,

$$X \leftarrow \underset{\max(c=0,1,\dots,\text{n_classes}-1)}{P(C = c)P(X|C = c)}$$

The classifier simplifies this calculation by assuming conditional independence:

$$P\left(X = \{x_1, x_2, \dots, x_k\} \mid C = c\right) = \prod_{j=1}^k P(x_j \mid C = c)$$

This is equivalent to assuming that the values of the input attributes, given C , are independent of one another, i.e.

$$P(x_i \mid x_j, C = c) = P(x_i \mid C = c) \quad \text{for all } i \neq j.$$

In real world data this assumption rarely holds, yet in many cases this approach results in surprisingly low classification error rates. Thus, the estimate of $P(C = c \mid X = \{x_1, x_2, \dots, x_k\})$ from a Naive Bayes classifier is generally an approximation, classifying patterns based upon the Naive Bayes algorithm can have acceptably low classification error rates.

Function `imsls_f_naive_bayes_trainer` is used to train a classifier from a set of training patterns that contains patterns with values for both the input and target attributes. This routine stores the trained classifier into an `imsls_f_nb_classifier` data structure. The trained classifier can in turn be stored to a file using `imsls_f_nb_classifier_write`, and later retrieved using `imsls_f_nb_classifier_read`.

Classifications of new patterns with unknown classifications can be predicted by passing the trained classifier data structure, `imsls_f_nb_classifier`, to `imsls_f_naive_bayes_classification`. If necessary, memory allocated to the trained classifier can be released using `imsls_f_nb_classifier_free`.

naive_bayes_trainer



[more...](#)

Trains a Naive Bayes classifier.

Synopsis

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

```
int *imsls_f_naive_bayes_trainer (int n_patterns, int n_classes,  
int classification[ ], ..., 0)
```

The type *double* function is `imsls_d_naive_bayes_trainer`.

Required Arguments

int n_patterns (Input)
Number of training patterns.

int n_classes (Input)
Number of target classifications.

int classification[] (Input)
Array of size `n_patterns` containing the target classifications for the training patterns. These must be encoded from zero to `n_classes-1`. Any value outside this range is considered a missing value. In this case, the data in that pattern are not used to train the Naive Bayes classifier. However, any pattern with missing values is still classified after the classifier is trained.

Return Value

An array of size $(n_classes+1) \times 2$ containing the number of classification errors and the number of non-missing classifications for each target classification plus the overall totals for these errors. For $i < n_classes$, the i -th row contains the number of classification errors for the i -th class and the number of patterns with non-missing classifications for that class. The last row contains the number of classification errors totaled over all target classifications and the total number of patterns with non-missing target classifications. The memory allocated for this array can be released using `imsls_free`.

If training is unsuccessful, NULL is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_f_naive_bayes_trainer (int n_patterns, int n_classes,
    int classification[],
    IMSLS_CONTINUOUS, int n_continuous, float continuous[],
    IMSLS_NOMINAL, int n_nominal, int n_categories[], int nominal[],
    IMSLS_PRINT_LEVEL, int print_level,
    IMSLS_IGNORE_MISSING_VALUE_PATTERNS,
    IMSLS_DISCRETE_SMOOTHING_PARM, float d_lambda,
    IMSLS_CONTINUOUS_SMOOTHING_PARM, float c_lambda,
    IMSLS_ZERO_CORRECTION, float zero_correction,
    IMSLS_SELECTED_PDF, int selected_pdf[],
    IMSLS_GAUSSIAN_PDF, float means[], float stdev[],
    IMSLS_LOG_NORMAL_PDF, float logMean[], float logStdev[],
    IMSLS_GAMMA_PDF, float a[], float b[],
    IMSLS_POISSON_PDF, float theta[],
    IMSLS_USER_PDF, float pdf(),
    IMSLS_USER_PDF_WITH_PARAMS, float pdf(), void *parms,
    IMSLS_STATISTICS, float **means, float **stdev,
    IMSLS_STATISTICS_USER, float means[], float stdev[],
    IMSLS_PREDICTED_CLASS, int **predicted_class,
    IMSLS_PREDICTED_CLASS_USER, int predicted_class[],
    IMSLS_PREDICTED_CLASS_PROB, float **pred_class_prob,
    IMSLS_PREDICTED_CLASS_PROB_USER, float pred_class_prob[],
    IMSLS_CLASS_ERROR, float **class_error,
    IMSLS_CLASS_ERROR_USER, float class_error[],
    IMSLS_COUNT_TABLE, int **count_table,
    IMSLS_COUNT_TABLE_USER, int count_table[],
    IMSLS_NB_CLASSIFIER, imsls_f_nb_classifier **nb_classifier,
```

```
IMSLS_RETURN_USER, int classErrors[ ],
0)
```

Optional Arguments

IMSLS_CONTINUOUS, *int n_continuous*, *float continuous[]* (Input)

n_continuous is the number of continuous attributes and **continuous** is an array of size **n_patterns** by **n_continuous** containing the training values for the continuous attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column of **continuous** contains the values for the *j*-th continuous attribute. Missing values should be set equal to **imsls_f_machine(6) = NaN**. Patterns with both non-missing and missing values are used to train the classifier unless the **IMSLS_IGNORE_MISSING_VALUE_PATTERNS** argument is supplied. If the **IMSLS_CONTINUOUS** argument is not supplied, **n_continuous** is assumed equal to zero.

IMSLS_NOMINAL, *int n_nominal*, *int n_categories[]*, *int nominal[]* (Input)

n_nominal is the number of nominal attributes. **n_categories** is an array of length **n_nominal** containing the number of categories associated with each nominal attribute. These must all be greater than zero. **nominal** is an array of size **n_patterns** by **n_nominal** containing the training values for the nominal attributes. The *i*-th row contains the nominal input attributes for the *i*-th pattern. The *j*-th column of this matrix contains the classifications for the *j*-th nominal attribute. The values for the *j*-th nominal attribute are expected to be encoded with integers starting from 0 to **n_categories[i] - 1**. Any value outside this range is treated as a missing value. Patterns with both non-missing and missing values are used to train the classifier unless the **IMSLS_IGNORE_MISSING_VALUE_PATTERNS** option is supplied. If the **IMSLS_NOMINAL** argument is not supplied, **n_nominal** is assumed equal to zero.

IMSLS_PRINT_LEVEL, *int print_level* (Input)

Print levels for printing data warnings and final results. **print_level** should be set to one of the following values:

print_level	Description
IMSLS_NONE	Printing of data warnings and final results is suppressed.
IMSLS_FINAL	Prints final summary of Naive Bayes classifier training.
IMSLS_DATA_WARNINGS	Prints information about missing values and PDF calculations equal to zero.
IMSLS_TRACE_ALL	Prints final summary plus all data warnings associated with missing values and PDF calculations equal to zero.

Default: `IMSLS_NONE`.

IMSLS_IGNORE_MISSING_VALUE_PATTERNS, (Input)

By default, patterns with both missing and non-missing values are used to train the classifier. This option causes the algorithm to ignore patterns with one or more missing input attributes during training. However, classification predictions are still returned for all patterns.

IMSLS_DISCRETE_SMOOTHING_PARM, *float* `d_lambda` (Input)

Parameter for calculating smoothed estimates of conditional probabilities for discrete attributes. This parameter must be non-negative.

Default: Laplace smoothing of conditional probabilities, i.e. `d_lambda=1`.

IMSLS_CONTINUOUS_SMOOTHING_PARM, *float* `c_lambda` (Input)

Parameter for calculating smoothed estimates of conditional probabilities for continuous attributes. This parameter must be non-negative.

Default: No smoothing of conditional probabilities for continuous attributes, i.e. `c_lambda=0`.

IMSLS_ZERO_CORRECTION, *float* `zero_correction` (Input)

Parameter used to replace conditional probabilities equal to zero numerically. This parameter must be non-negative.

Default: No correction, i.e. `zero_correction = 0`.

IMSLS_SELECTED_PDF, *int* `selected_pdf []` (Input)

An array of length `n_continuous` specifying the distribution for each continuous input attribute. If this argument is not supplied, conditional probabilities for all continuous attributes are calculated using the Gaussian probability density function with its parameters estimated from the training patterns, i.e. `selected_pdf [i] = IMSLS_GAUSSIAN`. This argument allows users to select other distributions using the following encoding:

<code>selected_pdf[i]</code>	Probability Density Function
<code>IMSLS_GAUSSIAN</code>	Gaussian (See IMSLS_GAUSSIAN_PDF).
<code>IMSLS_LOG_NORMAL</code>	Log-normal (See IMSLS_LOG_NORMAL_PDF).
<code>IMSLS_GAMMA</code>	Gamma (See IMSLS_GAMMA_PDF).
<code>IMSLS_POISSON</code>	Poisson (See IMSLS_POISSON_PDF).
<code>IMSLS_USER</code>	User Defined (See IMSLS_USER_PDF).

`selected_pdf [i]`, specifies the probability density function for the *i*-th continuous input attribute.

IMSLS_GAUSSIAN_PDF, *float means[]*, *float stdev[]* (Input)

The **means** and **stdev** are two arrays each of size **n_gauss** by **n_classes** where **n_gauss** represents the number of Gaussian attributes as specified by optional argument **IMSLS_SELECTED_PDF** (i.e., the number of elements in **selected_pdf** equal to **IMSLS_GAUSSIAN**). The *i*-th row of **means** and **stdev** contains the means and standard deviations respectively for the *i*-th Gaussian attribute in **continuous** for each value of the target classification. **means[i*n_classes+j]** is used as the mean for the *i*-th Gaussian attribute when the target classification equals *j*, and **stdev[i*n_classes+j]** is used as the standard deviation for the *i*-th Gaussian attribute when the target classification equals *j*. This argument is ignored if **n_continuous = 0**.

Default: The means and standard deviations for all Gaussian attributes are estimated from the means and standard deviations of the training patterns. These estimates are the traditional BLUE (Best Linear Unbiased Estimates) for the parameters of a Gaussian distribution.

IMSLS_LOG_NORMAL_PDF, *float logMean[]*, *float logStdev[]* (Input)

Two arrays each of size **n_logNormal** by **n_classes** where **n_logNormal** represents the number of log-normal attributes as specified by optional argument **IMSLS_SELECTED_PDF** (i.e., the number of elements in **selected_pdf** equal to **IMSLS_LOG_NORMAL**). The *i*-th row of **logMean** and **logStdev** contains the means and standard deviations respectively for the *i*-th log-normal attribute for each value of the target classification.

logMean[i*n_classes+j] is used as the mean for the *i*-th log-normal attribute when the target classification equals *j*, and **logStdev[i*n_classes+j]** is used as the standard deviation for the *i*-th log-normal attribute when the target classification equals *j*. This argument is ignored if **n_continuous = 0**.

Default: The means and standard deviations for all log-normal attributes are estimated from the means and standard deviations of the training patterns. These estimates are the traditional MLE (Maximum Likelihood Estimates) for the parameters of a log-normal distribution.

IMSLS_GAMMA_PDF, *float a[]*, *float b[]* (Input)

Two arrays each of size **n_gamma** by **n_classes** containing the means and standard deviations for the Gamma continuous attributes, where **n_gamma** represents the number of gamma distributed continuous variables as specified by the optional argument **IMSLS_SELECTED_PDF** (i.e. the number of elements in **selected_pdf** equal to **IMSLS_GAMMA**). The *i*-th row of **a** and **b** contains the shape and scale parameters for the *i*-th Gamma attribute for each value of the target classification. **a[i*n_classes+j]** is used as the shape parameter for the *i*-th Gamma attribute when the target classification equals *j*, and **b[i*n_classes+j]** is used as the scale parameter for the *i*-th Gamma attribute when the target classification equals *j*. This argument is ignored if **n_continuous = 0**.

Default: The shape and scale parameters for all Gamma attributes are estimated from the training patterns. These estimates are the traditional MLE (Maximum Likelihood Estimates) for the parameters of a Gamma distribution.

`IMSLS_POISSON_PDF, float theta[]` (Input)

An array of size `n_poisson` by `n_classes` containing the means for the Poisson attributes, where `n_poisson` represents the number of Poisson distributed continuous variables as specified by the optional argument `IMSLS_SELECTED_PDF` (i.e. the number of elements in `selected_pdf` equal to `IMSLS_POISSON`). The i -th row of `theta` contains the means for the i -th Poisson attribute for each value of the target classification. `theta[i*n_classes+j]` is used as the mean for the i -th Poisson attribute when the target classification equals j . This argument is ignored if `n_continuous = 0`.

Default: The means (`theta`) for all Poisson attributes are estimated from the means of the training patterns. These estimates are the traditional MLE (Maximum Likelihood Estimates) for the parameters of a Poisson distribution.

`IMSLS_USER_PDF, float pdf(int index[], float x)` (Input)

The user-supplied probability density function and parameters used to calculate the conditional probability density for continuous input attributes is required when `selected_pdf[i] = IMSLS_USER`.

When `pdf` is called, `x` will equal `continuous[i*n_continuous+j]`, and `index` is an array of length 3 which will contain the following values for i , j , and k :

<code>index[0]</code>	i = pattern index
<code>index[1]</code>	j = attribute index
<code>index[2]</code>	k = target classification

The pattern index ranges from 0 to `n_patterns-1` and identifies the pattern index for `x`. The attributes index ranges from 0 to `n_categories[i]-1`, and $k = \text{classification}[i]$.

This argument is ignored if `n_continuous = 0`. By default the Gaussian PDF is used for calculating the conditional probability densities using either the means and variances calculated from the training patterns or those supplied in `IMSLS_GAUSSIAN_PDF`.

On some platforms, `imsls_f_naive_bayes_trainer` can evaluate the user-supplied function `pdf` in parallel. This is done only if the function `imsls_omp_options` is called to flag user-defined functions as thread-safe. A function is thread-safe if there are no dependencies between calls. Such dependencies are usually the result of writing to global or static variables.

IMSLS_USER_PDF_WITH_PARMS, *float pdf(int index[], float x, void *parms), void *parms*
(Input)

The user-supplied probability density function and parameters used to calculate the conditional probability density for continuous input attributes is required when `selected_pdf[i]=IMSLS_USER`. `pdf` also accepts a pointer to `parms` supplied by the user. The parameters pointed to by `parms` are passed to `pdf` each time it is called. For an explanation of the other arguments, see `IMSLS_USER_PDF`.

IMSLS_STATISTICS, *float **means, float **stdev* (Output)

The address of pointers to two arrays of size `n_continuous` by `n_classes` containing the means and standard deviations for the continuous attributes segmented by the target classes. The structure of these matrices is identical to the structure described for the `IMSLS_GAUSSIAN_PDF` argument. The i -th row of `means` and `stdev` contains the means and standard deviations respectively of the i -th continuous attribute for each value of the target classification. That is, `means[i*n_classes+j]` is the mean for the i -th continuous attribute when the target classification equals j , and `stdev[i*n_classes+j]` is the standard deviation for the i -th continuous attribute when the target classification equals j , unless there are no training patterns for this condition. If there are no training patterns in the i, j -th cell then the mean and standard deviation for that cell is computed using the mean and standard deviation for the i -th continuous attribute calculated using all of its non-missing values. Standard deviations are estimated using the minimum variance unbiased estimator.

IMSLS_STATISTICS_USER, *float means[], float stdev[]* (Output)

Storage for matrices `means` and `stdev` provided by the user. See `IMSLS_STATISTICS`.

IMSLS_PREDICTED_CLASS, *int **pred_class* (Output)

The address of a pointer to an array of size `n_patterns` containing the predicted classification for each training pattern.

IMSLS_PREDICTED_CLASS_USER, *int pred_class[]* (Output)

Storage for array `pred_class` provided by the user. See `IMSLS_PREDICTED_CLASS`.

IMSLS_PREDICTED_CLASS_PROB, *float **pred_class_prob* (Output)

The address of a pointer to an array of size `n_patterns` by `n_classes`. The values in the i -th row are the predicted classification probabilities associated with the target classes. `pred_class_prob[i*n_classes+j]` is the estimated probability that the i -th pattern belongs to the j -th target class.

IMSLS_PREDICTED_CLASS_PROB_USER, *float pred_class_prob[]* (Output)

Storage for array `pred_class_prob` provided by the user. See `IMSLS_PREDICTED_CLASS_PROB` for a description.

IMSLS_CLASS_ERROR, *float **class_error* (Output)

The address of a pointer to an array with **n_patterns** containing the classification probability errors for each pattern in the training data. The classification error for the *i*-th training pattern is equal to $1 - \text{pred_class_prob}[i * \text{n_classes} + k]$ where $k = \text{classification}[i]$.

IMSLS_CLASS_ERROR_USER, *float class_error[]* (Output)

Storage for array **class_error** is provided by the user. See **IMSLS_CLASS_ERROR** for a description.

IMSLS_COUNT_TABLE *int **count_table* (Output)

The address of a pointer to an array of size

$$(m + 1) \left(\text{n_classes} + \text{n_classes} \sum_{i=0}^m \text{n_categories}[i] \right)$$

where $m = \text{n_nominal} - 1$.

count_table $[i * \text{n_nominal} * \text{n_classes} + j * \text{n_classes} + k]$ is equal to the number of training patterns for the *i*-th nominal attribute, when the **classification** $[i] = j$ and **nominal** $[i * \text{n_classes} + j] = k$.

IMSLS_COUNT_TABLE_USER, *int count_table[]* (Output)

Storage for matrix **count_table** provided by the user. See **IMSLS_COUNT_TABLE**.

IMSLS_NB_CLASSIFIER, *Impls_f_nb_classifier **nb_classifier* (Output)

The address of a pointer to an *Impls_f_nb_classifier* structure. Upon return, the structure is populated with the trained Naive Bayes classifier. This is required input to **impls_f_naive_bayes_classification**. Memory allocated to this structure is released using **impls_f_nb_classifier_free**.

IMSLS_RETURN_USER, *int classErrors[]* (Output)

An array of size $(\text{n_classes} + 1) \times 2$ containing the number of classification errors and the number of non-missing classifications for each target classification and the overall totals. For $0 \leq i < \text{n_classes}$, the *i*-th row contains the number of classification errors for the *i*-th class and the number of patterns with non-missing classifications for that class. The last row contains the number of classification errors totaled over all target classifications and the total number of patterns with non-missing target classifications.

Description

Function `imsls_f_naive_bayes_trainer` trains a Naive Bayes classifier for classifying data into one of `n_classes` target classes. Input attributes can be a combination of both nominal and continuous data. Ordinal data can be treated as either nominal attributes or continuous. If the distribution of the ordinal data is known or can be approximated using one of the continuous distributions, then associating them with continuous attributes allows a user to specify that distribution. Missing values are allowed.

Let C be the classification attribute with target categories $0, 1, \dots, n_classes-1$, and let $X^T = \{x_1, x_2, \dots, x_k\}$ be a vector valued array of $k = n_nominal + n_continuous$ input attributes. The classification problem simplifies to estimate the conditional probability $P(C|X)$ from a set of training patterns. The Bayes rule states that this probability can be expressed as the ratio:

$$P(C = c | X = \{x_1, x_2, \dots, x_k\}) = \frac{P(C = c)P(X = \{x_1, x_2, \dots, x_k\} | C = c)}{P(X = \{x_1, x_2, \dots, x_k\})}$$

where c is equal to one of the target classes $0, 1, \dots, n_classes-1$. In practice, the denominator of this expression is constant across all target classes since it is only a function of the given values of X . As a result, the Naive Bayes algorithm does not expend computational time estimating $P(X = \{x_1, x_2, \dots, x_k\})$ for every pattern.

Instead, a Naive Bayes classifier calculates the numerator $P(C = c)P(X = \{x_1, x_2, \dots, x_k\} | C = c)$ for each target class and then classifies X to the target class with the largest value, i.e.,

$$X \leftarrow \underset{\max(c=0,1,n_classes-1)}{P(C = c)P(X|C = c)}$$

The classifier simplifies this calculation by assuming conditional independence. That is it assumes that:

$$P(X = \{x_1, x_2, \dots, x_k\} | C = c) = \prod_{j=1}^k P(x_j | C = c)$$

This is equivalent to assuming that the values of the input attributes, given C , are independent of one another, i.e.,

$$P(x_i | x_j, C = c) = P(x_i | C = c) \quad \text{for all } i \neq j$$

In real world data this assumption rarely holds, yet in many cases this approach results in surprisingly low classification error rates. Thus, the estimate of $P(C = c | X = \{x_1, x_2, \dots, x_k\})$ from a Naive Bayes classifier is generally an approximation. Classifying patterns based upon the Naive Bayes algorithm can have acceptably low classification error rates.

For nominal attributes, this implementation of the Naive Bayes classifier estimates conditional probabilities using a smoothed estimate:

$$P(x_j|C = c) = \frac{\#N\{x_j \cap C = c\} + \lambda}{\#N\{C = c\} + \lambda J}$$

where $\#N\{Z\}$ is the number of training patterns with attribute Z and J is equal to the number of categories associated with the j -th nominal attribute.

The probability $P(C = c)$ is also estimated using a smoothed estimate:

$$P(C = c) = \frac{\#N\{C = c\} + \lambda}{n_patterns + \lambda \times n_classes}$$

These estimates correspond to the maximum a priori (MAP) estimates for a Dirichlet prior assuming equal priors. The smoothing parameter can be any non-negative value. Setting $\lambda = 0$ corresponds to no smoothing. The default smoothing used in this algorithm, $\lambda = 1$, is commonly referred to as Laplace smoothing. This can be changed using the optional argument `IMSLS_DISCRETE_SMOOTHING_PARM`.

For continuous attributes, the same conditional probability $P(x_j|C = c)$ in the Naive Bayes formula is replaced with the conditional probability density function $f(x_j|C = c)$. By default, the density function for continuous attributes is the Gaussian density function:

$$f(x_j|C = c) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x_j - \mu)^2}{2\sigma^2}}$$

where μ and σ are the conditional mean and variance, i.e. the mean and variance of x_j when $C = c$. By default the conditional mean and standard deviations are estimated using the sample mean and standard deviation of the training patterns. These are returned in the optional argument `IMSLS_STATISTICS`.

In addition to the default `IMSLS_GAUSSIAN`, users can select three other continuous distributions to model the continuous attributes using the argument `IMSLS_SELECTED_PDF`. These are the Log Normal, Gamma, and Poisson distributions selected by setting the entries in `selected_pdf` to `IMSLS_LOG_NORMAL`, `IMSLS_GAMMA` or `IMSLS_POISSON`. Their probability density functions are equal to:

$$f(x_j|C = c) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(\ln(x_j) - \mu)^2}{2\sigma^2}}$$

$$f(x_j|C = c) = \frac{x_j^{a-1} e^{-x_j/b}}{b^a \Gamma(a)}, x_j > 0, a > 0 \text{ and } b > 0,$$

and

$$f(x_j|C = c) = \frac{\theta^x e^{-\theta}}{x!}, x_j > 0 \text{ and } \theta > 0.$$

By default parameters for these distributions are estimated from the training patterns using the maximum likelihood method. However, they can also be supplied using the optional input arguments

`IMSLG_GAUSSIAN_PDF`, `IMSLG_LOG_NORMAL_PDF`, `IMSLG_GAMMA_PDF` and `IMSLG_POISSON_PDF`.

The default Gaussian PDF can be changed and each continuous attribute can be assigned a different density function using the argument `IMSLG_SELECTED_PDF`. If any entry in `selected_pdf` is equal to `IMSLG_USER`, the user must supply their own PDF calculation using the `IMSLG_USER_PDF` argument. Each continuous attribute can be modeled using a different distribution if appropriate.

Smoothing conditional probability calculations for continuous attributes is controlled by the `IMSLG_CONTINUOUS_SMOOTHING_PARM` and `IMSLG_ZERO_CORRECTION` optional arguments. By default conditional probability calculations for continuous attributes are unadjusted for calculations near zero. If the value of `c_lambda` is set using the `IMSLG_CONTINUOUS_SMOOTHING_PARM` argument, the algorithm adds `c_lambda` to each continuous probability calculation. This is similar to the effect of `d_lambda` for the corresponding discrete calculations. By default `c_lambda=0`.

The value of `zero_correction` from the `IMSLG_ZERO_CORRECTION` argument is used when $(f(x|C = c) + c_lambda) = 0$. If this condition occurs, the conditional probability is replaced with the value of `zero_correction`. By default `zero_correction = 0`.

Examples

Example 1

Fisher's (1936) Iris data is often used for benchmarking classification algorithms. It is one of the IMSL data sets and consists of the following continuous input attributes and classification target:

Continuous Attributes: X0(sepal length), X1(sepal width), X2(petal length), and X3(petal width)

Classification (Iris Type): Setosa, Versicolour, or Virginica.

This example trains a Naive Bayes classifier using 150 training patterns with these data.

```
#include <imsls.h>
#include <stdio.h>
int main(){
    int i, j;
    int n_patterns    =150; /* 150 training patterns          */
    int n_continuous  =4;  /* four continuous input attributes */
    int n_classes     =3;  /* three classification categories */
    int classification[150], *classErrors, *predictedClass;
    float *pred_class_prob, continuous[4*150] ;
    float *irisData;      /* Fishers Iris Data */
    char *classLabel[3] = {"Setosa", "Versicolour", "Virginica "};
    Imsls_f_nb_classifier *nb_classifier;

    imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    /* irisData[]: The raw data matrix. This is a 2-D matrix
       with 150 rows and 5 columns. The last 4 columns are the
       continuous input attributes and the 1st column is the
       classification category (1-3). These data contain no */
    /* nominal input attributes. */
    irisData = imsls_f_data_sets(3,0);
    /* Data corrections described in the KDD data mining archive */
    irisData[5*34+4] = 0.1;
    irisData[5*37+2] = 3.1;
    irisData[5*37+3] = 1.5;
    /* setup the required input arrays from the data matrix */
    for(i=0; i<n_patterns; i++){
        classification[i] = (int) irisData[i*5]-1;
        for(j=1; j<=n_continuous; j++)
            continuous[i*n_continuous+j-1] = irisData[i*5+j];
    }

    classErrors = imsls_f_naive_bayes_trainer(n_patterns,
        n_classes, classification,
        IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_NB_CLASSIFIER, &nb_classifier, 0);

    printf("    Iris Classification Error Rates\n");
    printf("-----\n");
    printf("  Setosa Versicolour Virginica |  TOTAL\n");
    printf("   %d/%d    %d/%d    %d/%d    |  %d/%d\n",
        classErrors[0], classErrors[1],
        classErrors[2], classErrors[3],
        classErrors[4], classErrors[5],
        classErrors[6], classErrors[7]);
    printf("-----\n\n");
};
```

Output

For Fisher's data, the Naive Bayes classifier incorrectly classified 6 of the 150 training patterns.

```
    Iris Classification Error Rates
-----
```

Setosa	Versicolour	Virginica	TOTAL
0/50	3/50	3/50	6/150

Example 2

This example trains a Naive Bayes classifier using 24 training patterns with four nominal input attributes. It illustrates the output available from the optional argument `IMSLS_PRINT_LEVEL`.

The first nominal attribute has three classifications and the others have three. The target classifications are contact lenses prescription: hard, soft or neither recommended. These data are benchmark data from the Knowledge Discovery Databases archive maintained at the University of California, Irvine:

<http://archive.ics.uci.edu/ml/datasets/Lenses>.

```
#include <imsls.h>

int main(){
    int inputData[5 * 24] = { /* DATA MATRIX */
        1, 1, 1, 1, 3, 1, 1, 1, 2, 2, 1, 1, 2, 1, 3, 1, 1, 2, 2, 1, 1,
        2, 1, 1, 3, 1, 2, 1, 2, 2, 1, 2, 2, 1, 3, 1, 2, 2, 2, 1, 2, 1,
        1, 1, 3, 2, 1, 1, 2, 2, 2, 1, 2, 1, 3, 2, 1, 2, 2, 1, 2, 2, 1,
        1, 3, 2, 2, 1, 2, 2, 2, 2, 2, 1, 3, 2, 2, 2, 3, 3, 1, 1, 1,
        3, 3, 1, 1, 2, 3, 3, 1, 2, 1, 3, 3, 1, 2, 2, 1, 3, 2, 1, 1, 3,
        3, 2, 1, 2, 2, 3, 2, 2, 1, 3, 3, 2, 2, 2, 3
    };

    int i, j;
    int n_patterns = 24; /* 24 training patterns */
    int n_nominal = 4; /* 2 nominal input attributes */
    int n_classes = 3; /* three classification categories */
    int n_categories[4] = {3, 2, 2, 2};
    int nominal[4 * 24], classification[24], *classErrors;
    char *classLabel[3] = {"Hard ", "Soft ", "Neither"};

    imsls_omp_options(
        IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1,
        0);

    /* setup the required input arrays from the data matrix */
    /* subtract 1 from the data to ensure classes start at zero */
    for(i = 0; i < n_patterns; i++){
        classification[i] = inputData[i * 5 + 4] - 1;

        for(j = 0; j < n_nominal; j++)
            nominal[i * n_nominal + j] = inputData[i * 5 + j] - 1;
    }

    classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
        classification,
        IMSLS_NOMINAL, n_nominal, n_categories, nominal,
        IMSLS_PRINT_LEVEL,
        IMSLS_FINAL,
        0);
}
```

Output

For these data, only one of the 24 training patterns is misclassified, pattern 17. The target classification for that pattern is 2 = "Neither". However, since $P(class = 2) = 0.3491 < P(class = 1) = 0.5085$, pattern 17 is classified as $class = 1$, "Soft Contacts" recommended. The classification error for this probability is calculated as $1.0 - 0.3491 = 0.6509$.

```

-----UNCONDITIONAL TARGET CLASS PROBABILITIES-----
P(Class=0) = 0.1852 P(Class=1) = 0.2222 P(Class=2) = 0.5926
-----
-----CONDITIONAL PROBABILITIES-----
-----NOMINAL ATTRIBUTE 0 WITH 3 CATEGORIES-----
P(X(0)=0|Class=0)=0.4286 P(X(0)=1|Class=0)=0.2857 P(X(0)=2|Class=0)=0.2857
P(X(0)=0|Class=1)=0.3750 P(X(0)=1|Class=1)=0.3750 P(X(0)=2|Class=1)=0.2500
P(X(0)=0|Class=2)=0.2778 P(X(0)=1|Class=2)=0.3333 P(X(0)=2|Class=2)=0.3889

-----
-----NOMINAL ATTRIBUTE 1 WITH 2 CATEGORIES-----
P(X(1)=0|Class=0) = 0.6667 P(X(1)=1|Class=0) = 0.3333
P(X(1)=0|Class=1) = 0.4286 P(X(1)=1|Class=1) = 0.5714
P(X(1)=0|Class=2) = 0.4706 P(X(1)=1|Class=2) = 0.5294

-----
-----NOMINAL ATTRIBUTE 2 WITH 2 CATEGORIES-----
P(X(2)=0|Class=0) = 0.1667 P(X(2)=1|Class=0) = 0.8333
P(X(2)=0|Class=1) = 0.8571 P(X(2)=1|Class=1) = 0.1429
P(X(2)=0|Class=2) = 0.4706 P(X(2)=1|Class=2) = 0.5294

-----
-----NOMINAL ATTRIBUTE 3 WITH 2 CATEGORIES-----
P(X(3)=0|Class=0) = 0.1667 P(X(3)=1|Class=0) = 0.8333
P(X(3)=0|Class=1) = 0.1429 P(X(3)=1|Class=1) = 0.8571
P(X(3)=0|Class=2) = 0.7647 P(X(3)=1|Class=2) = 0.2353
-----

```

PATTERN	TRAINING CLASS			PREDICTED CLASS		
	P(class=0)	P(class=1)	P(class=2)	CLASS	CLASS	ERROR
0	0.0436	0.1297	0.8267	2	2	0.1733
1	0.1743	0.6223	0.2034	1	1	0.3777
2	0.1863	0.0185	0.7952	2	2	0.2048
3	0.7238	0.0861	0.1901	0	0	0.2762
4	0.0194	0.1537	0.8269	2	2	0.1731
5	0.0761	0.7242	0.1997	1	1	0.2758
6	0.0920	0.0243	0.8836	2	2	0.1164
7	0.5240	0.1663	0.3096	0	0	0.4760
8	0.0253	0.1127	0.8621	2	2	0.1379
9	0.1182	0.6333	0.2484	1	1	0.3667
10	0.1132	0.0168	0.8699	2	2	0.1301
11	0.6056	0.1081	0.2863	0	0	0.3944
12	0.0111	0.1327	0.8562	2	2	0.1438
13	0.0500	0.7138	0.2362	1	1	0.2862
14	0.0535	0.0212	0.9252	2	2	0.0748
15	0.3937	0.1875	0.4188	2	2	0.5812
16	0.0228	0.0679	0.9092	2	2	0.0908
17	0.1424	0.5085	0.3491	2	1	0.6509
18	0.0994	0.0099	0.8907	2	2	0.1093
19	0.5986	0.0712	0.3301	0	0	0.4014
20	0.0101	0.0805	0.9093	2	2	0.0907

21	0.0624	0.5937	0.3439	1	1	0.4063
22	0.0467	0.0123	0.9410	2	2	0.0590
23	0.3909	0.1241	0.4850	2	2	0.5150

CLASSIFICATION ERRORS						
Classification 0: 0/4						
Classification 1: 0/5						
Classification 2: 1/15						
Total Errors: 1/24						

Example 3

This example illustrates the power of Naive Bayes classification for text mining applications. This example uses the spam benchmark data available from the Knowledge Discovery Databases archive maintained at the University of California, Irvine: <http://archive.ics.uci.edu/ml/datasets/Spambase> and is one of the IMSL data sets.

These data consist of 4601 patterns consisting of 57 continuous attributes and one classification binary classification attribute. 41% of these patterns are classified as spam and the remaining as non-spam. The first 54 continuous attributes are word or symbol percentages. That is, they are percents scaled from 0 to 100% representing the percentage of words or characters in the email that contain a particular word or character. The last three continuous attributes are word lengths. For a detailed description of these data visit the KDD archive at the above link.

In this example, the program was written to evaluate alternatives for modeling the continuous attributes. Since some are percentages and others are lengths with widely different ranges, the classification error rate can be influenced by scaling. Percentages are transformed using the arcsin/square root transformation $y = \sin^{-1}(\sqrt{p})$. This transformation often produces a continuous attribute that is more closely approximated by a Gaussian distribution. There are a variety of possible transformations for the word length attributes. In this example, the square root transformation is compared to a classifier with no transformation.

In addition, since this Naive Bayes algorithm allows users to select individual statistical distributions for modeling continuous attributes, the Gaussian and Log Normal distributions are investigated for modeling the continuous attributes.

```
#include <imsls.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
void print_error_rates(int classErrors[]);
int main(){
    int i, j;
    /* Inputs assuming all attributes, except family history,
    are continuous */
    int n_patterns;      /* 4601 */
    int n_variables;     /* 57 + 1 classification */
    int n_classes       = 2; /* (spam or no spam) */
    int n_continuous    = 57;
    int *classErrors    = NULL;
    int *classification = NULL;
    int selected_pdf[57];
```

```

/* additional double variables */
float *continuous, *unscaledContinuous;
float *spamData;
int n_spam = 0;
static char *fmt = "%10.2f";

imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

spamData = imsls_f_data_sets(11, IMSLS_N_OBSERVATIONS, &n_patterns,
    IMSLS_N_VARIABLES, &n_variables, 0);

continuous = (float*) malloc( n_patterns * (n_variables-1)
    * sizeof(float));
unscaledContinuous = (float*) malloc( n_patterns * (n_variables-1)
    * sizeof(float));
classification = (int*) malloc( n_patterns*sizeof(int) );

for(i=0; i<n_patterns; i++){
    for(j=0; j<(n_variables-1); j++) {
        if (j<54) {
            continuous[i*(n_variables-1)+j] = (float)
                asin(sqrt( spamData[i*n_variables+j]/100));
        } else {
            continuous[i*(n_variables-1)+j] =
                spamData[i*n_variables+j];
        }
        unscaledContinuous[i*(n_variables-1)+j] =
            spamData[i*n_variables+j];
    }
    classification[i] = (int)spamData[(i*n_variables)+n_variables-1];
    if(classification[i] == 1) n_spam++;
}

printf("Number of Patterns = %d \n", n_patterns);
printf(" Number Classified as Spam = %d \n\n", n_spam);

classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
    classification,
    IMSLS_CONTINUOUS, n_continuous, unscaledContinuous, 0);
printf("  Unscaled Gaussian Classification Error Rates \n");
printf("      No Attribute Transformations          \n");
printf("  All Attributes Modeled as Gaussian Variates.\n");
print_error_rates(classErrors);
imsls_free(classErrors);

classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
    classification, IMSLS_CONTINUOUS, n_continuous, continuous, 0);
printf("  Scaled Gaussian Classification Error Rates \n");
printf("  Arsin(sqrt) transformation of first 54 Vars. \n");
printf("  All Attributes Modeled as Gaussian Variates. \n");
print_error_rates(classErrors);
imsls_free(classErrors);

for(i=0; i<54; i++){
    selected_pdf[i] = IMSLS_GAUSSIAN;
}
for(i=54; i<57; i++){
    selected_pdf[i] = IMSLS_LOG_NORMAL;
}

```

```

    }
    classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
        classification,
        IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_SELECTED_PDF, selected_pdf, 0);
    printf(" Gaussian/Log Normal Classification Error Rates \n");
    printf(" Arsin(sqrt) transformation of 1st 54 Attributes. \n");
    printf(" Gaussian - 1st 54 & Log Normal - last 3 Attributes\n");
    print_error_rates(classErrors);
    imsls_free(classErrors);

    /* scale continuous attributes using z-score scaling */
    for(i=0; i<n_patterns; i++){
        for(j=54; j<57; j++) continuous[i*n_continuous+j] = (float)
            sqrt(unscaledContinuous[i*n_continuous+j]);
    }
    for(i=0; i<57; i++){
        selected_pdf[i] = IMSLS_GAUSSIAN;
    }
    classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
        classification,
        IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_SELECTED_PDF, selected_pdf, 0);
    printf("      Scaled Classification Error Rates      \n");
    printf(" Arsin(sqrt) transformation of 1st 54 Attributes\n");
    printf("      sqrt() transformation for last 3 Attributes \n");
    printf(" All Attributes Modeled as Gaussian Variates. \n");
    print_error_rates(classErrors);
    imsls_free(classErrors);

    for(i=54; i<57; i++){
        selected_pdf[i] = IMSLS_LOG_NORMAL;
    }
    classErrors = imsls_f_naive_bayes_trainer(n_patterns, n_classes,
        classification, IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_SELECTED_PDF, selected_pdf, 0);
    printf("      Scaled Classification Error Rates\n");
    printf(" Arsin(sqrt) transformation of 1st 54 Attributes \n");
    printf(" and sqrt() transformation for last 3 Attributes \n");
    printf(" Gaussian - 1st 54 & Log Normal - last 3 Attributes\n");
    print_error_rates(classErrors);
    imsls_free(classErrors);
}

void print_error_rates(int classErrors[]){
    float p0, p1, p2;
    p0 = (float)100.0*classErrors[0]/classErrors[1];
    p1 = (float)100.0*classErrors[2]/classErrors[3];
    p2 = (float)100.0*classErrors[4]/classErrors[5];
    printf("-----\n");
    printf("    Not Spam      Spam      |    TOTAL\n");
    printf(" %d/%d=%4.1f%%  %d/%d=%4.1f%% | %d/%d=%4.1f%%\n",
        classErrors[0], classErrors[1],
        p0, classErrors[2], classErrors[3],
        p1, classErrors[4], classErrors[5], p2);
    printf("-----\n\n");
}

```

Output

If the continuous attributes are left untransformed and modeled using the Gaussian distribution, the overall classification error rate is 18.4% with most of these occurring when spam is classified as "not spam." The error rate for correctly classifying non-spam is 26.6%.

The lowest overall classification error rate occurs when the percentages are transformed using the arc-sin/square root transformation and the length attributes are untransformed using logs. Representing the transformed percentages as Gaussian attributes and the transformed lengths as log-normal attributes reduces the overall error rate to 14.2%. However, although the error rate for correctly classifying non-spam email is low for this case, the error rate for correctly classifying spam is high, about 28%.

In the end, the best model to identify spam may depend upon which type of error is more important, incorrectly classifying non-spam email or incorrectly classifying spam.

```

Data File Opened Successfully
Number of Patterns = 4601
Number Classified as Spam = 1813

Unscaled Gaussian Classification Error Rates
No Attribute Transformations
All Attributes Modeled as Gaussian Variates.
-----
Not Spam      Spam      |  TOTAL
743/2788=26.6% 102/1813= 5.6% | 845/4601=18.4%
-----

Scaled Gaussian Classification Error Rates
Arasin(sqrt) transformation of first 54 Vars.
All Attributes Modeled as Gaussian Variates.
-----
Not Spam      Spam      |  TOTAL
84/2788= 3.0% 508/1813=28.0% | 592/4601=12.9%
-----

Gaussian/Log Normal Classification Error Rates
Arasin(sqrt) transformation of 1st 54 Attributes.
Gaussian - 1st 54 & Log Normal - last 3 Attributes
-----
Not Spam      Spam      |  TOTAL
81/2788= 2.9% 519/1813=28.6% | 600/4601=13.0%
-----

Scaled Classification Error Rates
Arasin(sqrt) transformation of 1st 54 Attributes
sqrt() transformation for last 3 Attributes
All Attributes Modeled as Gaussian Variates.
-----
Not Spam      Spam      |  TOTAL
74/2788= 2.7% 595/1813=32.8% | 669/4601=14.5%
-----

Scaled Classification Error Rates
Arasin(sqrt) transformation of 1st 54 Attributes
and sqrt() transformation for last 3 Attributes

```

Gaussian - 1st 54 & Log Normal - last 3 Attributes

Not Spam	Spam	TOTAL
73/2788= 2.6%	602/1813=33.2%	675/4601=14.7%

Fatal Errors

IMSL5_STOP_USER_FCN	Request from user supplied function to stop algorithm. User flag = "#".
IMSL5_N_OBS_PER_CLASS	Class # has # observation(s). All classes must have at least 2 observations.

naive_bayes_classification

Classifies unknown patterns using a previously trained Naive Bayes classifier. The classifier is contained in an *Imsls_f_nb_classifier* data structure, which is optional output from [imsls_f_naive_bayes_trainer](#).

Synopsis

```
#include <imsls.h>

int *imsls_f_naive_bayes_classification (Imsls_f_nb_classifier *nb_classifier,
int n_patterns, ..., 0)
```

The type *double* function is `imsls_d_naive_bayes_classification`.

Required Arguments

Imsls_f_nb_classifier *nb_classifier (Input)
 Pointer to a structure of the type *Imsls_f_nb_classifier* from `imsls_f_naive_bayes_trainer`.

int n_patterns (Input)
 Number of patterns to classify.

Return Value

Pointer to an array of size `n_patterns` containing the predicted classification associated with each input pattern.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_f_naive_bayes_classification (Imsls_f_nb_classifier nb_classifier,
int n_patterns,
IMSLS_NOMINAL, int nominal [],
IMSLS_CONTINUOUS, float continuous [],
IMSLS_PRINT_LEVEL, int print_level,
IMSLS_USER_PDF, float pdf ( ),
```

```

IMSLS_USER_PDF_WITH_PARMS, float pdf ( ), void *parms,
IMSLS_PREDICTED_CLASS_PROB, float **pred_class_prob,
IMSLS_PREDICTED_CLASS_PROB_USER, float pred_class_prob[ ],
IMSLS_RETURN_USER, int classification[ ],
0)

```

Optional Arguments

IMSLS_NOMINAL, *int nominal[]* (Input)

nominal is an array of size `n_patterns` by `nb_classifier->n_nominal` containing values for the nominal input attributes. The *i*-th row contains the nominal input attributes for the *i*-th pattern. The *j*-th column of this matrix contains the classifications for the *j*-th nominal attribute. They must be encoded with integers starting from 0 to `nb_classifier->n_categories[i]-1`. Any value outside this range is treated as a missing value. If `nb_classifier->n_nominal=0`, this array is ignored.

IMSLS_CONTINUOUS, *float continuous[]* (Input)

continuous is an array of size `n_patterns` by `nb_classifier->n_continuous` containing values for the continuous input attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column of this matrix contains the values for the *j*-th continuous attribute. Missing values should be set equal to `imsls_f_machine(6)=NaN`. Patterns with missing values are still used to train the classifier unless the `IMSLS_IGNORE_MISSING_VALUES` option is supplied. If `nb_classifier->n_continuous=0`, this matrix is ignored.

IMSLS_PRINT_LEVEL, *int print_level* (Input)

Print levels for printing data warnings and final results. `print_level` should be set to one of the following values:

print_level	Description
IMSLS_NONE	Printing of data warnings and final results is suppressed.
IMSLS_FINAL	Prints final summary of Naive Bayes classifier training.
IMSLS_DATA_WARNINGS	Prints information about missing values and PDF calculations equal to zero.
IMSLS_TRACE_ALL	Prints final summary plus all data warnings associated with missing values and PDF calculations equal to zero.

Default: `IMSLS_NONE`.

IMSLS_USER_PDF, *float pdf (int index[], float x)* (Input)

The user-supplied probability density function and parameters used to calculate the conditional probability density for continuous input attributes is required when the classifier was trained with **selected_pdf[i]= IMSLS_USER**.

When **pdf** is called, **x** will equal **continuous[i*n_continuous+j]**, and **index** will contain the following values for **i**, **j**, and **k**:

index[0]	i = pattern index
index[1]	j = attribute index
index[2]	k = target classification

The pattern index ranges from 0 to **n_patterns-1** and identifies the pattern index for **x**. The attributes index ranges from 0 to **n_categories[i]-1**, and **k=classification[i]**.

This argument is ignored if **n_continuous = 0**. By default the Gaussian PDF is used for calculating the conditional probability densities using either the means and variances calculated from the training patterns or those supplied in **IMSLS_GAUSSIAN_PDF**.

IMSLS_USER_PDF_WITH_PARMS, *float pdf(int index[], float x, void *parms), void *parms* (Input)

The user-supplied probability density function and parameters used to calculate the conditional probability density for continuous input attributes is required when **selected_pdf[i]= IMSLS_USER**. **pdf** also accepts a pointer to **parms** supplied by the user. The parameters pointed to by **parms** are passed to **pdf** each time it is called. For an explanation of the other arguments, see **IMSLS_USER_PDF**.

IMSLS_PREDICTED_CLASS_PROB, *float **pred_class_prob*, (Output)

The address of a pointer to an array of size **n_patterns** by **n_classes**, where **n_classes** is the number of target classifications. The values in the *i*-th row are the predicted classification probabilities associated with the target classes. **pred_class_prob[i*n_classes+j]** is the estimated probability that the *i*-th pattern belongs to the *j*-th target classes.

IMSLS_PREDICTED_CLASS_PROB_USER, *float pred_class_prob[]* (Output)

Storage for array **pred_class_prob** is provided by the user. See **IMSLS_PREDICTED_CLASS_PROB** for a description.

IMSLS_RETURN_USER, *int classification[]* (Output)

An array of length **n_patterns** containing the predicted classifications for each pattern described by the input attributes in **nominal** and **continuous**.

Description

Function `imsls_f_naive_bayes_classification` estimates classification probabilities from a previously trained Naive Bayes classifier. Two arrays are used to describe the values of the nominal and continuous attributes used for calculating these probabilities. The predicted classification returned by this function is the class with the largest estimated classification probability. The classification probability estimates for each pattern can be obtained using the optional argument `IMSLS_PREDICTED_CLASS_PROB`.

Examples

Example 1

Fisher's (1936) Iris data is often used for benchmarking classification algorithms. It is one of the IMSL data sets and consists of the following continuous input attributes and classification target:

Continuous Attributes: X0(sepal length), X1(sepal width), X2(petal length), and X3(petal width)

Classification (Iris Type): Setosa, Versicolour or Virginica.

This example trains a Naive Bayes classifier using 150 training patterns from Fisher's data then classifies ten unknown plants using their sepal and petal measurements.

```
#include <imsls.h>
#include <stdio.h>
int main(){
    int i, j;
    int n_patterns    =150; /* 150 training patterns          */
    int n_continuous  =4;  /* four continuous input attributes */
    int n_classes     =3;  /* three classification categories */
    int classification[150], *classErrors, *predictedClass;
    float *pred_class_prob, continuous[150*4] ;
    float *irisData;      /* Fishers Iris Data */
    char *classLabel[] = {"Setosa   ", "Versicolour", "Virginica "};
    char dashes[] = {
        "-----"
    };

    Imsls_f_nb_classifier *nb_classifier;

    /* irisData[]: The raw data matrix. This is a 2-D matrix with 150
    /*                rows and 5 columns. The last 4 columns are the
    /*                continuous input attributes and the 1st column is
    /*                the classification category (1-3). These data
    /*                contain no categorical input attributes.          */
    irisData = imsls_f_data_sets(3,0);
    /* Data corrections described in the KDD data mining archive      */
    irisData[5*34+4] = 0.1;
    irisData[5*37+2] = 3.1;
    irisData[5*37+3] = 1.5;
    /* setup the required input arrays from the data matrix */
    for(i=0; i<n_patterns; i++){
```

```

        classification[i] = (int) irisData[i*5]-1;
        for(j=1; j<=n_continuous; j++) {
            continuous[i*n_continuous+j-1] = irisData[i*5+j];
        }
    }

    classErrors = imsls_f_naive_bayes_trainer(
        n_patterns, n_classes, classification,
        IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_NB_CLASSIFIER, &nb_classifier, 0);

    printf("    Iris Classification Error Rates\n");
    printf("%s\n", dashes);
    printf("    Setosa Versicolour Virginica | TOTAL\n");
    printf("    %d/%d      %d/%d      %d/%d      | %d/%d\n",
        classErrors[0], classErrors[1],
        classErrors[2], classErrors[3], classErrors[4],
        classErrors[5], classErrors[6], classErrors[7]);
    printf("%s\n\n", dashes);

    /* CALL NAIVE_BAYES_CLASSIFICATION ***** */
    predictedClass = imsls_f_naive_bayes_classification(
        nb_classifier, n_patterns,
        IMSLS_CONTINUOUS, continuous,
        IMSLS_PREDICTED_CLASS_PROB,
        &pred_class_prob, 0);
    printf("    PROBABILITIES FOR INCORRECT CLASSIFICATIONS\n", dashes);
    printf("\nTRAINING PATTERNS| PREDICTED\t|\n");
    printf(" X1 X2 X3 X4 | CLASS\t| CLASS\tP(0) P(1) P(2)|\n");

    printf("%s|\n", dashes);
    for(i=0; i<n_patterns; i++){
        if(classification[i] == predictedClass[i]) continue;
        printf(" %4.1f%4.1f%4.1f%4.1f| %s\t| %s\t%4.2f %4.2f %4.2f|\n",
            continuous[i*n_continuous], continuous[i*n_continuous+1],
            continuous[i*n_continuous+2], continuous[i*n_continuous+3],
            classLabel[classification[i]], classLabel[predictedClass[i]],
            pred_class_prob[i*n_classes], pred_class_prob[i*n_classes+1],
            pred_class_prob[i*n_classes+2]);
    }
    printf("%s|\n", dashes);
    imsls_f_nb_classifier_free(nb_classifier);
}

```

Output

For Fisher's data, the Naive Bayes classifier incorrectly classified 6 of the 150 training patterns.

```

    Iris Classification Error Rates
-----
    Setosa Versicolour Virginica | TOTAL
    0/50      3/50      3/50      | 6/150
-----

    PROBABILITIES FOR INCORRECT CLASSIFICATIONS

TRAINING PATTERNS| PREDICTED |
X1  X2  X3  X4 | CLASS      | CLASS      P(0) P(1) P(2)|
-----|-----|

```

6.9	3.1	4.9	1.5	Versicolour	Virginica	0.00	0.46	0.54
5.9	3.2	4.8	1.8	Versicolour	Virginica	0.00	0.16	0.84
6.7	3.0	5.0	1.7	Versicolour	Virginica	0.00	0.08	0.92
4.9	2.5	4.5	1.7	Virginica	Versicolour	0.00	0.97	0.03
6.0	2.2	5.0	1.5	Virginica	Versicolour	0.00	0.96	0.04
6.3	2.8	5.1	1.5	Virginica	Versicolour	0.00	0.71	0.29

Example 2

This example uses the spam benchmark data available from the Knowledge Discovery Databases archive maintained at the University of California, Irvine: <http://archive.ics.uci.edu/ml/datasets/Spambase>.

These data contain of 4601 patterns consisting of 57 continuous attributes and one classification. 41% of these patterns are classified as spam and the remaining as non-spam. The first 54 continuous attributes are word or symbol percentages. That is, they are percents scaled from 0 to 100% representing the percentage of words or characters in the email that contain a particular word or character. The last three continuous attributes are word lengths. For a detailed description of these data visit the KDD archive at the above link.

In this example, percentages are transformed using the arcsin/square root transformation $y = \sin^{-1}(\sqrt{p})$. The last three attributes, word lengths, are transformed using square roots. Transformed percentages and the first word length attribute are modeled using the Gaussian distribution. The last two word lengths are modeled using the log normal distribution.

```
#include <imsls.h>
#include <stdlib.h>
#include <stdio.h>
static void printErrorRates(int classification_errors[6],
                           int n, char *label);

int main(){
    int i, j, k;
    int condPdfTableLength = 0;
    int n_patterns;
    int n_variables;
    int n_sample      = 2000;
    int n_classes     = 2;      /* spam or no spam */
    int n_continuous  = 57;
    int *classErrors  = NULL;
    int *classification = NULL;
    int classSample[2000];
    int *predictedClass = NULL;
    int *rndSampleIndex = NULL;
    int classification_errors[6];
    float *continuous, *continuousSample;
    char* label1 =
        " Trainer from Training Dataset of %d Observations \n";
    char* label2 =
        " Classifier for Entire Dataset of %d Observations \n";
    Imsls_f_nb_classifier *nb_classifier=NULL;
    float *spamData;
    int n_spam = 0;

    spamData = imsls_f_data_sets(11, IMSLS_N_OBSERVATIONS, &n_patterns,
```

```

    IMSLS_N_VARIABLES, &n_variables, 0);

    continuous      =
        (float*)malloc((n_patterns*n_continuous)*sizeof(float));
    continuousSample =
        (float*)malloc((n_sample*n_continuous)*sizeof(float));
    classification  = (int*)malloc(n_patterns*sizeof(int));

    /* map continuous attributes into transformed representation */
    for(i=0; i<n_patterns; i++){
        for(j=0; j<n_continuous; j++) {
            if (j < 54 ) {
                continuous[i*(n_variables-1)+j] = (float)
                    asin(sqrt( spamData[i*n_variables+j]/100));
            } else {
                continuous[i*(n_variables-1)+j] =
                    spamData[i*n_variables+j];
            }
        }
        classification[i] = (int)spamData[(i*n_variables)+n_variables-1];
        if(classification[i] == 1) n_spam++;
    }
    printf("Number of Patterns = %d Number Classified as Spam = %d \n\n",
        n_patterns, n_spam);

    /* select random sample for training Naive Bayes Classifier */
    imsls_random_seed_set(1234567);
    rndSampleIndex=imsls_random_sample_indices(n_sample, n_patterns, 0);
    for(k=0; k<n_sample; k++){
        i = rndSampleIndex[k]-1;
        classSample[k] = classification[i];
        for(j=0; j<n_continuous; j++) {
            continuousSample[k*n_continuous+j] =
                continuous[i*n_continuous+j];
        }
    }

    /* Train Naive Bayes Classifier */
    classErrors = imsls_f_naive_bayes_trainer(n_sample, n_classes,
        classSample,
        IMSLS_CONTINUOUS, n_continuous, continuousSample,
        IMSLS_NB_CLASSIFIER, &nb_classifier, 0);
    /* print error rates for training sample */
    printErrorRates(classErrors, n_sample, label1);

    /* CALL NAIVE_BAYES_CLASSIFICATION TO CLASSIFY ENTIRE DATASET */
    predictedClass = imsls_f_naive_bayes_classification(nb_classifier,
        n_patterns,
        IMSLS_CONTINUOUS, continuous, 0);
    /* calculate classification error rates for entire dataset */
    for(i=0; i<6; i++) classification_errors[i] = 0;
    for(i=0; i<n_patterns; i++){
        switch (classification[i])
        {
            case 0:
                classification_errors[1]++;
                if(classification[i] != predictedClass[i])
                    classification_errors[0]++;
                break;
            case 1:

```

```

        classification_errors[3]++;
        if(classification[i] != predictedClass[i])
            classification_errors[2]++;
        break;
    }
    classification_errors[5] =
        classification_errors[1]+classification_errors[3];
    classification_errors[4] =
        classification_errors[0]+classification_errors[2];
}
/* print error rates for entire dataset */
printErrorRates(classification_errors, n_patterns, label2);
}
static void printErrorRates(int classification_errors[6],
                            int n, char *label)
{
    double p, p1, p0;
    p0 = 100.0*classification_errors[0]/classification_errors[1];
    p1 = 100.0*classification_errors[2]/classification_errors[3];
    p = 100.0*classification_errors[4]/classification_errors[5];
    printf("    Classification Error Rates Reported by\n");
    printf(label, n);
    printf("-----\n");
    printf("    Not Spam      Spam      |    TOTAL\n");
    printf(" %d/%d=%4.1f%%  %d/%d=%4.1f%%  | %d/%d=%4.1f%%\n",
classification_errors[0], classification_errors[1],

        p0, classification_errors[2], classification_errors[3],
        p1, classification_errors[4], classification_errors[5], p);
    printf("-----\n\n");
    return;
}

```

Output

It is interesting to note that the classification error rates obtained by training a classifier from a random sample is slightly lower than those obtained from training a classifier with all 4601 patterns. When the classifier is trained using all 4601 patterns, the overall classification error rate was 12.9% (see [Example 3](#) for `ims1s_f_naive_bayes_trainer`). It is 12.4% for a random sample of 2000 patterns.

```
Number of Patterns = 4601 Number Classified as Spam = 1813
```

```
Classification Error Rates Reported by
Trainer from Training Dataset of 2000 Observations
```

```
-----
Not Spam      Spam      |    TOTAL
31/1202= 2.6%  218/798=27.3%  |  249/2000=12.4%
-----
```

```
Classification Error Rates Reported by
Classifier for Entire Dataset of 4601 Observations
```

```
-----
Not Spam      Spam      |    TOTAL
81/2788= 2.9%  549/1813=30.3%  |  630/4601=13.7%
-----
```

Fatal Errors

IMSLS_STOP_USER_FCN

Request from user supplied function to stop algorithm.
User flag = "#".

nb_classifier_free

Frees memory allocated to an *Imcls_f_nb_classifier* data structure.

Synopsis

```
#include <imcls.h>
```

```
void imcls_f_nb_classifier_free (Imcls_f_nb_classifier *nb_classifier)
```

The type *double* function is `imcls_d_nb_classifier_free`.

Required Arguments

Imcls_f_nb_classifier *nb_classifier (Input)

Pointer to a structure of the type *Imcls_f_nb_classifier* from `imcls_f_naive_bayes_trainer`.

Description

An *Imcls_f_nb_classifier* data structure is created by `imcls_f_naive_bayes_trainer` or `imcls_f_nb_classifier_read`. This function frees the memory allocated to this structure when it is no longer needed.

nb_classifier_write

Writes a Naive Bayes Classifier to an ASCII file for later retrieval using [imsls_f_nb_classifier_read](#).

Synopsis

```
#include <imsls.h>

void imsls_f_nb_classifier_write (imsls_f_nb_classifier *nb_classifier,
    char *filename, ..., 0)
```

The type *double* function is `imsls_d_nb_classifier_write`.

Required Arguments

imsls_f_nb_classifier *nb_classifier (Input)
A trained Naive Bayes Classifier.

char *filename (Input)
The name of an ASCII file to be created. A full or relative path can be given. If this file exists, it is replaced with the Naive Bayes Classifier. If it does not exist, it is created. If the optional argument `IMSLS_FILE` is used, `filename` is ignored.

Synopsis with Optional Arguments

```
#include <imsls.h>

void imsls_f_nb_classifier_write (imsls_f_nb_classifier *nb_classifier,
    char *filename,
    IMSLS_PRINT,
    IMSLS_FILE, FILE *file,
    0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)
Prints status of file opening, writing and closing.

Default: No printing.

IMSLS_FILE, *FILE *file* (Input/Output)

A **FILE** pointer to a file opened for writing. This file is written but not closed. If this option is provided, **filename** is ignored. This option allows users to write additional data and multiple classifiers to the same file (see [Example 2](#)). To ensure the file is opened and closed with the same C run-time library used by the product, open and close this file using [imsls_fopen](#) and [imsls_fclose](#).

Description

This function stores an *imsls_f_nb_classifier* data structure containing a trained Naive Bayes Classifier into an ASCII file. If the optional argument **IMSLS_FILE** is provided, `imsls_f_nb_classifier_write` writes the file and returns without closing the file. If this argument is not provided, `imsls_f_nb_classifier_write` creates a file using the path and name provided in **filename**, writes the data structure to that file and then closes the file before returning.

Examples

Example 1

This example trains a classifier using Fisher's Iris data. These data consist of 150 patterns. The input attributes consist of four continuous attributes and one classification attribute with three classes. The classifier is stored into four lines of an ASCII file named `NB_Classifier_Ex1.txt`.

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

int main()
{
    char *filename = "NB_Classifier_Ex1.txt";
    int i, j;
    int n_patterns = 150; /* 150 training patterns */
    int n_continuous = 4; /* four continuous input attributes */
    int n_classes = 3; /* three classification categories */
    int classification[150], *classErrors, *predictedClass;
    float *pred_class_prob, continuous[4*150] ;
    float *irisData; /* Fishers Iris Data */
    char *classLabel[3] = {"Setosa", "Versicolour", "Virginica"};
    Imsls_f_nb_classifier *nb_classifier;
    irisData = imsls_f_data_sets(3,0);

    /* setup the required input arrays from the data matrix */
    for(i=0; i<n_patterns; i++){
        classification[i] = (int) irisData[i*5]-1;
        for(j=1; j<=n_continuous; j++)
            continuous[i*n_continuous+j-1] = irisData[i*5+j];
    }
}
```

```

    }

    classErrors = imsls_f_naive_bayes_trainer(n_patterns,
        n_classes, classification,
        IMSLS_CONTINUOUS, n_continuous, continuous,
        IMSLS_NB_CLASSIFIER, &nb_classifier, 0);

    printf("    Iris Classification Error Rates\n");
    printf("-----\n");
    printf("    Setosa Versicolour Virginica |    TOTAL\n");
    printf("    %d/%d    %d/%d    %d/%d    |    %d/%d\n",
        classErrors[0], classErrors[1],
        classErrors[2], classErrors[3],
        classErrors[4], classErrors[5],
        classErrors[6], classErrors[7]);
    printf("-----\n\n");

    imsls_f_nb_classifier_write(nb_classifier, filename,
        IMSLS_PRINT, 0);
}

```

Output

```

=====

    Iris Classification Error Rates
-----
    Setosa Versicolour Virginica |    TOTAL
    0/50    3/50    3/50    |    6/150
-----

Opening NB_Classifier_Ex1.txt for writing Naive Bayes data structure
Writing Naive Bayes data structure... 5 Lines written.
File NB_Classifier_Ex1.txt closed

```

Example 2

This example illustrates the use of the optional argument `IMSLS_FILE` to store multiple classifiers into one file. Two Naive Bayes classifiers are trained using Fisher's Iris data. These data consist of 150 patterns. The input attributes consist of four continuous attributes and one classification attribute. The first classifier is trained using all four inputs and the second using only the first two. The networks are stored into 10 lines of an ASCII file named `NB_Classifier_Ex2.txt`.

```

#include <imsls.h>
#include <stdio.h>
extern FILE* imsls_fopen(char* filename, char* mode);
extern void imsls_fclose(FILE* file);

int main()
{
    FILE *file;
    char *filename = "NB_Classifier_Ex2.txt";
    int i, j;

```

```

int n_patterns    =150; /* 150 training patterns          */
int n_cont4      =4;  /* four continuous input attributes */
int n_cont2      =2;  /* two continuous input attributes  */
int n_classes    =3;  /* three classification categories  */
int n_classifiers=2;  /* two classifiers in this example  */
int classification[150], *classErrors;
float cont4[4*150], cont2[2*150];
float *irisData; /* Fishers Iris Data */
char *classLabel[3] = {"Setosa", "Versicolour", "Virginica"};
Imsls_f_nb_classifier *nb_classifier;
irisData = imsls_f_data_sets(3,0);

/* setup the required input arrays from the data matrix */
for(i=0; i<n_patterns; i++){
    classification[i] = (int) irisData[i*5]-1;
    for(j=1; j<=n_cont4; j++) {
        cont4[i*n_cont4+j-1] = irisData[i*5+j];
        if(j<3) cont2[i*n_cont2+j-1] = irisData[i*5+j];
    }
}
printf("Opening file %s\n\n", filename);
file = imsls_fopen(filename, "w");
fprintf(file, "%d\n", 2);
classErrors = imsls_f_naive_bayes_trainer(n_patterns,
    n_classes, classification,
    IMSLS_CONTINUOUS, n_cont4, cont4,
    IMSLS_NB_CLASSIFIER, &nb_classifier, 0);

printf("Iris Classification Error Rates - Classifier 1\n");
printf("-----\n");
printf("  Setosa  Versicolour  Virginica  |  TOTAL\n");
printf("  %d/%d    %d/%d    %d/%d    |  %d/%d\n",
    classErrors[0], classErrors[1],
    classErrors[2], classErrors[3],
    classErrors[4], classErrors[5],
    classErrors[6], classErrors[7]);
printf("-----\n\n");
imsls_free(classErrors);
/* write first classifier */
imsls_f_nb_classifier_write(nb_classifier, NULL, IMSLS_PRINT,
    IMSLS_FILE, file, 0);

imsls_f_nb_classifier_free(nb_classifier);
classErrors = imsls_f_naive_bayes_trainer(n_patterns,
    n_classes, classification,
    IMSLS_CONTINUOUS, n_cont2, cont2,
    IMSLS_NB_CLASSIFIER, &nb_classifier, 0);
printf("Iris Classification Error Rates - Classifier 2\n");
printf("-----\n");
printf("  Setosa  Versicolour  Virginica  |  TOTAL\n");
printf("  %d/%d    %d/%d    %d/%d    |  %d/%d\n",
    classErrors[0], classErrors[1],
    classErrors[2], classErrors[3],
    classErrors[4], classErrors[5],
    classErrors[6], classErrors[7]);
printf("-----\n\n");
imsls_f_nb_classifier_write(nb_classifier, NULL, IMSLS_PRINT,
    IMSLS_FILE, file, 0);
imsls_free(classErrors);
printf("Closing Classifier File\n");

```

```
    imsls_fclose(file);  
    return;  
}
```

Output

```
Opening file NB_Classifier_Ex2.txt
```

```
Iris Classification Error Rates - Classifier 1
```

```
-----  
Setosa Versicolour Virginica | TOTAL  
0/50      3/50      3/50      | 6/150  
-----
```

```
Writing Naive Bayes data structure to file stream.
```

```
Writing Naive Bayes data structure... 5 Lines written to file.
```

```
File not closed.
```

```
Iris Classification Error Rates - Classifier 2
```

```
-----  
Setosa Versicolour Virginica | TOTAL  
1/50      13/50     19/50     | 33/150  
-----
```

```
Writing Naive Bayes data structure to file stream.
```

```
Writing Naive Bayes data structure... 5 Lines written to file.
```

```
File not closed.
```

```
Closing Classifier File
```

Fatal Errors

IMSLS_FILE_OPEN_FAILURE

Unable to open file for writing network.

nb_classifier_read

Retrieves a Naive Bayes Classifier previously filed using [imsls_f_nb_classifier_write](#).

Synopsis

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

```
Imsls_f_nb_classifier *imsls_f_nb_classifier_read(char *filename, ..., 0)
```

The type *double* function is `imsls_d_nb_classifier_read`.

Required Arguments

*char *filename* (Input)

The name of an ASCII file containing a Naive Bayes Classifier previously saved using [imsls_f_nb_classifier_write](#). A full or relative path can be given. If the optional argument `IMSLS_FILE` is used, *filename* is ignored.

Return Value

A pointer to an *Imsls_f_nb_classifier* data structure containing a Naive Bayes Classifier previously stored using [imsls_f_nb_classifier_write](#).

Synopsis with Optional Arguments

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

```
Imsls_f_nb_classifier *imsls_f_nb_classifier_read(char *filename,  
  
IMSLS_PRINT,  
  
IMSLS_FILE, FILE *file,  
  
0)
```

Optional Arguments

IMSLS_PRINT, (Input)

Prints status of file opening, reading and closing.

Default: No printing.

IMSLS_FILE, *FILE *file* (Input)

A **FILE** pointer to a file opened for reading. This file is read but not closed. If this option is provided, **filename** is ignored. This argument allows users to read additional user-defined data and multiple classifiers from the same file (see [Example 2](#) below). To ensure the file is opened and closed with the same C run-time library used by the product, open and close this file using [imsls_fopen](#) and [imsls_fclose](#).

Description

Function `nb_classifier_read` reads a classifier from an ASCII file previously stored using [imsls_f_nb_classifier_write](#) and returns a Naive Bayes Classifier in the form of an *imsls_f_nb_classifier* data structure. If the optional argument **IMSLS_FILE** is provided, a classifier is read from the file and returned without closing the file. If this argument is not provided, `imsls_f_nb_classifier_read` opens the file using the path and name provided in **filename**, reads the classifier then closes the file and returns the data structure.

Examples

Example 1

This example reads a classifier previously trained using Fisher's Iris data (see [Example 2](#) of [imsls_f_nb_classifier_write](#)). These data consist of 150 patterns, each with four continuous attributes and one dependent variable. The classifier is read from an ASCII file named `NB_Classifier_Ex1.txt`.

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

int main()
{
    char *filename = "NB_Classifier_Ex1.txt";
    int i, j;
    int n_patterns    =150; /* 150 training patterns          */
    int n_continuous  =4;  /* four continuous input attributes */
    int n_classes     =3;  /* three classification categories  */
    int classification[150], *predictedClass;
    int classErrors[8];
    float continuous[4*150] ;
    float *irisData;      /* Fishers Iris Data */
}
```

```

char *classLabel[3] = {"Setosa      ", "Versicolour", "Virginica  "};
Imsls_f_nb_classifier *nb_classifier;
irisData = imsls_f_data_sets(3,0);

/* setup the required input arrays from the data matrix */
for(i=0; i<n_patterns; i++){
    classification[i] = (int) irisData[i*5]-1;
    for(j=1; j<=n_continuous; j++){
        continuous[i*n_continuous+j-1] = irisData[i*5+j];
    }
}
nb_classifier = imsls_f_nb_classifier_read(filename, IMSLS_PRINT,0);
predictedClass = imsls_f_naive_bayes_classification(nb_classifier,
    n_patterns, IMSLS_CONTINUOUS, continuous, 0);
for(i=0; i<6; i++) classErrors[i] = 0;
for(i=0; i<n_patterns; i++){
    switch (classification[i])
    {
    case 0:
        classErrors[1]++;
        if(classification[i] != predictedClass[i])
            classErrors[0]++;
        break;
    case 1:
        classErrors[3]++;
        if(classification[i] != predictedClass[i])
            classErrors[2]++;
        break;
    case 2:
        classErrors[5]++;
        if(classification[i] != predictedClass[i])
            classErrors[4]++;
        break;
    }
}
classErrors[6] = classErrors[0]+classErrors[2]+classErrors[4];
classErrors[7] = classErrors[1]+classErrors[3]+classErrors[5];
printf("      Iris Classification Error Rates\n");
printf("-----\n");
printf("   Setosa  Versicolour  Virginica  |  TOTAL\n");
printf("   %d/%d    %d/%d        %d/%d    |  %d/%d\n",
    classErrors[0], classErrors[1],
    classErrors[2], classErrors[3],
    classErrors[4], classErrors[5],
    classErrors[6], classErrors[7]);
printf("-----\n\n");
return;
}

```

Output`

```

Attempting to open NB_Classifier_Ex1.txt
for reading Naive Bayes data structure
File NB_Classifier_Ex1.txt Successfully Opened
File NB_Classifier_Ex1.txt closed
      Iris Classification Error Rates

```

```

-----
   Setosa  Versicolour  Virginica  |  TOTAL
   0/50    3/50        3/50    |  6/150

```

Example 2

This example illustrates the use of the optional argument `IMSL5_FILE` to read multiple classifiers stored previously into a single file using `imsls_f_nb_classifier_write` (see [Example 2 of `imsls_f_nb_classifier_write`](#)). Two Naive Bayes classifiers were trained using Fisher's Iris data. These data consist of 150 patterns. The input attributes consist of four continuous attributes and one classification attribute with three classes. The first classifier was trained using all four inputs and the second using only the first two. The classifiers are read from an ASCII file named `NB_Classifier_Ex2.txt`.

```
#include <imsls.h>
#include <stdio.h>
extern FILE* imsls_fopen(char* filename, char* mode);
extern void imsls_fclose(FILE* file);

int main()
{
    FILE *file;
    char *filename = "NB_Classifier_Ex2.txt";
    int i, j;
    int n_patterns    =150; /* 150 training patterns          */
    int n_cont4       =4;  /* four continuous input attributes */
    int n_cont2       =2;  /* two continuous input attributes  */
    int n_classes     =3;  /* three classification categories  */
    int n_classifiers =0;  /* number of classifiers            */
    int classification[150], *predictedClass;
    int classErrors[8];
    float cont4[4*150], cont2[2*150] ;
    float *irisData; /* Fishers Iris Data */
    char *classLabel[3] = {"Setosa", "Versicolour", "Virginica"};
    Imsls_f_nb_classifier *nb_classifier4, *nb_classifier2;
    irisData = imsls_f_data_sets(3,0);

    /* setup the required input arrays from the data matrix */
    for(i=0; i<n_patterns; i++){
        classification[i] = (int) irisData[i*5]-1;
        for(j=1; j<=n_cont4; j++) {
            cont4[i*n_cont4+j-1] = irisData[i*5+j];
            if(j<3) cont2[i*n_cont2+j-1] = irisData[i*5+j];
        }
    }
    printf("Opening file %s\n\n", filename);
    file = imsls_fopen(filename, "r");
    fscanf(file, "%d", &n_classifiers);

    nb_classifier4 = imsls_f_nb_classifier_read(" ", IMSLS_PRINT,
        IMSLS_FILE, file, 0);
    predictedClass = imsls_f_naive_bayes_classification(nb_classifier4,
        n_patterns, IMSLS_CONTINUOUS, cont4, 0);
    for(i=0; i<6; i++) classErrors[i] = 0;
    for(i=0; i<n_patterns; i++){
        switch (classification[i])
        {
            case 0:
```

```

        classErrors[1]++;
        if(classification[i] != predictedClass[i])
            classErrors[0]++;
        break;
    case 1:
        classErrors[3]++;
        if(classification[i] != predictedClass[i])
            classErrors[2]++;
        break;
    case 2:
        classErrors[5]++;
        if(classification[i] != predictedClass[i])
            classErrors[4]++;
        break;
    }
}
classErrors[6] = classErrors[0]+classErrors[2]+classErrors[4];
classErrors[7] = classErrors[1]+classErrors[3]+classErrors[5];
printf("    Iris Classification Error Rates\n");
printf("-----\n");
printf("    Setosa  Versicolour  Virginica  |  TOTAL\n");
printf("    %d/%d    %d/%d    %d/%d    |  %d/%d\n",
    classErrors[0], classErrors[1],
    classErrors[2], classErrors[3], classErrors[4], classErrors[5],
    classErrors[6], classErrors[7]);
printf("-----\n\n");
imsls_free(predictedClass);
nb_classifier2 = imsls_f_nb_classifier_read(" ", IMSLS_PRINT,
    IMSLS_FILE, file, 0);
predictedClass = imsls_f_naive_bayes_classification(nb_classifier2,
    n_patterns,
    IMSLS_CONTINUOUS, cont2, 0);
for(i=0; i<6; i++) classErrors[i] = 0;
for(i=0; i<n_patterns; i++){
    switch (classification[i])
    {
    case 0:
        classErrors[1]++;
        if(classification[i] != predictedClass[i])
            classErrors[0]++;
        break;
    case 1:
        classErrors[3]++;
        if(classification[i] != predictedClass[i])
            classErrors[2]++;
        break;
    case 2:
        classErrors[5]++;
        if(classification[i] != predictedClass[i])
            classErrors[4]++;
        break;
    }
}
classErrors[6] = classErrors[0]+classErrors[2]+classErrors[4];
classErrors[7] = classErrors[1]+classErrors[3]+classErrors[5];
printf("    Iris Classification Error Rates\n");
printf("-----\n");
printf("    Setosa  Versicolour  Virginica  |  TOTAL\n");
printf("    %d/%d    %d/%d    %d/%d    |  %d/%d\n",
    classErrors[0], classErrors[1],

```

```
        classErrors[2], classErrors[3], classErrors[4], classErrors[5],
        classErrors[6], classErrors[7]);
    printf("-----\n\n");
    imsls_free(predictedClass);
    printf("Closing Classifier File.\n");
    imsls_fclose(file);
}
```

Output

```
Opening file NB_Classifier_Ex2.txt
```

```
Naive Bayes Classifier restored from file. File not closed.
Iris Classification Error Rates
```

```
-----
Setosa Versicolour Virginica | TOTAL
0/50    3/50    3/50    | 6/150
-----
```

```
Naive Bayes Classifier restored from file. File not closed.
Iris Classification Error Rates
```

```
-----
Setosa Versicolour Virginica | TOTAL
1/50    13/50    19/50    | 33/150
-----
```

Fatal Errors

IMSLS_FILE_OPEN_FAILURE

Unable to open file for reading neural network.

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 nominal 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.

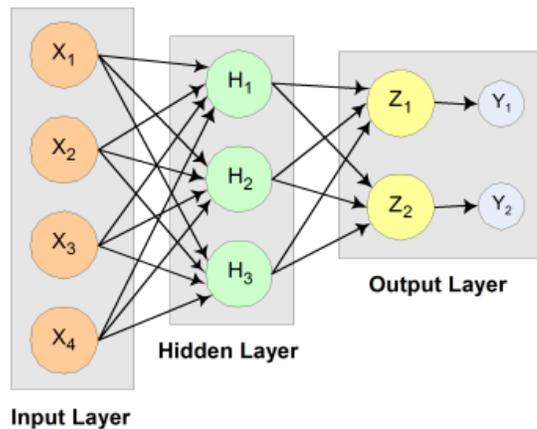


Figure 23, A 2-layer, Feedforward Network with 4 inputs and 2 outputs

Other networks, such as recurrent neural networks, allow data and information to flow in both directions, see Mandic and Chambers (2001).

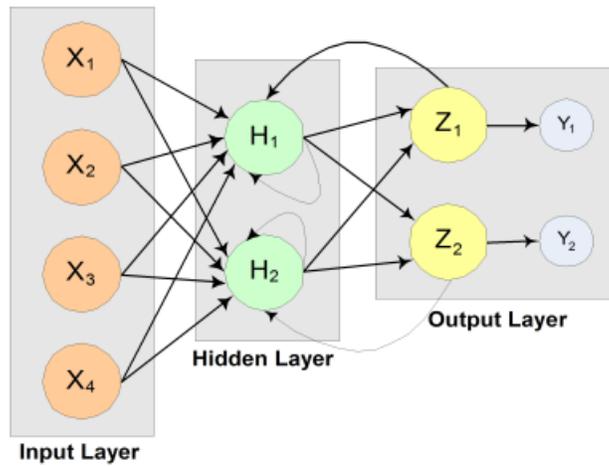


Figure 24, A recurrent neural network with 4 inputs and 2 outputs

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.

Table 49 – Synonyms between Neural Network and 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

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.

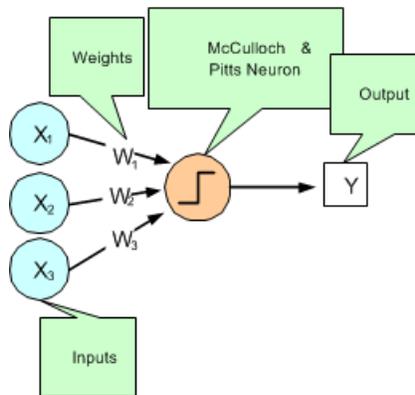


Figure 25, The McCulloch and Pitts' Threshold Neuron

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 m is the number of inputs and 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:

$$Y = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{if } Z \leq 0 \end{cases}$$

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 26](#) below.

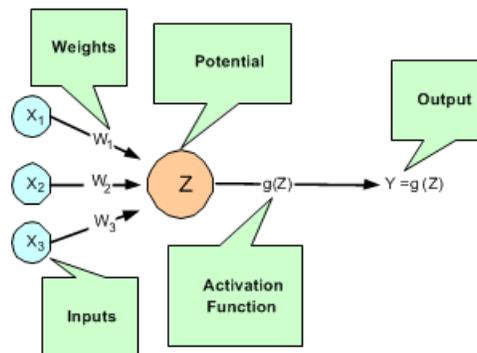


Figure 26, A Neural Net Perceptron

As shown in [Figure 26](#), 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.

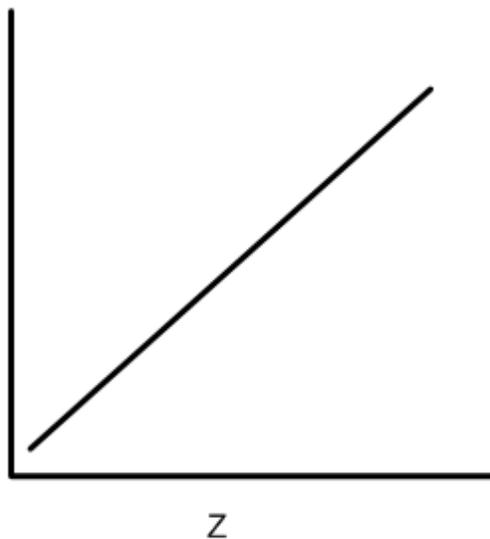


Figure 27, An Identity (Linear) Activation Function

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 $Y_i = \beta_0 + \beta_1 x_1 + \dots + \beta_k x_k$, where x_1, x_2, \dots, x_k are the k network inputs, Y_i is the i -th network output and $\beta_0, \beta_1, \dots, \beta_k$ 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., $\mathbb{R}^k \rightarrow \mathbb{R}$ where k is the number of perceptron inputs.

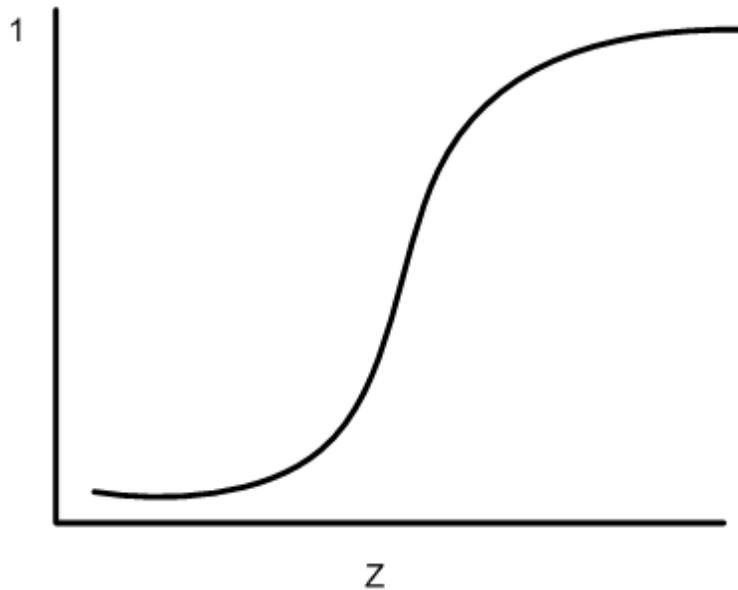


Figure 28, A Sigmoid Activation Function

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 $g(Z) = \tanh(Z) = \frac{e^{aZ} - e^{-aZ}}{e^{aZ} + e^{-aZ}}$,
- 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., $\mathfrak{R}^M \rightarrow \mathfrak{R}^C$ 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, flows 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 24](#)). 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, or hold a stock are decisions involving three 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

$$\text{softmax}_i = \frac{e^{Z_i}}{\sum_{j=1}^C e^{Z_j}}$$

The softmax activation function ensures that all outputs conform to the requirements for multivariate probabilities. That is, $0 < \text{softmax}_i < 1$, for all $i = 1, 2, \dots, C$ 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.

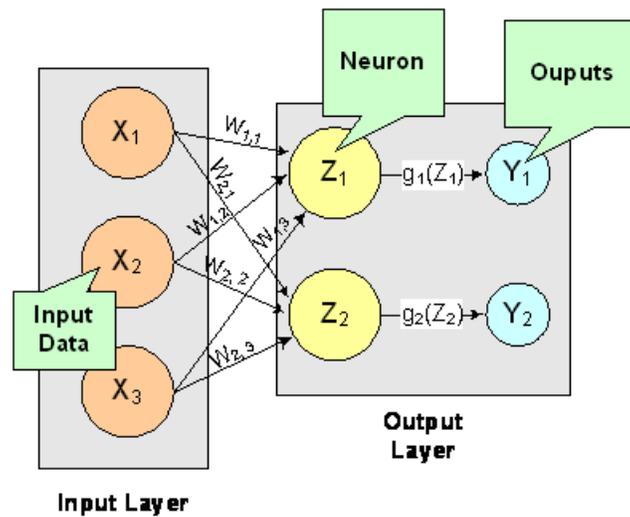


Figure 29, A Single-Layer Feedforward Neural Net

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

$$Z_1 = \sum_{i=1}^3 W_{1,i}x_i + \mu_1 \text{ and } Z_2 = \sum_{i=1}^3 W_{2,i}x_i + \mu_2$$

$$Y_1 = g_1(Z_1) = g_1\left(\sum_{i=1}^3 W_{1,i}x_i + \mu_1\right)$$

and

$$Y_2 = g_2(Z_2) = g_2\left(\sum_{i=1}^3 W_{2,i}x_i + \mu_2\right)$$

When the activation functions g_1 and g_2 are identity activation functions, the single-layer neural network is equivalent to a linear regression model. Similarly, if g_1 and g_2 are logistic activation functions, then the single-layer neural network 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:

$$E = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^C (t_{ij} - \hat{t}_{ij})^2$$

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:

$$E^{RMS} = \frac{\sum_{i=1}^N \sum_{j=1}^C (t_{ij} - \hat{t}_{ij})^2}{\sum_{i=1}^N \sum_{j=1}^C (t_{ij} - \bar{t})^2}$$

where \bar{t} is the average output:

$$\bar{t} = \frac{\sum_{i=1}^N \sum_{j=1}^C t_{ij}}{NC}$$

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:

$$E^{RMS} = \frac{2}{t} E$$

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:

$$E^R = \sum_{i=1}^N \sum_{j=1}^C |t_{ij} - \hat{t}_{ij}|^R$$

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 Minkowski-R error simplifies to the sum of absolute differences:

$$L = E^1 = \sum_{i=1}^N \sum_{j=1}^C |t_{ij} - \hat{t}_{ij}|$$

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:

$$L^{RMS} = \frac{\sum_{i=1}^N \sum_{j=1}^C |t_{ij} - \hat{t}_{ij}|}{\sum_{i=1}^N \sum_{j=1}^C |t_{ij} - \bar{t}|}$$

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 vari-

able 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.:

$$t_i = \begin{cases} 1 & \text{if training pattern } i = \text{"yes"} \\ 0 & \text{if training pattern } i = \text{"no"} \end{cases}$$

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 i -th case (either 1 or 0), and \hat{t}_i is the network output for the i -th training pattern. This is equal to the neural network's estimate of the probability that the i -th 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:

$$E_{\min}^C = - \sum_{i=1}^N \{ t_i \ln(t_i) + (1 - t_i) \ln(1 - t_i) \}$$

Subtracting E_{\min}^C from E^C gives an error term bounded below by zero, i.e.,

$$E^{CE} \geq 0$$

where

$$E^{CE} = E^C - E_{\min}^C = - \sum_{i=1}^N \left\{ t_i \ln\left(\frac{\hat{t}_i}{t_i}\right) + (1 - t_i) \ln\left(\frac{1 - \hat{t}_i}{1 - t_i}\right) \right\}$$

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 j -th 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., $0 \leq \hat{t}_{ij} \leq 1$.

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. Yet, 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 j -th category if and only if $\hat{t}_{ij} > \delta$ and $\hat{t}_{ik} \leq \delta$ for all $k \neq j$, where δ is the threshold. If this does not occur, then the pattern is marked as *unclassified*.

Another approach to multiclass 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

$$\sum_{j=1}^C \hat{t}_{ij} = 1 \text{ for all } i = 1, 2, \dots, N \text{ and } 0 \leq \hat{t}_{ij} \leq 1 \text{ for all } i = 1, 2, \dots, N$$

and

$$j = 1, 2, \dots, C$$

The softmax activation function estimates classification probabilities using the following softmax activation function:

$$\hat{t}_{ij} = \frac{e^{Z_{ij}}}{\sum_{j=1}^C e^{Z_{ij}}}$$

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:

$$0 \leq \hat{t}_{ij} \leq 1 \text{ for all } i = 1, 2, \dots, N, j = 1, 2, \dots, C \text{ and}$$

$$\sum_{j=1}^C \hat{t}_{ij} = 1 \text{ for all } i = 1, 2, \dots, N$$

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:

$$E_{\min}^C = - \sum_{i=1}^N \sum_{j=1}^C t_{ij} \ln(t_{ij})$$

Subtracting this from E^C gives an error term bounded below by zero, i.e., $E^{CE} \geq 0$ where:

$$E^{CE} = E^C - E_{\min}^C = - \sum_{i=1}^N \sum_{j=1}^C t_{ij} \ln\left(\frac{\hat{t}_{ij}}{t_{ij}}\right)$$

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:

$$E = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^C (t_{ij} - \hat{t}_{ij})^2$$

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. Function `imspls_f_mlff_network_trainer` Stage I training is implemented using Quasi-Newton optimization and steepest ascent with gradients estimated using the back-propagation method. Stage II training is implemented using Quasi-Newton optimization.

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:

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k$$

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, $g'(a_j)$, varies among these functions. See the following table:

Table 50: Activation Functions and Their Derivatives

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	$g(a) = \tanh(a)$	$g'(a) = \operatorname{sech}^2(a) = 1 - \tanh^2(a)$
Squash	$g(a) = \frac{a}{1+ a }$	$g'(a) = \frac{1}{(1+ a)^2} = (1- g(a))^2$

mlff_network_init

Initializes an *Imsls_f_NN_Network* data structure for use in training a neural network.

Synopsis

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

```
Imsls_f_NN_Network *imsls_f_mlff_network_init (int n_inputs, int n_outputs)
```

The type *double* function is `imsls_d_mlff_network_init`.

Required Arguments

int n_inputs (Input)

The number of network inputs. If the network uses nominal input attributes, the number of inputs equals the number of encoded columns used to represent these attributes plus the number of continuous and ordinal input attributes, if any.

int n_outputs (Input)

The number of network outputs. For neural networks used for forecasting continuous responses, `n_outputs` is equal to the number of variables being forecasted. Networks used for binary classification have only one output. Other classification networks have one output for every possible target category.

Return Value

An *Imsls_f_NN_Network* data structure initialized with the number of inputs and outputs specified by `n_inputs` and `n_outputs`. To release this space use [imsls_f_mlff_network_free](#).

Description

The function `imsls_f_mlff_network_init` is used to initialize the network, the function [imsls_f_mlff_network](#) is used to build up the network in preparation for training, and the function [imsls_f_mlff_network_free](#) is used to free the internally allocated structure.

Function `imsls_f_mlff_network_init` initializes and returns an *Imsls_f_NN_Network* data structure. This structure is required input to `imsls_f_mlff_network` and the network trainers. This function initializes the structure to accommodate a network with the number of inputs and outputs specified by `n_inputs` and `n_outputs` respectively. This function must be called prior to building the complete network architecture using `imsls_f_mlff_network`.

Function `imsls_f_mlff_network` modifies the structure initialized by this function and builds the network architecture consisting of hidden layers, perceptrons and links among these objects. This architecture is independent of the training data. Once the architecture is complete, the *Imsls_f_NN_Network* data structure can be used with `imsls_f_mlff_network_trainer` or `imsls_f_classification_trainer` to train the network.

After the network is trained, not only does the *Imsls_f_NN_Network* data structure contains a description of the network architecture, it also contains the network weights needed for neural network forecasting or classification.

Example

For details, see the [Examples](#) section of `mlff_network`.

mlff_network

Creates a multilayered feedforward neural network.

Synopsis

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

```
void imsls_f_mlff_network (Imsls_f_NN_Network *network, ..., 0)
```

The type *double* functions is `imsls_d_mlff_network`.

Required Arguments

Imsls_f_NN_Network *network (Input/Output)

A pointer to the structure containing the neural network that was initialized by *imsls_f_mlff_network_init*. On output, the data structure will be updated depending on the optional arguments used.

Synopsis with Optional Arguments

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

```
void imsls_f_mlff_network (Imsls_f_NN_Network *network,  
    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, or  
    IMSLS_LINK_LAYER, int to, int from, or  
    IMSLS_LINK_NODE, int to, int from, or  
    IMSLS_REMOVE_LINK, int to, int from,  
    IMSLS_N_LINKS, int *n_links,  
    IMSLS_DISPLAY_NETWORK,
```

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:

Activation Function	Description
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`.

or

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_N_LINKS, *int *n_links* (Output)

Returns the number of links in the network.

IMSLS_DISPLAY_NETWORK (Input)

Displays the contents of the network structure.

Default: No printing is done.

Description

A multilayered 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`. 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 feedforward 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 perceptron 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 `Imsls_f_NN_Network`:

```
typedef struct
{
    int          n_inputs;
    int          n_outputs;
    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;          /* An array containing the indices into the
                                Node array that belong to this layer */
} Imsls_NN_Layer;
```

`Imsls_NN_Link` is:

```
typedef struct
{
    float        weight;
    float        gradient;
    int          to_node; /* index of to node */
    int          from_node; /* index of from node */
} Imsls_f_NN_Link;
```

and, *Imsls_NN_Node* is:

```
typedef struct
{
    int      layer_id;
    int      n_inLinks;
    int      n_outLinks;
    int      *inLinks; /* index to Links array */
    int      *outLinks; /* index to Links array */
    float    gradient;
    float    bias;
    int      ActivationFcn;
} Imsls_f_NN_Node;
```

In particular, if `network` is a pointer to the structure of type *Imsls_f_NN_Network*, then:

Table 51: Structure Members and Their Descriptions

Structure member	Description
<code>network->n_layers</code>	Number of layers in network. Layers are numbered starting at 0 for the input layer.
<code>network->n_nodes</code>	Total number of nodes in network, including the input attributes.
<code>network->n_links</code>	Total number of links or connections between input attributes and perceptrons and between perceptrons from layer to layer.
<code>network->layers[0]</code>	Input layer with <code>n_inputs</code> attributes.
<code>network->layers[network->n_layers-1]</code>	Output layer with <code>n_outputs</code> perceptrons.
<code>network->n_inputs</code> which is equal to <code>network->layers[0].n_nodes</code>	<code>n_inputs</code> (number of input attributes).
<code>network->n_outputs</code> which is equal to <code>network->layers[network->n_layers-1].n_nodes</code>	<code>n_outputs</code> (number of output perceptrons).
<code>network->layers[1].n_nodes</code>	Number of perceptrons in first hidden layer, or number of output perceptrons if no hidden layer.
<code>network->links[i].weight</code>	Initial weight for the <i>i</i> -th link in network. After the training has completed the structure member contains the weight used for forecasting.
<code>network->nodes[i].bias</code>	Initial bias value for the <i>i</i> -th node. After the training has completed the bias value is updated.

Nodes are numbered starting at zero with the input nodes, followed by the hidden layer perceptrons and finally the output perceptrons.

Layers are numbered starting at zero with the input layer, followed by the hidden layers and finally the output layer. If there are no hidden layers, the output layer is numbered one.

Links are numbered starting at zero in the order the links were created. If the `IMSLS_LINK_ALL` option was used, the first link is the input link from the first input node to the first node in the next layer. The second link is the input link from the first input node to the second node in the next layer, continuing to the link from the last node in the next to last layer to the last node in the output layer. However, due to the possible variations in the order the links may be created, it is advised to initialize the weights using the `imsls_f_initialize_weights` routine or use the optional argument `IMSLS_WEIGHT_INITIALIZATION_METHOD` in functions `imsls_f_mlff_network_trainer` and `imsls_f_mlff_classification_trainer`. Alternatively, the weights can be initialized in the `ImSls_f_NN_Network` data structure. The following code is an example of how to initialize the network weights in an `ImSls_f_NN_Network` variable created with the name `network`:

```

for (j=network->n_inputs; j < network->n_nodes; j++)
{
    for (k=0; k < network->nodes[j].n_inLinks; k++)
    {
        wIdx = network->nodes[j].inLinks[k];
        /* set specific layer weights */
        if (network->nodes[j].layer_id == 1) {
            network->links[wIdx].weight = 0.5;
        } else if (network->nodes[j].layer_id == 2) {
            network->links[wIdx].weight = 0.33;
        } else {
            network->links[wIdx].weight = 0.25;
        }
    }
}

```

The first for loop, `j` iterates through each perceptron in the network. Since input nodes are not perceptrons, they are excluded. The second for loop, `k` iterates through each of the perceptron's input links, `network->nodes[j].inLinks[k]`. `network->nodes[j].n_inLinks` is the number of input links for `network->nodes[j]`. `network->nodes[j].inLinks[k]` contains the index of each input link to `network->nodes[j]` in the `network->links` array.

This example also illustrates how to set the weights based on the `layer_id` number.

`network->nodes[j].layer_id` contains the layer identification number. This is used to set the weights for the first hidden layer to 0.5, the second hidden layer weights to 0.33 and all others to 0.25.

Examples

Example 1

This example creates a single-layer feedforward network. The network inputs are directly connected to the output perceptrons using the `IMSLS_LINK_ALL` argument. The output perceptrons use the default linear activation function and default bias values of 0.0. The `IMSLS_DISPLAY_NETWORK` argument is used to show the default settings of the network.

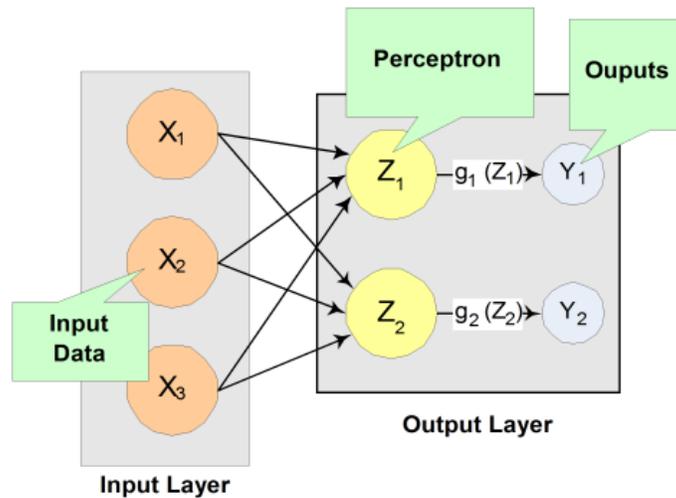


Figure 30, A Single-Layer Feedforward Neural Net

```
#include <imsls.h>
int main()
{
    Imsls_f_NN_Network *network;

    network = imsls_f_mlff_network_init(3,2);
    imsls_f_mlff_network(network,
        IMSLS_LINK_ALL,
        IMSLS_DISPLAY_NETWORK,
        0);

    imsls_f_mlff_network_free(network);
}
```

Output

```

+++++++
Input Layer
-----
    NODE_0
Activation Fcn = 0
Bias = 0.000000
  Output Links : 0 1
    NODE_1
Activation Fcn = 0
Bias = 0.000000
  Output Links : 2 3
    NODE_2
Activation Fcn = 0
Bias = 0.000000
  Output Links : 4 5

Output Layer
-----
    NODE_3
Activation Fcn = 0
Bias = 0.000000
  Input Links : 0 2 4
    NODE_4
Activation Fcn = 0
Bias = 0.000000
  Input Links : 1 3 5

***** Links *****
network->links[0].weight = 0.00000000000000000000
network->links[0].gradient = 1.00000000000000000000
network->links[0].to_node = 3
network->links[0].from_node = 0
network->links[1].weight = 0.00000000000000000000
network->links[1].gradient = 1.00000000000000000000
network->links[1].to_node = 4
network->links[1].from_node = 0
network->links[2].weight = 0.00000000000000000000
network->links[2].gradient = 1.00000000000000000000
network->links[2].to_node = 3
network->links[2].from_node = 1
network->links[3].weight = 0.00000000000000000000
network->links[3].gradient = 1.00000000000000000000
network->links[3].to_node = 4
network->links[3].from_node = 1
network->links[4].weight = 0.00000000000000000000
network->links[4].gradient = 1.00000000000000000000
network->links[4].to_node = 3
network->links[4].from_node = 2
network->links[5].weight = 0.00000000000000000000
network->links[5].gradient = 1.00000000000000000000
network->links[5].to_node = 4
network->links[5].from_node = 2

```

Example 2

This example 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 example demonstrates how to change the activation function and bias values for hidden and output layer perceptrons as shown in [Figure 31](#) below.

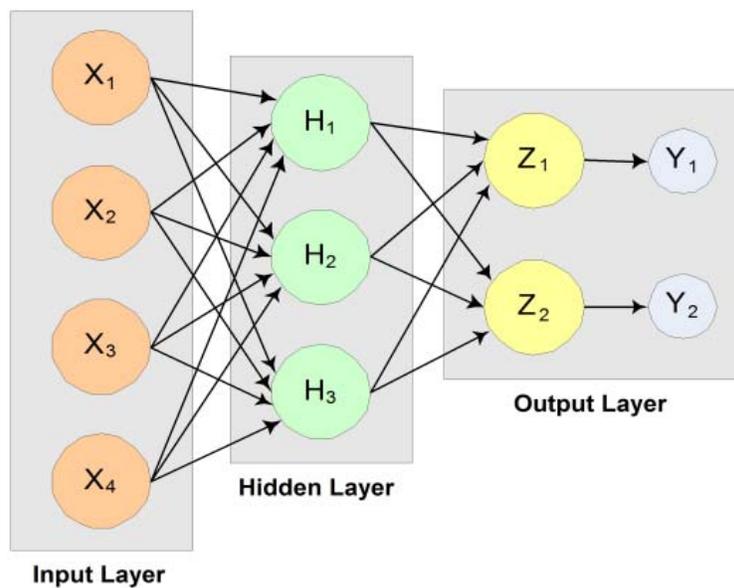


Figure 31, A 2-layer Feedforward Network with 4 Inputs and 2 Outputs

```
#include <imsls.h>

int main()
{
    Imsls_f_NN_Network *network;
    int hidActFcn[3] = {IMSLS_LINEAR, IMSLS_LINEAR, IMSLS_LINEAR};
    float outbias[2] = {1.0, 1.0};
    float hidbias[3] = {1.0, 1.0, 1.0};

    network = imsls_f_mlff_network_init(4,2);
    imsls_f_mlff_network(network,
        IMSLS_CREATE_HIDDEN_LAYER, 3,
        IMSLS_ACTIVATION_FCN, 1, &hidActFcn,
        IMSLS_BIAS, 2, outbias,
        IMSLS_LINK_ALL,
        0);
}
```

```

imsls_f_mlff_network(network,
    IMSLS_BIAS, 1, hidbias,
    0);

imsls_f_mlff_network_free(network);
}

```

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:

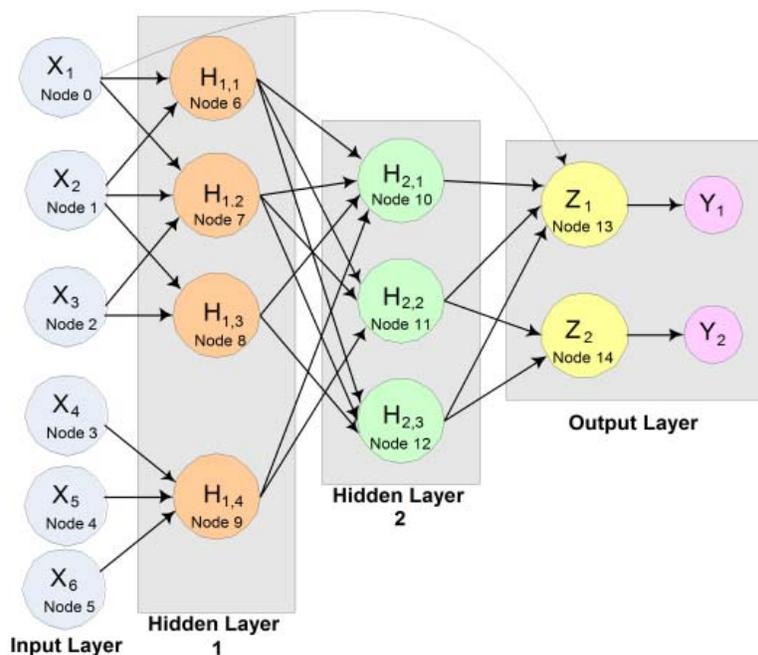


Figure 32, A network that uses a total of nine perceptrons to produce two forecasts from six input attributes

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 to construct individual links, or, an alternative approach is to first create all links and then remove those that are not needed. This example illustrates the latter approach.

```
#include <imspls.h>
int main()
{
    Imspls_f_NN_Network *network;

    network = imspl_f_mlff_network_init(6,2);
    /* Create 2 hidden layers and link all nodes */
    imspl_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 4, 0);
    imspl_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 3,
        IMSLS_LINK_ALL, 0);
    /* Remove unwanted links from Input 1 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,8,0, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,9,0, 0);
    /* Remove unwanted links from Input 2 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,9,1, 0);
    /* Remove unwanted links from Input 3 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,6,2, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,9,2, 0);
    /* Remove unwanted links from Input 4 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,6,3, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,7,3, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,8,3, 0);
    /* Remove unwanted links from Input 5 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,6,4, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,7,4, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,8,4, 0);
    /* Remove unwanted links from Input 6 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,6,5, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,7,5, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,8,5, 0);
    /* Add link from Input 1 to Output Perceptron 1 */
    imspl_f_mlff_network(network, IMSLS_LINK_NODE,13,0, 0);

    /* Remove unwanted links between hidden Layer 1 and hidden layer 2 */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,11,8, 0);
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,12,9, 0);

    /* Remove unwanted links between hidden Layer 2 and output layer */
    imspl_f_mlff_network(network, IMSLS_REMOVE_LINK,14,10, 0);

    imspl_f_mlff_network_free(network);
}
```

Another approach is to use keywords `IMSLS_LINK_NODE` and `IMSLS_LINK_LAYER` to combine links between the two hidden layers, create individual links, and remove the links that are not needed. This example illustrates this approach:

```
#include <imspls.h>
int main()
{
    Imspls_f_NN_Network *network;
```

```
network = imsls_f_mlff_network_init(6,2);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 4, 0);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 3, 0);

/* Link input attributes to first hidden layer */
imsls_f_mlff_network(network, IMSLS_LINK_NODE,6,0, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,7,0, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,6,1, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,7,1, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,8,1, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,7,2, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,8,2, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,9,3, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,9,4, 0);
imsls_f_mlff_network(network, IMSLS_LINK_NODE,9,5, 0);

/* Link hidden layer 1 to hidden layer 2 then remove unwanted links */
imsls_f_mlff_network(network, IMSLS_LINK_LAYER,2,1, 0);
imsls_f_mlff_network(network, IMSLS_REMOVE_LINK,11,8, 0);
imsls_f_mlff_network(network, IMSLS_REMOVE_LINK,12,9, 0);

/* Link hidden layer 2 to output layer then remove unwanted link */
imsls_f_mlff_network(network, IMSLS_LINK_LAYER,3,2, 0);
imsls_f_mlff_network(network, IMSLS_REMOVE_LINK,14,10, 0);

imsls_f_mlff_network_free(network);
}
```

mlff_network_free

Frees memory allocated to an *Imsls_f_NN_Network* data structure.

Synopsis

```
#include <imsls.h>

void imsls_f_mlff_network_free (Imsls_f_NN_Network *network)
```

The type *double* function is `imsls_d_mlff_network_free`.

Required Arguments

Imsls_f_NN_Network ***network** (Input)
Pointer to a structure of the type *Imsls_f_NN_Network*.

Description

Function `mlff_network_free` frees memory allocated for an *Imsls_f_NN_Network* data structure created by [imsls_f_mlff_network_init](#) and [imsls_f_mlff_network](#). If it is necessary to maintain the network information contained in this structure for developing forecasts or classifications in the future, the structure can be stored to a file and retrieved later using [imsls_f_mlff_network_write](#) and [imsls_f_mlff_network_read](#), respectively.

Example

For details, see the [Examples](#) section of `mlff_network`.

mlff_network_write

Writes a trained neural network to an ASCII file for later retrieval using `imsls_f_mlff_network_read`.

Synopsis

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

```
void imsls_f_mlff_network_write(Imsls_f_NN_Network *network, char *filename, ..., 0)
```

The type *double* function is `imsls_d_mlff_network_write`.

Required Arguments

Imsls_f_NN_Network *network (Input)

A trained neural network.

char *filename (Input)

The name of an ASCII file to be created. A complete or relative path can be used. If this file exists, it is replaced with a description of the neural network. If it does not exist, it is created. If the optional argument `IMSLS_FILE` is used, `filename` is ignored.

Synopsis with Optional Arguments

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

```
void imsls_f_mlff_network_write (Imsls_f_NN_Network *network,  
    charImsls_f_NN_Network*filename,  
    IMSLS_PRINT,  
    IMSLS_FILE, FILE *file,  
    0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

Prints status of file open, writing and closing.

Default: no printing.

IMSLS_FILE, *FILE *file* (Input/Output)

A **FILE** pointer to a file opened for writing. This file is written but not closed. If this option is provided, **filename** is ignored. This option allows users to read additional user-defined data and multiple networks from the same file (see [Example 2](#)). To ensure this file is opened and closed with the same C run-time library used by the product, open and close this file using [imsls_fopen](#) and [imsls_fclose](#) instead of **fopen** and **fclose**.

Description

This function stores an *Imsls_f_NN_Network* data structure containing a trained neural network into an ASCII file. If the optional argument **IMSLS_FILE** is provided, [imsls_f_mlff_network_write](#) writes the data structure and returns without closing the file. If this argument is not provided, [imsls_f_mlff_network_write](#) creates a file using the path and name provided in **filename**, writes the data structure to that file, and then closes the file before returning.

Examples

Example 1

This example trains a network using the Draper-Smith data. These data consist of 13 patterns. The input attributes consist of four continuous attributes and one dependent variable. The network is stored into 73 lines of an ASCII file named **NeuralNetworkEx1.txt**.

```
#include <imsls.h>
#include <stdio.h>
int main(){
    char *filename = "NeuralNetworkEx1.txt";
    float *trainStats;
    int i, j;
    int n_patterns      =13;
    int n_inputs        =4;
    int n_nominal       =0;
    int n_continuous    =4;
    int n_outputs       =1;
    int *nominalAtt=NULL;
    float ss;
    float continuous[4*13], y[13];
    float *draperSmithData;
    float forecasts[13];
    Imsls_f_NN_Network *networkStructure;
    draperSmithData = imsls_f_data_sets(5,0);

    for(i=0; i < n_patterns; i++){
        y[i] = draperSmithData[5*i+4];
```

```

        for(j=0; j<4; j++)
            continuous[i*4+j] = draperSmithData[5*i+j];
    }
    networkStructure = imsls_f_mlff_network_init(n_inputs, n_outputs);
    imsls_f_mlff_network(networkStructure,
        IMSLS_CREATE_HIDDEN_LAYER, 4, IMSLS_LINK_ALL, 0);
    imsls_random_seed_set(5555);
    trainStats = imsls_f_mlff_network_trainer(networkStructure,
        n_patterns, 0, n_continuous, nominalAtt,
        continuous, y, IMSLS_STAGE_I, 100, 13,
        IMSLS_FORECASTS_USER, forecasts, 0);
    printf("OBS   X1     X2     X3     X4     Y     ");
    printf("FORECAST\n");
    ss = 0;
    for(i=0; i<n_patterns; i++) {
        ss += (y[i]-forecasts[i])*(y[i]-forecasts[i]);
        printf("%2d %7.2f %7.2f %7.2f %7.2f %7.2f %7.2f\n",
            i, continuous[i*4], continuous[i*4+1], continuous[i*4+2],
            continuous[i*4+3], y[i], forecasts[i]);
    }
    printf("Sum of Squared Residuals: %7.2f\n\n", ss);
    imsls_f_mlff_network_write(networkStructure, filename,
        IMSLS_PRINT, 0);
}

```

Output

OBS	X1	X2	X3	X4	Y	FORECAST
0	7.00	26.00	6.00	60.00	78.50	78.50
1	1.00	29.00	15.00	52.00	74.30	74.30
2	11.00	56.00	8.00	20.00	104.30	104.22
3	11.00	31.00	8.00	47.00	87.60	87.60
4	7.00	52.00	6.00	33.00	95.90	95.78
5	11.00	55.00	9.00	22.00	109.20	109.34
6	3.00	71.00	17.00	6.00	102.70	102.55
7	1.00	31.00	22.00	44.00	72.50	72.50
8	2.00	54.00	18.00	22.00	93.10	93.24
9	21.00	47.00	4.00	26.00	115.90	116.05
10	1.00	40.00	23.00	34.00	83.80	83.80
11	11.00	66.00	9.00	12.00	113.30	112.30
12	10.00	68.00	8.00	12.00	109.40	110.33
Sum of Squared Residuals:					1.97	

```

Opening NeuralNetworkEx1.txt for writing network data structure
Writing Neural Network... 73 Lines written to network file.
File NeuralNetworkEx1.txt closed.

```

Example 2

This example illustrates the use of the optional argument `IMSLS_FILE` to store multiple neural networks into one file. Two networks are trained using the Draper-Smith data. These data consist of 13 patterns each with four continuous attributes and one dependent variable. The first network is trained for forecasting the dependent variable using all four attributes, and the second is trained using only the first three. The networks are stored into 133 lines of an ASCII file named `NeuralNetworkEx2.txt`.

```
#include <imsls.h>
#include <stdio.h>
extern FILE* imsls_fopen(char* filename, char* mode);
extern void imsls_fclose(FILE* file);
int main(){
    FILE *file;
    char *filename = "NeuralNetworkEx2.txt";
    float *trainStats;
    int i, j;
    int n_patterns      =13;
    int n_inputs4       =4;
    int n_inputs3       =3;
    int n_cont4         =4;
    int n_cont3         =3;
    int n_outputs       =1;
    int *categoricalAtt=NULL;
    float ss3, ss4;
    float cont4[4*13], cont3[3*13], y[13];
    float *draperSmithData;
    float forecasts3[13], forecasts4[13];
    Imsls_f_NN_Network *networkStructure;
    draperSmithData = imsls_f_data_sets(5,0);

    for(i=0; i < n_patterns; i++){
        y[i] = draperSmithData[5*i+4];
        for(j=0; j<4; j++){
            cont4[i*4+j] = draperSmithData[5*i+j];
            if(j<3) cont3[i*3+j] = draperSmithData[5*i+j];
        }
    }
    networkStructure = imsls_f_mlff_network_init(n_inputs4,
                                                n_outputs);
    imsls_f_mlff_network(networkStructure,
                        IMSLS_CREATE_HIDDEN_LAYER, 4,
                        IMSLS_LINK_ALL, 0);
    imsls_random_seed_set(5555);
    trainStats = imsls_f_mlff_network_trainer(networkStructure,
        n_patterns, 0, n_cont4,
        categoricalAtt, cont4, y,
        IMSLS_MAX_STEP, 100.0,
        IMSLS_STAGE_I, 50, n_patterns,
        IMSLS_FORECASTS_USER, &forecasts4, 0);
    /* open filestream */
    file = imsls_fopen(filename, "w");
    /* Write the number of network being placed into this file */
    fprintf(file, "%d\n", 2);
    printf("Writing network for model with 4 continuous attributes\n");
    imsls_f_mlff_network_write(networkStructure, NULL,
                              IMSLS_PRINT, IMSLS_FILE, file, 0);
```

```

/* Create second neural network */
imsls_f_mlff_network_free(networkStructure);
imsls_free(trainStats);
networkStructure = imsls_f_mlff_network_init(n_inputs3,
                                             n_outputs);

imsls_f_mlff_network(networkStructure,
                     IMSLS_CREATE_HIDDEN_LAYER, 4,
                     IMSLS_LINK_ALL, 0);
imsls_random_seed_set(5555);
trainStats = imsls_f_mlff_network_trainer(networkStructure,
     n_patterns, 0, n_cont3,
     categoricalAtt, cont3, y,
     IMSLS_MAX_STEP, 100.0,
     IMSLS_STAGE_I, 50, n_patterns,
     IMSLS_FORECASTS_USER, &forecasts3, 0);
printf("Writing network for model with 3 continuous attributes\n");
imsls_f_mlff_network_write(networkStructure, NULL, IMSLS_PRINT,
                           IMSLS_FILE, file, 0);

imsls_fclose(file);
printf("File %s Closed.\n", filename);
printf("\nPrinting Forecasts for models with 3 and 4");
printf(" continuous attributes:\n");
printf("\n                ");
printf("FORECAST FORECAST\n");
printf("OBS    X1      X2      X3      X4      Y      ");
printf("n_cont=3  n_cont=4\n");
ss4 = 0;
ss3 = 0;
for(i=0; i<n_patterns; i++) {
    ss4 += (y[i]-forecasts4[i])*(y[i]-forecasts4[i]);
    ss3 += (y[i]-forecasts3[i])*(y[i]-forecasts3[i]);
    printf("%2d %7.2f %7.2f %7.2f %7.2f %7.2f %9.2f\n",
        i, cont4[i*4], cont4[i*4+1], cont4[i*4+2],
        cont4[i*4+3], y[i], forecasts3[i], forecasts4[i]);
}
printf("Sum of Squared Residuals for X1-X3: %7.2f\n", ss3);
printf("Sum of Squared Residuals for X1-X4: %7.2f\n", ss4);
}

```

Output

```

Writing network for model with 4 continuous attributes
Writing Neural Network... 73 Lines written to network file.
File not closed.
Writing network for model with 3 continuous attributes
Writing Neural Network... 60 Lines written to network file.
File not closed.
File NeuralNetworkEx2.txt Closed.

Printing Forecasts for models with 3 and 4 continuous attributes:

```

OBS	FORECAST FORECAST					n_cont=3	n_cont=4
	X1	X2	X3	X4	Y		
0	7.00	26.00	6.00	60.00	78.50	78.97	78.50
1	1.00	29.00	15.00	52.00	74.30	73.62	74.30
2	11.00	56.00	8.00	20.00	104.30	104.61	104.30
3	11.00	31.00	8.00	47.00	87.60	87.32	87.60
4	7.00	52.00	6.00	33.00	95.90	94.88	95.90

5	11.00	55.00	9.00	22.00	109.20	108.82	109.20
6	3.00	71.00	17.00	6.00	102.70	102.69	102.70
7	1.00	31.00	22.00	44.00	72.50	72.97	72.50
8	2.00	54.00	18.00	22.00	93.10	94.28	93.10
9	21.00	47.00	4.00	26.00	115.90	115.04	115.90
10	1.00	40.00	23.00	34.00	83.80	83.64	83.80
11	11.00	66.00	9.00	12.00	113.30	114.45	113.30
12	10.00	68.00	8.00	12.00	109.40	109.20	109.40
Sum of Squared Residuals for X1-X3:						5.76	
Sum of Squared Residuals for X1-X4:						0.00	

Fatal Errors

IMSL_FILE_OPEN_FAILURE

Unable to open file for writing network.

mlff_network_read

Retrieves a neural network from a file previously saved using `imsls_f_mlff_network_write`.

Synopsis

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

```
Imsls_f_NN_Network *imsls_f_mlff_network_read(char *filename, ..., 0)
```

The type *double* function is `imsls_d_mlff_network_read`.

Required Arguments

*char *filename* (Input)

The name of an ASCII file containing a description of a trained neural network previously saved using `imsls_f_mlff_network_write`. A complete or relative path can be used. If the optional argument `IMSLS_FILE` is used, *filename* is ignored and the file is not closed before returning.

Return Value

A pointer to an *Imsls_f_NN_Network* data structure containing the neural network stored using `imsls_f_mlff_network_write`. This space can be released by using the `imsls_free` function.

Synopsis with Optional Arguments

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

```
Imsls_f_NN_Network *imsls_f_mlff_network_read(char *filename,  
IMSLS_PRINT,  
IMSLS_FILE, FILE *file,  
0)
```

Optional Arguments

`IMSLS_PRINT`, (Input)

Prints status of file open, reading and closing.

Default: No printing.

`IMSLS_FILE`, `FILE *file` (Input)

A `FILE` pointer to a file opened for reading. This file is read but not closed. If this option is provided, `filename` is ignored. This option allows users to read additional user-defined data and multiple networks from the same file (see [Example 2](#)). To ensure this file is opened and closed with the same C run-time library used by the product, open and close this file using `imsls_fopen` and `imsls_fclose` instead of `fopen` and `fclose`.

Description

This function reads an `imsls_f_NN_Network` data structure, a neural network previously stored as an ASCII file using `imsls_f_mlff_network_write`. If the optional argument `IMSLS_FILE` is provided, the data structure is read from that file stream and the file stream is not closed. If this argument is not provided, `imsls_f_mlff_network_read` opens a file using the path and name provided in `filename`, reads the data structure from that file, and then closes the file before returning.

Examples

Example 1

This example reads a network previously trained using the Draper-Smith data. These data consist of 13 patterns, each with four continuous attributes and one dependent variable. The network was stored into 73 lines of an ASCII file named `NeuralNetworkEx1.txt` using `imsls_f_mlff_network_write` (see [Example 1](#) of `imsls_f_mlff_network_write`).

```
#include <imsls.h>
#include <stdio.h>
int main(){
    char *filename = "NeuralNetworkEx1.txt";
    int i, j;
    int n_patterns      =13;
    int n_inputs        =4;
    int n_categorical   =0;
    int n_continuous    =4;
    int n_outputs       =1;
    int *categoricalAtt=NULL;
    float ss;
    float continuous[4*13], y[13], contAtt[4];
    float *draperSmithData;
```

```

float forecast[1], forecasts[13];
Imsls_f_NN_Network *network;
draperSmithData = imsls_f_data_sets(5,0);

for(i=0; i < n_patterns; i++){
    y[i] = draperSmithData[5*i+4];
    for(j=0; j<n_continuous; j++)
        continuous[i*n_continuous+j] = draperSmithData[5*i+j];
}
network = imsls_f_mlff_network_read(filename,
                                   IMSLS_PRINT, 0);

for(i=0; i<n_patterns; i++){
    for(j=0; j<n_inputs; j++)
        contAtt[j] = continuous[i*n_continuous+j];
    imsls_f_mlff_network_forecast(network,
    n_categorical, n_continuous, categoricalAtt, contAtt,
    IMSLS_RETURN_USER, forecast, 0);
    forecasts[i] = forecast[0];
}
printf("\nOBS      X1      X2      X3      X4      Y");
printf("      FORECAST\n");
ss = 0;
for(i=0; i<n_patterns; i++) {
    ss += (y[i]-forecasts[i])*(y[i]-forecasts[i]);
    printf("%2d %7.2f %7.2f %7.2f %7.2f %7.2f %7.2f\n",
    i, continuous[i*n_continuous], continuous[i*n_continuous+1],
    continuous[i*n_continuous+2], continuous[i*n_continuous+3],
    y[i], forecasts[i]);
}
printf("Sum of Squared Residuals: %7.2f\n", ss);
}

```

Output

Notice that the forecasts produced using `imsls_f_mlff_network_forecast` are identical to the original forecasts in [Example 1](#) of `imsls_f_mlff_network_write`.

```

Attempting to open NeuralNetworkEx1.txt for
reading network data structure
File NeuralNetworkEx1.txt Successfully Opened
File NeuralNetworkEx1.txt closed

```

OBS	X1	X2	X3	X4	Y	FORECAST
0	7.00	26.00	6.00	60.00	78.50	78.50
1	1.00	29.00	15.00	52.00	74.30	74.30
2	11.00	56.00	8.00	20.00	104.30	104.22
3	11.00	31.00	8.00	47.00	87.60	87.60
4	7.00	52.00	6.00	33.00	95.90	95.78
5	11.00	55.00	9.00	22.00	109.20	109.34
6	3.00	71.00	17.00	6.00	102.70	102.55
7	1.00	31.00	22.00	44.00	72.50	72.50
8	2.00	54.00	18.00	22.00	93.10	93.24
9	21.00	47.00	4.00	26.00	115.90	116.05
10	1.00	40.00	23.00	34.00	83.80	83.80

11	11.00	66.00	9.00	12.00	113.30	112.30
12	10.00	68.00	8.00	12.00	109.40	110.33
Sum of Squared Residuals:					1.97	

Example 2

This example illustrates the use of the optional argument `IMSLS_FILE` to read multiple neural networks previously stored into a single file using `imsls_f_mlff_network_write`. Two networks were trained using the Draper-Smith data. These data consist of 13 patterns, each with four continuous attributes and one dependent variable. The first network is trained to forecast the dependent variable using all 4 inputs and the second using only the first 3. The networks are read from an ASCII file previously created using `imsls_f_mlff_network_write` named `NeuralNetworkEx2.txt` (see [Example 2 of imsls_f_mlff_network_write](#)).

```
#include <imsls.h>
#include <stdio.h>
#include <stdlib.h>
extern FILE* imsls_fopen(char* filename, char* mode);
extern int imsls_fclose(FILE* file);

int main(){
    FILE *file;
    char *filename = "NeuralNetworkEx2.txt";
    int i, j, n;
    int n_patterns      =13;
    int n_inputs        =4;
    int n_categorical   =0;
    int n_continuous    =4;
    int n_outputs       =1;
    int n_networks      =0;
    int *categoricalAtt=NULL;
    float ss3, ss4;
    float cont4[4*13], y[13], contAtt4[4];
    float cont3[3*13];
    float *draperSmithData;
    float forecast[1], forecasts[2*13];
    Imsls_f_NN_Network **neural_network;
    draperSmithData = imsls_f_data_sets(5,0);

    for(i=0; i < n_patterns; i++){
        y[i] = draperSmithData[5*i+4];
        for(j=0; j<n_continuous; j++){
            cont4[i*n_continuous+j] = draperSmithData[5*i+j];
            if(j<3) cont3[i*3+j] = draperSmithData[5*i+j];
        }
    }

    /* open filestream */
    file = imsls_fopen(filename, "r");
    printf("File %s Opened\n", filename);
    /* Read the number of network being placed into this file */
    fscanf(file, "%d", &n_networks);
    printf("File contains %d neural networks\n", n_networks);
    neural_network = (Imsls_f_NN_Network **) malloc(n_networks*
        sizeof(Imsls_f_NN_Network *));
```

```

printf("Reading Networks and Preparing Forecasts...\n");
for(n=0; n<n_networks; n++){
    neural_network[n] = imsls_f_mlff_network_read(NULL,
        IMSLS_PRINT, IMSLS_FILE, file, 0);
    n_continuous = neural_network[n]->layers[0].n_nodes;
    printf("Preparing forecasts for network with");
    printf(" %d continuous attributes\n", n_continuous);
    for(i=0; i<n_patterns; i++){
        for(j=0; j<4; j++) contAtt4[j] = cont4[i*4+j];
        imsls_f_mlff_network_forecast(neural_network[n],
            n_categorical, n_continuous,
            categoricalAtt, contAtt4,
            IMSLS_RETURN_USER, forecast, 0);
        forecasts[n*n_patterns + i] = forecast[0];
    }
}

imsls_fclose(file);
printf("File %s Closed.\n\n", filename);
printf("                                ");
printf("FORECAST  FORECAST\n");
printf("OBS      X1      X2      X3      X4      Y      ");
printf("n_cont=3  n_cont=4\n");
ss4 = 0;
ss3 = 0;
for(i=0; i<n_patterns; i++) {
    ss4 += (y[i]-forecasts[i])*(y[i]-forecasts[i]);
    ss3 += (y[i]-forecasts[n_patterns+i])*
        (y[i]-forecasts[n_patterns+i]);
    printf("%2d %7.2f %7.2f %7.2f %7.2f %7.2f %7.2f %9.2f\n",
        i, cont4[i*4], cont4[i*4+1], cont4[i*4+2], cont4[i*4+3],
        y[i], forecasts[n_patterns+i], forecasts[i]);
}
printf("Sum of Squared Residuals for X1-X3: %7.2f\n", ss3);
printf("Sum of Squared Residuals for X1-X4: %7.2f\n", ss4);
}

```

Output

Notice that the forecasts produced using `imsls_f_mlff_network_forecast` are identical to the original forecasts in [Example 2](#) of `imsls_f_mlff_network_write`.

```

File NeuralNetworkEx2.txt Opened
File contains 2 neural networks
Reading Networks and Preparing Forecasts...
Network restored from file. File not closed.
Preparing forecasts for network with 4 continuous attributes
Network restored from file. File not closed.
Preparing forecasts for network with 3 continuous attributes
File NeuralNetworkEx2.txt Closed.

```

OBS	X1	X2	X3	X4	Y	FORECAST FORECAST	
						n_cont=3	n_cont=4
0	7.00	26.00	6.00	60.00	78.50	78.97	78.50
1	1.00	29.00	15.00	52.00	74.30	73.62	74.30
2	11.00	56.00	8.00	20.00	104.30	104.61	104.30
3	11.00	31.00	8.00	47.00	87.60	87.32	87.60

4	7.00	52.00	6.00	33.00	95.90	94.88	95.90
5	11.00	55.00	9.00	22.00	109.20	108.82	109.20
6	3.00	71.00	17.00	6.00	102.70	102.69	102.70
7	1.00	31.00	22.00	44.00	72.50	72.97	72.50
8	2.00	54.00	18.00	22.00	93.10	94.28	93.10
9	21.00	47.00	4.00	26.00	115.90	115.04	115.90
10	1.00	40.00	23.00	34.00	83.80	83.64	83.80
11	11.00	66.00	9.00	12.00	113.30	114.45	113.30
12	10.00	68.00	8.00	12.00	109.40	109.20	109.40
Sum of Squared Residuals for X1-X3:						5.76	
Sum of Squared Residuals for X1-X4:						0.00	

Fatal Errors

IMSLS_FILE_OPEN_FAILURE

Unable to open file for reading neural network.

mlff_initialize_weights

Initializes weights for multilayered feedforward neural networks prior to network training using one of four user selected methods.

Synopsis

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

```
float *imsls_f_mlff_initialize_weights (Imsls_f_NN_Network *network, int n_patterns,  
int n_nominal, int n_continuous, int nominal[ ], float continuous[ ], ..., 0)
```

The type *double* function is `imsls_d_mlff_initialize_weights`.

Required Arguments

Imsls_f_NN_Network *network (Input/Output)

Pointer to a structure of type *Imsls_f_NN_Network* containing the parameters that define the feedforward network's architecture, including network weights and bias values. For more details, see [imsls_f_mlff_network](#). When network training is successful, the weights and bias values in `network` are replaced with the values calculated for the optimum trained network.

int n_patterns (Input)

Number of training patterns.

int n_nominal (Input)

Number of unencoded nominal attributes.

int nominal[] (Input)

Array of size `n_patterns` by `n_nominal` containing the nominal input variables.

int n_continuous (Input)

Number of continuous attributes, including ordinal attributes encoded using cumulative percentage.

float continuous[] (Input)

Array of size `n_patterns` by `n_continuous` containing the continuous and scaled ordinal input variables.

Return Value

Pointer to an array of length `network->n_links + (network->n_nodes-network->n_inputs)` containing the initialized weights. See the [Description](#) section for details on weight ordering. This space can be released by using the `ims1s_free` function.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_mlff_initialize_weights (Imsls_f_NN_Network *network, int n_patterns,
    int n_nominal, int n_continuous, int nominal[ ], float continuous[ ],

    IMSLS_METHOD, int method,

    IMSLS_PRINT,

    IMSLS_CLASSIFICATION, int classification[ ],

    IMSLS_RETURN_USER, float weights[ ],

    0)
```

Optional Arguments

IMSLS_METHOD, *int method* (Input)

Specifies the algorithm to use for initializing weights. **method** contains the weight initialization method to be used. Valid values for **method** are:

method	Algorithm
IMSLS_EQUAL	Equal weights
IMSLS_RANDOM	Random Weights
IMSLS_PRINCIPAL_COMPONENTS	Principal Component Weights
IMSLS_DISCRIMINANT	Discriminant Analysis Weights

The discriminant weights method can only be used to initialize weights for classification networks without binary encoded nominal attributes. See the [Description](#) section for details.

Default: **method** = IMSLS_RANDOM.

IMSLS_PRINT, (Input)

Initial weights are printed.

Default: No printing is performed.

IMSLS_CLASSIFICATION, *int classification[]* (Input)

An array of length **n_patterns** containing the encoded training target classifications which must be integers from 0 to **n_classes**-1. Here **n_classes** = **network->n_outputs** except when **n_outputs**=1 then **n_classes** =2. **classification[i]** is the target classification for the *i*-th training pattern described by **nominal[i]** and **continuous[i]**. This option is used by the discriminant analysis weight initialization. This option is ignored for all other methods.

IMSLS_RETURN_USER, *float weights*[] (Output)

If specified, the initialized weights are returned in a user provided array of length

-network--->n_links---(network->n_nodes-network->n_inputs)

Description

Function `imsls_f_mlff_initialize_weights` calculates initial values for the weights of a feedforward neural network using one of the following algorithms:

method	Algorithm
IMSLS_EQUAL	Equal weights
IMSLS_RANDOM	Random Weights
IMSLS_PRINCIPAL_COMPONENTS	Principal Component Weights
IMSLS_DISCRIMINANT	Discriminant Analysis Weights

The keyword `IMSLS_METHOD` can be used to select the algorithm for weight initialization. By default, the random weights algorithm will be used.

The 3-layer feed forward network with 3 input attributes and 6 perceptrons in [Figure 33](#) is used to describe the initialization algorithms. In this example, one of the input attributes is continuous (X_3) and the others are nominal (X_1 and X_2).

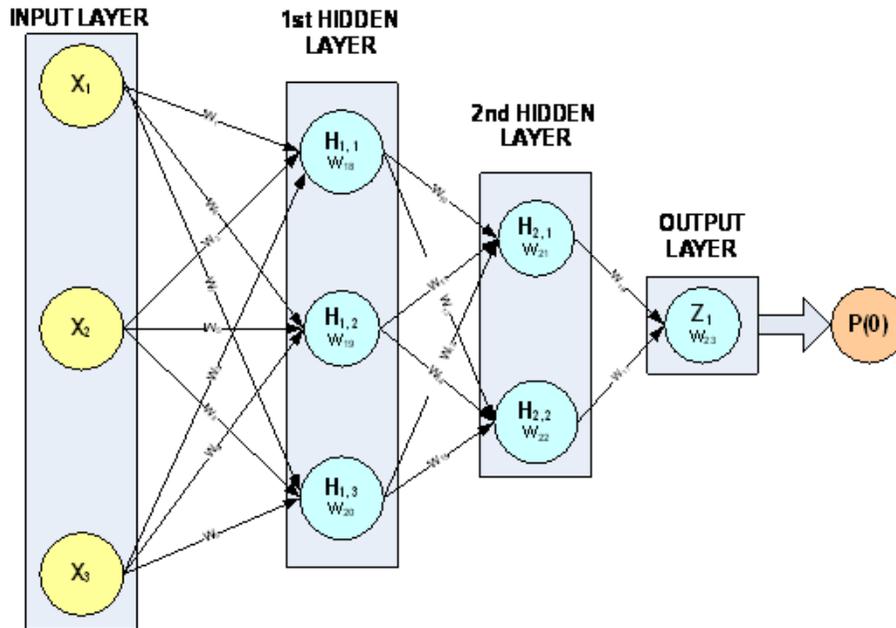


Figure 33, A 3-layer, Feed Forward Network with 3 Input Attributes and 6 Perceptrons

This network has a total of 23 weights. The first nine weights, labeled W_1, W_2, \dots, W_9 , are the weights assigned to the links connecting the network inputs to the perceptrons in the first hidden layer. Note that W_1, W_2, W_4, W_5, W_7 , and W_8 are assigned to the two nominal attributes and W_3, W_6 and W_9 are assigned to the continuous attribute. All neural network functions in the C Stat Library use this weight ordering. Weights for all nominal attributes are placed before the weights for any continuous attributes.

PERCEPTRON	POTENTIAL
$H_{1,1}$	$g_1 = W_{18} + W_1X_1 + W_2X_2 + W_3X_3$
$H_{1,2}$	$g_2 = W_{19} + W_4X_1 + W_5X_2 + W_6X_3$
$H_{1,3}$	$g_3 = W_{20} + W_7X_1 + W_8X_2 + W_9X_3$
$H_{2,1}$	$g_4 = W_{21} + W_{10}g_1 + W_{11}g_2 + W_{12}g_3$
$H_{2,2}$	$g_5 = W_{22} + W_{13}g_1 + W_{14}g_2 + W_{15}g_3$
Z_1	$g_6 = W_{23} + W_{16}g_4 + W_{17}g_5$

The next six weights are the weights between the first and second hidden layers, and W_{16} and W_{17} are the weights for the links connecting the second hidden layer to the output layer. The last six elements in the `weights` array are the perceptron bias weights. These weights, $W_{18}, W_{19}, \dots, W_{23}$ are the weights for perceptrons $H_{1,1}, \dots, H_{1,3}, H_{2,1}, \dots, H_{2,3}$, and Z_1 , respectively.

The perceptron potential calculations for this network are described in the table above. Following the notation presented in the introduction to this chapter, g_1, g_2, \dots, g_5 are the perceptron activations from perceptrons $H_{1,1}, \dots, H_{1,3}, H_{2,1}, \dots, H_{2,3}$, respectively.

All initialization algorithms in `mlff_initialize_weights` set the weights for perceptrons not linked directly to the input perceptrons in the same manner. Bias weights for perceptrons not directly linked to input attributes are set to zero. All non-bias weights for these same perceptrons are assigned a value of $1/k$ where k =the number of links into that perceptron (`network->nodes[i].n_inlinks`).

For example, in this network, the last three bias weights W_{21}, W_{22} and W_{23} are initialized to zero since perceptrons $H_{2,1}, H_{2,1}$ and Z_1 and not directly connected to the input attributes. The other weights to perceptrons $H_{2,1}$ and $H_{2,2}$ are assigned a value of one half since these perceptrons each have only two input links. The weights to the output perceptron, Z_1 , are also one half since Z_1 has two inputs links.

The calculation of the weights for the links between the input attributes and their perceptrons are initialized differently by the four algorithms. All algorithms, however, scale these weights so that the average potential for the first layer perceptrons is zero. This reduces the possibility of saturation or numerical overflow during the initial stages of optimization.

Equal Weights (`method=IMSL5_EQUAL`)

In this algorithm, the non-bias weights for each link between the input attributes and the perceptrons in the first layer are initialized to:

$$W_i = \frac{1}{nS_i}$$

where W_i is the weight for all links between the i -th input attributes, n is equal to the total number of input attributes and S_i is equal to the standard deviation of the potential for the i -th input attribute. In the above example, the values for weights W_1, W_2, \dots, W_9 , each would be set to:

$$\frac{1}{3S_i}$$

since this network has three input attributes.

Next the average potential for each of the perceptrons connected to the input layer is calculated by:

$$W_i = \frac{-\bar{X}_i}{S_i}$$

where \bar{X}_i is equal to the average potential for the i -th input attribute. All other bias weights are set to zero.

Random Weights (`method=IMSLs_RANDOM`)

This algorithm first generates random values for the input layer weights using the Uniform [-0.5, +0.5] distribution. These are then scaled using the standard deviation of the input layer potentials.

$$W_i = \frac{U}{S_i}$$

where U is a random number uniformly distributed on the interval [-0.5,+0.5] and S_i is equal to the standard deviation of the potential for the i -th input attribute.

Next the average potential for each of the perceptrons connected to the input layer is calculated by:

$$W_i = \frac{-\bar{X}_i}{S_i}$$

where \bar{X}_i is equal to the average potential for the i -th input attribute. All other bias weights are set to zero.

Principal Component Weights (`method=IMSLs_PRINCIPAL_COMPONENTS`)

This uses principal component analysis to generate weights. The arrays nominal and continuous are combined into a single matrix. The correlation matrix of this matrix is decomposed using principal component analysis. The elements of the principal components from this analysis are used to initialize weights associated with the network inputs. As with the other methods the principal component weights are scaled by using the standard deviation of the potential for the perceptrons connected to the input layer:

$$W_i = \frac{\xi_{ij}}{S_j}$$

where W_i is the weight for the link between the i -th input attribute and the j -th perceptron, ξ_{ij} is the i -th value of the j -th principal component, and S_j is equal to the standard deviation of the potential for the i -th input attribute.

If the number of principal components is less than the number of perceptrons in the first layer, i.e., $(n_continuous+n_nominal) < n_layer1$, where n_layer1 is the number of perceptrons in the first layer, then it is not possible to initialize all weights with principal components. In this case, the first $(n_continuous + n_nominal)$ perceptrons are initialized using the principal components and then the remainder are initialized using random weights (`method=IMSLs_RANDOM`).

As with the other methods, the bias weights for each of the first layer perceptrons is set to ensure that the average potential in this layer is equal to zero:

$$W_i = \frac{-\bar{X}_{ij}}{S_{ij}}$$

where \bar{X}_{ij} is equal to the average potential for the link between i -th input attribute and the j -th first layer perceptron, and S_{ij} is the standard deviation for this same potential.

Discriminant Weights (`method=IMSLD_DISCRIMINANT`)

This method is very similar to principal component weights. Instead the discriminant analysis elements replace the principal component elements. The weights between the i -th input attribute and the j -th perceptron in the first layer are calculated by:

$$W_i = \frac{\theta_{ij}}{S_i}$$

Where W_i is the weight for the link between the i -th input attribute and the j -th perceptron, θ_{ij} is the i -th value of the j -th discriminant component, and S_i is equal to the standard deviation of the potential for the i -th input attribute.

If the number of discriminant components is less than the number of perceptrons in the first layer, i.e., $(n_continuous + n_nominal) < n_layer1$, where n_layer1 is the number of perceptrons in the first layer, then it is not possible to initialize all weights with components from the discriminant analysis. In this case, the first $(n_continuous + n_nominal)$ perceptrons are initialized using the discriminant components and then the remainder are initialized using random weights (`method=IMSLD_RANDOM`).

As with the other methods, the bias weights for each of the first layer perceptrons is set to ensure that the average potential in this layer is equal to zero:

$$W_i = \frac{-\bar{X}_{ij}}{S_{ij}}$$

where \bar{X}_{ij} is equal to the average potential for the link between i -th input attribute and the j -th first layer perceptron, and S_{ij} is the standard deviation for this same potential.

Examples

Example 1

This example illustrates random initialization algorithms for a three layer network with one output. The first and second hidden layers contain three and two perceptrons for a total of five network perceptrons, respectively.

The nine input attributes consist of two continuous attributes plus seven binary attributes encoded from two nominal attributes using binary encoding.

The weights are initialized using the random weights algorithm. This results in different weights for every perceptron in the first hidden layer. The weights in other layers are initialized using equal weights. It should be noted that the bias weights in the first layer are not random. Except for the discriminant weights algorithm, the bias weights are always calculated to ensure that the average potential for each perceptron in the first layer is zero.

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

int main(){
    Imsls_f_NN_Network *network;
    int i, j, k, m;
    int n_patterns =24; /* no. of training patterns */
    int n_nvars =2; /* 2 nominal unencoded variables */
    int n_nominal =7; /* 7 inputs for the binary encoded
                       nominal vars */
    int n_continuous =2; /* 2 continuous input attributes */
    int nominalIn[24]; /* work arrays used to encode */
    int *nominalOut; /* nominal data */
    int n_classes;
    int classification[24] = {
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1
    };
    /* raw nominal input data */
    int nominal_unencoded[2*24] =
    {
        0, 0, 0, 1, 0, 2,
        1, 0, 1, 1, 1, 2,
        2, 0, 2, 1, 2, 2,
        3, 0, 3, 1, 3, 2,
        0, 0, 0, 1, 0, 2,
        1, 0, 1, 1, 1, 2,
        2, 0, 2, 1, 2, 2,
        3, 0, 3, 1, 3, 2
    };

    /* input array for binary encoded version of
       nominal_unencoded[] array above */
    int nominal[7*24];
    float *weights;
    float continuous[2*24] =
    {
        0.00,0.00,0.02,0.02,0.04,0.04,0.06,0.06,0.08,0.08,0.10,0.10,
        0.12,0.12,0.14,0.14,0.16,0.16,0.18,0.18,0.20,0.20,0.22,0.22,
        0.24,0.28,0.26,0.30,0.28,0.32,0.30,0.34,0.32,0.36,0.34,0.38,
        0.36,0.40,0.38,0.42,0.40,0.44,0.42,0.46,0.44,0.48,0.46,0.50
    };

    /* Setup Nominal Input Attributes Using Binary Encoding */
    m=0;
    for (i=0; i<n_nvars; i++){
        for (j=0; j<n_patterns; j++) {
            nominalIn[j] = nominal_unencoded[2*j+i] + 1;
        }
    }
}
```

```

        nominalOut = imsls_unsupervised_nominal_filter(n_patterns,
            &n_classes, nominalIn, 0);
        for(k=0; k<n_classes; k++){
            for(j=0; j<n_patterns; j++){
                nominal[j*n_nominal+m] = nominalOut[j*n_classes+k];
            }
            m++;
        }
    }
    printf("\tINPUT TRAINING PATTERNS\n");
    printf("\tY  Nom1 Nom2  X0  X1 \n");
    for(i=0; i<n_patterns; i++){
        printf("\t%d  %d \t %d  %f  %f \n", classification[i],
            nominal_unencododed[i*2], nominal_unencododed[i*2+1],
            continuous[i*2], continuous[i*2+1]);
    }

    /* Binary classification network 9 inputs 1 output = 2 classes */
    network = imsls_f_mlff_network_init(9, 1);
    imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 3, 0);
    imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 2,
        IMSLS_LINK_ALL, 0);

    /* Note the following statement is for repeatable output */
    imsls_random_seed_set(5555);

    /* Random Weights */
    weights = imsls_f_mlff_initialize_weights(network, n_patterns,
        n_nominal, n_continuous, nominal, continuous,
        IMSLS_PRINT, 0);
}

```

Output

INPUT TRAINING PATTERNS				
Y	Nom1	Nom2	X0	X1
0	0	0	0.000000	0.000000
0	0	1	0.020000	0.020000
0	0	2	0.040000	0.040000
0	1	0	0.060000	0.060000
0	1	1	0.080000	0.080000
0	1	2	0.100000	0.100000
1	2	0	0.120000	0.120000
1	2	1	0.140000	0.140000
1	2	2	0.160000	0.160000
1	3	0	0.180000	0.180000
1	3	1	0.200000	0.200000
1	3	2	0.220000	0.220000
0	0	0	0.240000	0.280000
0	0	1	0.260000	0.300000
0	0	2	0.280000	0.320000
0	1	0	0.300000	0.340000
0	1	1	0.320000	0.360000
0	1	2	0.340000	0.380000
1	2	0	0.360000	0.400000
1	2	1	0.380000	0.420000
1	2	2	0.400000	0.440000
1	3	0	0.420000	0.460000

```
1 3 1 0.440000 0.480000
1 3 2 0.460000 0.500000
```

```
-----
- NETWORK WEIGHTS INITIALIZED USING
- RANDOM WEIGHTS
- Input Attributes: 9
- Nominal: 2
- Nominal(encoded): 7
- Continuous: 2
- Output Attributes: 1
- Layers: 3
- Perceptrons: 6
- Weights: 41
- Patterns: 24
```

```
-----
----- HIDDEN LAYER 1 -----
```

```
--- Perceptron 0 ---
```

```
Link from Input Node Weight
N0 0.937069
N1 -0.547569
N2 1.468248
N3 0.107160
N4 -0.884992
N5 -0.814069
N6 -1.979680
X7 -0.041228
X8 -1.368315
Bias 3.3099
```

```
--- Perceptron 1 ---
```

```
Link from Input Node Weight
N0 -0.308421
N1 -1.058450
N2 -0.981207
N3 1.040820
N4 -0.033493
N5 -0.575732
N6 0.571939
X7 0.811886
X8 -0.415498
Bias 0.573286
```

```
--- Perceptron 2 ---
```

```
Link from Input Node Weight
N0 -1.117744
N1 0.620799
N2 0.174895
N3 -0.100458
N4 -0.961071
N5 0.854179
N6 0.046423
X7 0.880998
X8 -0.903982
Bias 1.00437
```

```
-----
----- HIDDEN LAYER 2 -----
```

```

--- Perceptron 0 ---
Link from Input Node Weight
P0          0.333333
P1          0.333333
P2          0.333333
Bias 0

--- Perceptron 1 ---
Link from Input Node Weight
P0          0.333333
P1          0.333333
P2          0.333333
Bias 0

-----
----- OUTPUT   LAYER -----
-----

--- Perceptron 0 ---
Link from Input Node Weight
P3          0.500000
P4          0.500000
Bias 0

-----

```

Example 2

This example illustrates the discriminant weights initialization algorithm for a three layer network with one output. The first and second hidden layers contain three and two perceptrons for a total of five network perceptrons, respectively.

The data are the same as Example 1, and the network structure is the same except that all nominal input attributes are removed. This was necessary since the discriminant weights algorithm only works when all input attributes are continuous.

The discriminant weights algorithm initializes the weights in the first hidden layer to the coefficients of the discriminant functions. Since this example is a binary classification example, the number of discriminant functions is equal to the number of classes, two, but there are three perceptrons in the first layer. The weights for the first two perceptrons in this layer are the discriminant function coefficients, including the bias weight. The weights for the last perceptron in this layer were determined randomly.

```

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

int main(){
    Imsls_f_NN_Network *network;
    int i, j, k, m;
    int n_patterns =24; /* no. of training patterns */
    int n_continuous =2; /* 2 continuous input attributes */
    int classification[24] = {
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1
    };
};

```

```

float *weights;
float continuous[2*24] =
{
    0.00,0.00,0.02,0.02,0.04,0.04,0.06,0.06,0.08,0.08,0.10,0.10,
    0.12,0.12,0.14,0.14,0.16,0.16,0.18,0.18,0.20,0.20,0.22,0.22,
    0.24,0.28,0.26,0.30,0.28,0.32,0.30,0.34,0.32,0.36,0.34,0.38,
    0.36,0.40,0.38,0.42,0.40,0.44,0.42,0.46,0.44,0.48,0.46,0.50
};

printf("\tINPUT TRAINING PATTERNS\n");
printf("\tY      X0      X1 \n");
for(i=0; i<n_patterns; i++){
    printf("\t%d  %f  %f \n", classification[i],
        continuous[i*2], continuous[i*2+1]);
}

/* Binary classification network 2 inputs 1 output = 2 classes */
network = imsls_f_mlff_network_init(2, 1);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 3, 0);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 2,
    IMSLS_LINK_ALL, 0);

/* Discriminant weights */
/* Set seed for consistent results */
imsls_random_seed_set(12357);
weights = imsls_f_mlff_initialize_weights(network, n_patterns,
    0, n_continuous, NULL, continuous,
    IMSLS_METHOD, IMSLS_DISCRIMINANT,
    IMSLS_CLASSIFICATION, classification,
    IMSLS_PRINT, 0);
}

```

Output

```

INPUT TRAINING PATTERNS
Y      X0      X1
0  0.000000  0.000000
0  0.020000  0.020000
0  0.040000  0.040000
0  0.060000  0.060000
0  0.080000  0.080000
0  0.100000  0.100000
1  0.120000  0.120000
1  0.140000  0.140000
1  0.160000  0.160000
1  0.180000  0.180000
1  0.200000  0.200000
1  0.220000  0.220000
0  0.240000  0.280000
0  0.260000  0.300000
0  0.280000  0.320000
0  0.300000  0.340000
0  0.320000  0.360000
0  0.340000  0.380000
1  0.360000  0.400000
1  0.380000  0.420000

```

```
1 0.400000 0.440000
1 0.420000 0.460000
1 0.440000 0.480000
1 0.460000 0.500000
Discriminant Analysis Classification Error Rate = 0.000000
```

```
-----
- NETWORK WEIGHTS INITIALIZED USING
- DISCRIMINANT WEIGHTS
- Input Attributes: 2
- Nominal: 0
- Nominal(encoded): 0
- Continuous: 2
- Output Attributes: 1
- n_classes: 2
- Layers: 3
- Perceptrons: 6
- Weights: 20
- Patterns: 24
-----
```

```
----- HIDDEN LAYER 1 -----
```

```
--- Perceptron 0 ---
```

```
Link from Input Node Weight
X0 229.165253
X1 -189.879715
Bias -2.13362
```

```
--- Perceptron 1 ---
```

```
Link from Input Node Weight
X0 889.167236
X1 -755.595703
Bias -12.5051
```

```
--- Perceptron 2 ---
```

```
Link from Input Node Weight
X0 -4.495886
X1 -0.976032
Bias 6.07217
```

```
----- HIDDEN LAYER 2 -----
```

```
--- Perceptron 0 ---
```

```
Link from Input Node Weight
P0 0.333333
P1 0.333333
P2 0.333333
Bias 0
```

```
--- Perceptron 1 ---
```

```
Link from Input Node Weight
P0 0.333333
P1 0.333333
P2 0.333333
Bias 0
```

```
----- OUTPUT LAYER -----
```

--- Perceptron 0 ---	
Link from Input Node	Weight
P3	0.500000
P4	0.500000
Bias 0	

mlff_network_trainer



[more...](#)

Trains a multilayered feedforward neural network.

Synopsis

```
#include <imsls.h>

float *imsls_f_mlff_network_trainer (Imsls_f_NN_Network *network, int n_patterns,
    int n_nominal, int n_continuous, int nominal[ ], float continuous[ ], float output[ ], ...,
    0)
```

The type *double* function is `imsls_d_mlff_network_trainer`.

Required Arguments

Imsls_f_NN_Network *network (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_patterns (Input)

Number of network training patterns.

int n_nominal (Input)

Number of nominal attributes. `n_nominal + n_continuous` must equal `n_inputs`, where `n_inputs` is the number of input attributes in the network. `n_inputs = network->n_inputs`. For more details, see [imsls_f_mlff_network](#).

int n_continuous (Input)

Number of continuous attributes. `n_nominal + n_continuous` must equal `n_inputs`, where `n_inputs` is the number of input attributes in the network. `n_inputs = network->n_inputs`. For more details, see [imsls_f_mlff_network](#).

int nominal[] (Input)

Array of size `n_patterns` by `n_nominal` containing values for the nominal input attributes. The *i*-th row contains the nominal input attributes for the *i*-th training pattern.

float continuous[] (Input)

Array of size `n_patterns` by `n_continuous` containing values for the continuous input attributes. The *i*-th row contains the continuous input attributes for the *i*-th training pattern.

float output[] (Input)

Array of size `n_patterns` by `n_outputs` containing the output training patterns, where `n_outputs` is the number of output perceptrons in the network.

`n_outputs = network->n_outputs`. For more details, see [imsls_f_mlff_network](#).

Return Value

An array of length 5 containing the summary statistics from the network training, organized as follows:

	Training Statistics
0	Error sum of squares at the optimum.
1	Total number of Stage I iterations.
2	Smallest error sum of squares after Stage I training.
3	Total number of Stage II iterations.
4	Smallest error sum of squares after Stage II training.

This space can be released by using the [imsls_free](#) function.

If training is unsuccessful, `NULL` is returned.

Synopsis with Optional Arguments

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

```
float *imsls_f_mlff_network_trainer (imsls_f_NN_Network *network, int n_patterns,
    int n_nominal, int n_continuous, float nominal[ ], 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_WEIGHT_INITIALIZATION_METHOD, int method,  
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_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 patterns used during each epoch. If epoch training is not needed, set **epoch_size = n_patterns** and **n_epochs = 1**. Stage I is implemented using Quasi-Newton optimization and steepest ascent with gradients estimated using the backward propagation method.
Default: **n_epochs=15**, **epoch_size = n_patterns**.
- IMSLS_NO_STAGE_II**, (Input)
Specifies no Stage II training is performed. Stage II is implemented using Quasi-Newton optimization with numerical gradients.
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, $\epsilon^{2/3}$)`, where ϵ is the machine precision, `max(10-20, $\epsilon^{2/3}$)` is used in double precision.

IMSLS_GRAD_TOL, *float grad_tol* (Input)

Scaled gradient tolerance in the optimizer.

Default: `grad_tol = $\epsilon^{1/2}$` , where ϵ is the machine precision, `$\epsilon^{1/3}$` is used in double precision.

IMSLS_TOLERANCE, *float tolerance* (Input)

Absolute accuracy tolerance for the sum of squared errors in the optimizer.

Default: `tolerance = 0.1`.

IMSLS_WEIGHT_INITIALIZATION_METHOD, *int method[]* (Input)

The method to use for initializing network weights prior to network training. One of the following four values is accepted:

method	Algorithm
IMSLS_EQUAL	Equal weights.
IMSLS_RANDOM	Random weights.
IMSLS_PRINCIPAL_COMPONENTS	Principal Component Weights.
IMSLS_NN_NETWORK	No initialization method will be performed. Weights in <code>imsls_f_NN_Network</code> structure <code>network</code> will be used instead.

See [imsls_f_mlff_initialize_weights](#) for a detailed description of the initialization methods.

Default: `method = IMSLS_RANDOM`.

IMSLS_PRINT, (Input)

Intermediate results are printed during network training.

Default: No printing is performed.

IMSLS_RESIDUAL, *float **residuals* (Output)

The address of a pointer to the internally allocated array of size `n_patterns` 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 = network->n_outputs`.

IMSLS_RESIDUAL_USER, *float residuals[]* (Output)

Storage for array `residuals` provided by user. See `IMSLS_RESIDUAL`.

`IMSLS_GRADIENT`, *float **gradients* (Output)

The address of a pointer to the internally allocated 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 = network->n_links`, `n_nodes = network->n_nodes`, and `n_inputs = network->n_inputs`.

`IMSLS_GRADIENT_USER`, *float gradients[]* (Output)

Storage for array `gradients` provided by user. See `IMSLS_GRADIENT`.

`IMSLS_FORECASTS`, *float **forecasts* (Output)

The address of a pointer to the internally allocated array of size `n_patterns` by `n_outputs`, where `n_outputs` is the number of output perceptrons in the network.

`n_outputs = network->n_outputs`. 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. See [Return Value](#) for details.

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. nominal input attributes
2. continuous input attributes
3. continuous output

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:

$$\mathit{nominalAtt} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Continuous input and output data are passed to the training engine using two double precision arrays: `continuous` and `output`. The number of rows in each of these matrices is `n_patterns`. The number of columns in `continuous` and `output`, 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`, `network->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`.

The trained network can be saved and retrieved using `imsls_f_mlff_network_write` and `imsls_f_mlff_network_read`.

Example

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:

$$Y = 10 * X_1 + 20 * X_2 + 30 * X_3 + 20 * X_4,$$

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:


```

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,
0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,0,0,1,
};

/* 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 *network;
float *stats;
int n_patterns= 100, n_nominal=3, n_continuous=1;
int i,j,k, wIdx;
float *residuals, *forecasts;
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 * nominal[i*3]) + (20 * nominal[i*3+1]) +
        (30 * nominal[i*3+2]) + (20 * continuous[i]);
}

/* Create network */
network = imsls_f_mlff_network_init(4,1);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 3,
    IMSLS_ACTIVATION_FCN, 1, &hidActFcn,
    IMSLS_LINK_ALL, 0);

```

```
/* Set initial weights */
for (j=network->n_inputs; j < network->n_nodes ; j++)
{
    for (k=0; k < network->nodes[j].n_inLinks; k++)
    {
        wIdx = network->nodes[j].inLinks[k];
        /* set specific layer weights */
        if (network->nodes[j].layer_id == 1) {
            network->links[wIdx].weight = 0.25;
        } else if (network->nodes[j].layer_id == 2) {
            network->links[wIdx].weight = 0.33;
        }
    }
}

/* Initialize seed for consisten results */
imsls_random_seed_set(12345);
stats = imsls_f_mlff_network_trainer(network, n_patterns,
    n_nominal, n_continuous, nominal, 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 = network->nodes[network->n_nodes-4].bias *
    network->links[12].weight +
    network->nodes[network->n_nodes-3].bias *
    network->links[13].weight +
    network->nodes[network->n_nodes-2].bias *
    network->links[14].weight;
/* the bias of the output node */
bias += network->nodes[network->n_nodes-1].bias;
coef1 = network->links[0].weight * network->links[12].weight;
coef1 += network->links[1].weight * network->links[13].weight;
coef1 += network->links[2].weight * network->links[14].weight;
coef2 = network->links[3].weight * network->links[12].weight;
coef2 += network->links[4].weight * network->links[13].weight;
coef2 += network->links[5].weight * network->links[14].weight;
coef3 = network->links[6].weight * network->links[12].weight;
coef3 += network->links[7].weight * network->links[13].weight;
coef3 += network->links[8].weight * network->links[14].weight;
coef4 = network->links[9].weight * network->links[12].weight;
coef4 += network->links[10].weight * network->links[13].weight;
coef4 += network->links[11].weight * network->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(network);
}

```

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-020
grad_tol       = 1e-020
tolerance      = 0.000010

```

STAGE I TRAINING STARTING

```

Stage I: Epoch 1 - Epoch Error SS = 3870.44 (Iterations=7)
Stage I: Epoch 2 - Epoch Error SS = 7.41238e-011 (Iterations=74)
Stage I Training Converged at Epoch = 2

```

STAGE I FINAL ERROR SS = 0.000000

OPTIMUM WEIGHTS AFTER STAGE I TRAINING:

```

weight[0] = 2.29881   weight[1] = 4.67622   weight[2] = 5.82167
weight[3] = 2.01955   weight[4] = 3.02815   weight[5] = 5.61873
weight[6] = 9.46591   weight[7] = 7.44722   weight[8] = 1.82561
weight[9] = 6.08981   weight[10] = 11.1714  weight[11] = 5.30152
weight[12] = 2.15733  weight[13] = 2.26835   weight[14] = -0.23573
weight[15] = -3.4723  weight[16] = 0.100865  weight[17] = 2.71152
weight[18] = 6.50345

```

STAGE I TRAINING CONVERGED

STAGE I ERROR SS = 0.000000

GRADIENT AT THE OPTIMUM WEIGHTS

```

g[0] = 0.000014      weight[0] = 2.298808
g[1] = 0.000045      weight[1] = 4.676218
g[2] = -0.000012     weight[2] = 5.821669
g[3] = 0.000026      weight[3] = 2.019547
g[4] = 0.000015      weight[4] = 3.028149
g[5] = 0.000048      weight[5] = 5.618728
g[6] = -0.000013     weight[6] = 9.465913
g[7] = 0.000027      weight[7] = 7.447217
g[8] = -0.000002     weight[8] = 1.825613
g[9] = -0.000005     weight[9] = 6.089811
g[10] = 0.000001     weight[10] = 11.171391

```

g[11] =	-0.000003	weight[11] =	5.301523
g[12] =	0.000028	weight[12] =	2.157330
g[13] =	0.000176	weight[13] =	2.268351
g[14] =	0.000199	weight[14] =	-0.235730
g[15] =	0.000047	weight[15] =	-3.472301
g[16] =	0.000050	weight[16] =	0.100865
g[17] =	-0.000005	weight[17] =	2.711518
g[18] =	0.000022	weight[18] =	6.503448

Training Completed

Predictions for Last Ten Observations:

observation[90]	49.297478	Prediction	49.297474	Residual	-0.000004
observation[91]	32.435097	Prediction	32.435093	Residual	-0.000004
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	48.623795	Residual	0.000000
observation[95]	37.623909	Prediction	37.623909	Residual	0.000000
observation[96]	41.569431	Prediction	41.569431	Residual	0.000000
observation[97]	36.828972	Prediction	36.828972	Residual	0.000000
observation[98]	48.690826	Prediction	48.690830	Residual	0.000004
observation[99]	32.048107	Prediction	32.048107	Residual	0.000000
Bias: -1.397840					
X1: 10.000000					
X2: 20.000000					
X3: 30.000000					
X4: 20.000000					

mlff_network_forecast

Calculates forecasts for trained multilayered feedforward neural network.

Synopsis

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

```
float *imsls_f_mlff_network_forecast (Imsls_f_NN_Network *network, int n_nominal,  
int n_continuous, int nominal[], float continuous[], ..., 0)
```

The type *double* function is `imsls_d_mlff_network_forecast`.

Required Arguments

Imsls_f_NN_Network *network (Input)

Pointer to a structure of type *Imsls_f_NN_Network* containing the trained feedforward network. See [imsls_f_mlff_network](#).

int n_nominal (Input)

Number of nominal attributes.

int n_continuous (Input)

Number of continuous attributes.

int nominal[] (Input)

Array of size `n_nominal` containing the nominal input variables.

float continuous[] (Input)

Array of size `n_continuous` containing the continuous input variables.

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 = network->n_outputs`. This space can be released by using the [imsls_free](#) function.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_mlff_network_forecast (Imsls_f_NN_Network *network, int n_nominal,
    int n_continuous, int nominal[ ], 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 a user-supplied array `forecasts` of size `n_outputs`, where `n_outputs` is the number of perceptrons in the network, `n_outputs = network->n_outputs`.

Description

Function `imsls_f_mlff_network_forecast` 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 nominal and continuous attributes that are to be used as network inputs for calculating the forecast.

Training Data

Neural network training data consists of the following three types of data:

1. nominal 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:

$$\text{nominal} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Continuous input and output data are passed to the training engine using two double precision arrays: `continuous` and `output`. The number of rows in each of these matrices is `n_patterns`. The number of columns in `continuous` and `output`, 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 forecast routine through the structure `lmsls_f_NN_Network`. See [lmsls_f_mlff_network](#).

Forecast Calculation

The forecast is calculated from the input attributes, network structure and weights provided in the structure `lmsls_f_NN_Network`.

Example

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:

$$Y = 10 * X_1 + 20 * X_2 + 30 * X_3 + 20 * X_4,$$

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:

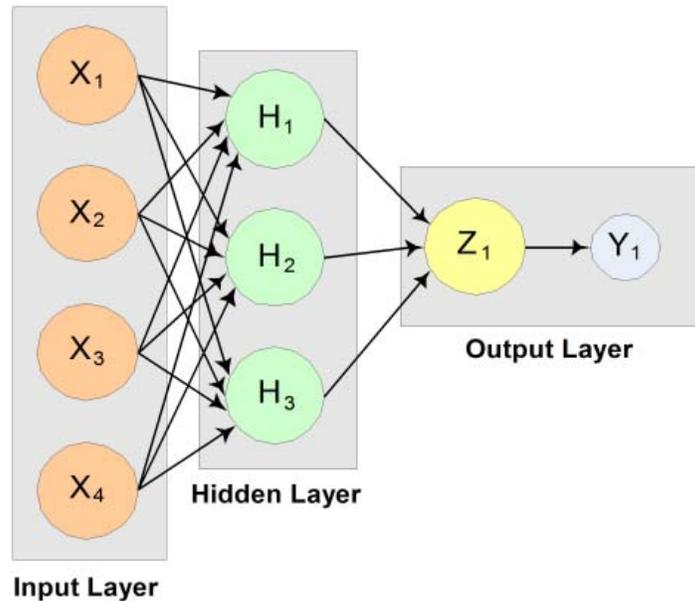


Figure 35, A 2-layer, Feedforward Network with 4 Inputs and 1 Output

There are a total of 100 outputs. The first 90 outputs use `imsls_f_mlff_network_trainer` to train the network and the last 10 outputs use `imsls_mlff_network_forecast` to forecast and compare the actual outputs.

```
#include <imsls.h>
#include <stdio.h>
int main ()
{
    int nominal[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, 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, 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, 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,
        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
};

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
};

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 NOMINAL(i,j) nominal[i*n_nominal+j]
#define NOMINALOBS(i,j) nominalObs[i*n_nominal+j]
Imsls_f_NN_Network *network;

float *stats;
int n_patterns = 100, n_nominal = 3, n_continuous = 1;

```

```

int i, j, k, wIdx;
float *forecasts;
/* for forecasting */
int nominalObs[3] = { 0, 0, 0 };
float continuousObs[1] = { 0 };
float x, y;
float *cont;

/* Scale continuous attribute to the interval [0, 1] */
cont = imsls_f_scale_filter (n_patterns, continuous, 1,
    IMSLS_SCALE_LIMITS, 0.0, 10.0, 0.0, 1.0, 0);
network = imsls_f_mlff_network_init (4, 1);
imsls_f_mlff_network (network,
    IMSLS_CREATE_HIDDEN_LAYER, 3, IMSLS_LINK_ALL, 0);
for (j=network->n_inputs; j < network->n_nodes ; j++)
{
    for (k=0; k < network->nodes[j].n_inLinks; k++)
    {
        wIdx = network->nodes[j].inLinks[k];
        /* set specific layer weights */
        if (network->nodes[j].layer_id == 1) {
            network->links[wIdx].weight = 0.25;
        } else if (network->nodes[j].layer_id == 2) {
            network->links[wIdx].weight = 0.33;
        }
    }
}

imsls_random_seed_set (12345);
stats = imsls_f_mlff_network_trainer (network, n_patterns - 10,
    n_nominal, n_continuous, nominal, 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_nominal; j++)
    {
        NOMINALOBS (0, j) = NOMINAL (i, j);
    }
    forecasts = imsls_f_mlff_network_forecast (network, n_nominal,
        n_continuous, nominalObs, continuousObs, 0);

    x = output[i];
    y = forecasts[0];
    printf("observation[%d] %8.4f Prediction %8.4f ", i, x, y);
    printf("Residual %8.4f \n", x - y);
}

imsls_f_mlff_network_free (network);
#undef NOMINAL
#undef NOMINALOBS
}

```

Output

NOTE: Because multiple optima are possible during training, the output of this example may vary by platform.

Predictions for Observations 90 to 100:

observation[90]	49.2975	Prediction	49.1823	Residual	0.1152
observation[91]	32.4351	Prediction	32.4410	Residual	-0.0059
observation[92]	37.8178	Prediction	37.7998	Residual	0.0179
observation[93]	38.5066	Prediction	38.4955	Residual	0.0111
observation[94]	48.6238	Prediction	48.5475	Residual	0.0763
observation[95]	37.6239	Prediction	37.6043	Residual	0.0196
observation[96]	41.5694	Prediction	41.5935	Residual	-0.0241
observation[97]	36.8290	Prediction	36.8038	Residual	0.0251
observation[98]	48.6908	Prediction	48.6110	Residual	0.0798
observation[99]	32.0481	Prediction	32.0631	Residual	-0.0150

mlff_classification_trainer



[more...](#)

Trains a multilayered feedforward neural network for classification.

Synopsis

```
#include <imsls.h>

float *imsls_f_mlff_classification_trainer (Imsls_f_NN_Network *network,
    int n_patterns, int n_nominal, int n_continuous, int classification[ ],
    int nominal[ ], float continuous[ ], ..., 0)
```

The type *double* function is `imsls_d_mlff_classification_trainer`.

Required Arguments

- Imsls_f_NN_Network* ***network** (Input/Output)
Pointer to a structure of type *Imsls_f_NN_Network* containing the feedforward network's architecture, including network weights and bias values. For more details, see [imsls_f_mlff_network](#). When network training is successful, the weights and bias values in **network** are replaced with the values calculated for the optimum trained network.
- int* **n_patterns** (Input)
Number of network training patterns.
- int* **n_nominal** (Input)
Number of nominal input attributes. Note that **n_nominal** + **n_continuous** must be equal to the total number of input attributes in the network, `network->n_inputs`. For more details, see [imsls_f_mlff_network](#).
- int* **n_continuous** (Input)
Number of continuous input attributes, including ordinal attributes encoded to percentages. Note that **n_nominal** + **n_continuous** must equal the total number of input attributes in the network, `network->n_inputs`. For more details, see [imsls_f_mlff_network](#).

int **classification**[] (Input)

Array of size **n_patterns** containing the target classifications for the training patterns. These must be numbered sequentially from 0 to $n_classes-1$, where $n_classes$ is the number of target categories. For binary classification problems, $n_classes = 2$. For other problems, $n_classes = n_outputs = network->n_outputs$. For more details, see [ims1s_f_mlff_network](#).

int **nominal**[] (Input)

Array of size **n_patterns** by **n_nominal** containing values for the nominal input attributes. The i -th row contains the nominal input attributes for the i -th training pattern. If **n_nominal** = 0, this argument is ignored.

float **continuous**[] (Input)

Array of size **n_patterns** by **n_continuous** containing values for the continuous input attributes. The i -th row contains the continuous input attributes for the i -th training pattern. If **n_continuous** = 0, this argument is ignored.

Return Value

An array of training statistics, containing six summary statistics from the classification neural network, organized as follows:

Element	Training Statistics
0	Minimum Cross-Entropy at the optimum.
1	Total number of Stage I iterations.
2	Minimum Cross-Entropy after Stage I training.
3	Total number of Stage II iterations.
4	Minimum Cross-Entropy after Stage II training.
5	Classification error rate from optimum network.

The classification error rate is calculated using the ratio $n_errors/n_patterns$, where n_errors is the number of patterns that are incorrectly classified using the trained neural network. For each training pattern, the probability that it belongs to each of the target classes is calculated from the trained network. A pattern is considered incorrectly classified if the classification probability for its target classification is not the largest among that pattern's classification probabilities.

A classification error of zero indicates that all training patterns are correctly classified into their target classifications. A value near one indicates that most patterns are not classified into their target classification.

If training is unsuccessful, **NULL** is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_mlff_classification_trainer (Imsls_f_NN_Network *network,
    int n_patterns, int n_nominal, int n_continuous, int classification[],
    int nominal[], float continuous[],
    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_WEIGHT_INITIALIZATION_METHOD, int method,
    IMSLS_LOGISTIC_TABLE,
    IMSLS_PREDICTED_CLASS, int **predicted_class,
    IMSLS_PREDICTED_CLASS_USER, int predicted_class[],
    IMSLS_GRADIENT, float **gradients,
    IMSLS_GRADIENT_USER, float gradients[],
    IMSLS_PREDICTED_CLASS_PROB, float **predicted_class_prob,
    IMSLS_PREDICTED_CLASS_PROB_USER, float predicted_class_prob[],
    IMSLS_CLASS_ERROR, float **class_error,
    IMSLS_CLASS_ERROR_USER, float class_error[],
    IMSLS_RETURN_USER, float trainStat[],
    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_patterns** and **n_epochs=1**. Stage I training is implemented using steepest ascent optimization and backward propagation for gradient calculations.

Default: **n_epochs=15, epoch_size = n_patterns**.

IMSLS_NO_STAGE_II, (Input)

Specifies no Stage II training is needed. Stage II training is implemented using Quasi-Newton optimization with numerical gradients.

Default: Stage II training is performed.

IMSLS_MAX_STEP, *float max_step* (Input)

Maximum allowable step size in the optimizer.

Default: **max_step = 10**.

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=1000**.

IMSLS_REL_FCN_TOL, *float rfcn_tol* (Input)

Relative function tolerance in the optimizer.

Default: **rfcn_tol = max(10⁻¹⁰, $\epsilon^{2/3}$)**, where ϵ is the machine precision.

rfcn_tol = max(10⁻²⁰, $\epsilon^{2/3}$) in double.

IMSLS_GRAD_TOL, *float grad_tol* (Input)

Scaled gradient tolerance in the optimizer.

Default: **grad_tol = $\epsilon^{1/2}$** , where ϵ is the machine precision.

grad_tol = $\epsilon^{1/3}$ in double.

IMSLS_TOLERANCE, *float tolerance* (Input)

Absolute accuracy tolerance for the entropy. If the network entropy for an epoch during Stage I training falls below **tolerance**, the network is considered optimized, training is halted and the network with the minimum entropy is returned.

Default: $\text{tolerance} = \epsilon^{1/3}$, where ϵ is the machine precision $\text{tolerance} = \epsilon^{2/3}$ in double.

IMSLS_PRINT, (Input)

Intermediate results are printed during network training.

Default: No printing is performed.

IMSLS_WEIGHT_INITIALIZATION_METHOD, *int method* (Input)

The method to use for initializing network weights prior to network training. One of the following five values is accepted:

method	Algorithm
IMSLS_EQUAL	Equal weights.
IMSLS_RANDOM	Random weights.
IMSLS_PRINCIPAL_COMPONENTS	Principal Component Weights.
IMSLS_DISCRIMINANT	Discriminant Analysis Weights.
IMSLS_NN_NETWORK	No initialization method will be performed. Weights in <i>Imsls_f_NN_Network</i> structure <code>network</code> will be used instead.

Default: `method = IMSLS_RANDOM`.

IMSLS_LOGISTIC_TABLE, (Input)

If this option is selected, during Stage I optimization all logistic activation functions in the hidden layers are calculated using a table lookup approximation to the logistic function. This reduces the time for Stage I training with logistic activation. However, during Stage II optimization this setting is ignored.

Default: All logistic activations are calculated without table lookup.

IMSLS_PREDICTED_CLASS, *int **predicted_class* (Output)

The address of a pointer to an array of size `n_patterns` containing the predicted classification for each training pattern.

IMSLS_PREDICTED_CLASS_USER, *int predicted_class[]* (Output)

Storage for array `predicted_class` provided by user. See **IMSLS_PREDICTED_CLASS**.

IMSLS_GRADIENT, *float **gradients* (Output)

The address of a pointer to an array of size `network->n_links + network->n_nodes - network->n_inputs` containing the gradients for each weight in the optimum network.

IMSLS_GRADIENT_USER, *float gradients[]* (Output)

Storage for array `gradients` provided by user. See **IMSLS_GRADIENT**.

IMSLS_PREDICTED_CLASS_PROB, *float **predicted_class_prob* (Output)

The address of a pointer to an array of size `n_patterns` by `n_classes`, where `n_classes` is the number of target classes in the network. For binary classification problems, `n_classes = 2`, but for all other problems `n_classes = n_outputs`, where `n_outputs` is the number of outputs in the network, `network->n_outputs`. The values of the *i*-th row are the predicted probabilities associated with the target classes. For binary classification, `predicted_class_prob[i]` is the predicted probability that the *i*-th pattern is associated with `class = 0`. For other classification problems values in the *i*-th row of `predicted_class_prob` are the predicted probabilities that this pattern belongs to each of the target classes.

IMSLS_PREDICTED_CLASS_PROB_USER, *float predicted_class_prob[]* (Output)

Storage for array `predicted_class_prob` provided by user. See **IMSLS_PREDICTED_CLASS_PROB** for a description.

IMSLS_CLASS_ERROR *float, **class_error* (Output)

The address of a pointer to an array with `n_patterns` containing the classification probability errors for each pattern in the training data. The classification error for the *i*-th training pattern is equal to `1-predicted_class[k]` where `k=classification[i]`.

IMSLS_CLASS_ERROR_USER, *float class_error[]* (Output)

Storage for array `class_error` provided by user. See **IMSLS_CLASS_ERROR** for a description.

IMSLS_RETURN_USER, *float trainStat[]* (Output)

User-supplied array of length 6. Upon completion, `trainStat` contains the return array of training statistics.

Description

Function `imsls_f_mlff_classification_trainer` trains a multilayered feedforward neural network for classifying patterns. It returns training summaries, the classification probabilities associated with training patterns, their classification errors, the optimum network weights and gradients. Linkages among perceptrons allow for skipped layers, including linkages between inputs and output perceptrons. Except for output perceptrons, the linkages and activation function for each perceptron 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](#).

Binary classification is handled differently from classification problems involving more than two classes. Binary classification problems only have two target classes, which must be coded as either zero or one. This is represented using a single network output with logistic activation. The output is an estimate of the probability that the pattern belongs to `class = 0`. The probability associated with `class = 1` can be calculated from the relationship $P(\text{class} = 1) = 1 - P(\text{class} = 0)$.

Networks designed to classify patterns into more than two categories use one output for each target class, i.e. $n_classes = n_outputs$. The first output predicts $P(class = 0)$, the second $P(class = 1)$, etc. All output perceptrons are normalized using softmax activation. This ensures that the estimated class probabilities are between zero and one, and that they always sum to one.

Training Patterns

Neural network training patterns consist of the following three types of data:

1. nominal input attributes
2. continuous input attributes, including encoded ordinal attributes,
3. pattern classifications numbered 0, 1, ..., $n_classes - 1$

The first data type, nominal data, contains the encoding of nominal input attributes, if any. Nominal input attributes must be encoded into multiple columns for network input. Although not required, binary encoding is typically used to create these input columns. Binary 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 several nominal attributes appear in the data, then each attribute is associated with several columns, one for each of its classes.

The `ims1s_f_unsupervised_nominal_filter` can be used to generate these columns. For a nominal attribute with m classes, `ims1s_f_unsupervised_nominal_filter` returns an $n_patterns$ by m matrix. Each column of this matrix consists of zeros and ones. The column value is set to zero if the pattern is not associated with this classification; otherwise, the value is set to one indicating that this pattern is associated with this classification.

Consider an example with one nominal variable that has two classes: *male* and *female* and five training patterns: male, male, female, male, female. With binary encoding, the following matrix is used as nominal network input to represent these patterns:

$$\text{nominal} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Continuous input attribute data, including ordinal data encoded to cumulative percentages, are passed to this routine in a separate floating point array, `continuous`. The number of rows in this array is $n_patterns$. The number of columns is $n_continuous$. If the continuous input attributes have widely different ranges, then typ-

ically it is advantageous to scale these attributes before using them in network training. The routine `imsls_f_scale_filter` can be used for scaling continuous input attributes before using it in network training. Ordinal attributes can be encoded using `imsls_f_unsupervised_ordinal_filter`.

It is important to note that if input attributes are encoded or scaled for network training, then the network weights are calculated for that encoding and scaling. Subsequent pattern classifications using these weights must also use the identical encoding and scaling used during training.

Training pattern classification targets are stored in the one-dimensional integer array `classification`. The i -th value in this array is the class assignment for the i -th training pattern. Class assignments must be represented using the integers $0, 1, \dots, n_classes - 1$. This encoding is arbitrary, but it should be consistent. For example, if the class assignments correspond to the colors red, white and blue, then they must be encoded as zero, one, and two. However, it is arbitrary whether red gets assigned to `class = 0, 1` or `2` provided that assignment is used for every pattern.

Network Configuration

The network configuration consists of the following:

- number of inputs and outputs,
- number of hidden layers,
- description of the number of perceptrons in each layer,
- description of the linkages among the perceptrons, and
- initial values for network weights, including bias weights.

This description is passed into `imsls_f_mlff_classification_trainer` using the structure `Imsls_f_N_N_Network`. See `imsls_f_mlff_network`.

Training Efficiency

INITIAL NETWORK WEIGHTS: The training efficiency determines the speed of network training. This is controlled by several factors. One of the most important factors is the initial weights used by the optimization algorithm. By default, these are set randomly. Other options can be specified through the optional argument `IMSLS_INITIALIZE_WEIGHTS_METHOD`. See `imsls_f_mlff_initialize_weights` for a detailed description of the available initialization methods.

Initial weights are scaled to reduce the possibility of perceptron saturation during the initial phases of network training. Saturation occurs when initial perceptron potential calculations are so large, or so small, that the activation calculation for a potential is driven to the largest or smallest possible values that can be represented on the computer in the stated precision (single or double). If saturation occurs, warning messages may appear indicating that network training did not converge to an optimum solution.

The scaled initial weights are modified prior to every epoch by adding noise to these base values. The noise component is uniformly distributed over the interval $[-0.5, +0.5]$.

SCALING INPUTS: Although automatic scaling of network weights protects against saturation during initial training iterations, the training algorithm can push the weights into regions that may cause saturation. Typically this occurs when input attributes have widely different scaling. For that reason, it is recommended to also scale all continuous input attributes to z-scores or a common interval, such as $[-1, +1]$. The routine `imsls_f_scale_filter` can be used to scale continuous input attributes to z-scores or a common interval.

LOGISTIC CALCULATIONS: If Stage I training is slow, the optional argument `IMSLS_LOGISTIC_TABLE` can reduce this time by using a table lookup for calculating the logistic activation function in hidden layer perceptrons. This option is ignored during Stage II training. If Stage II training is used, then weights for the optimum network will be calculated using exact calculations for any logistic activation functions. If Stage II training is not used and the `IMSLS_LOGISTIC_TABLE` option is invoked, care must be taken to ensure that this option is also used for any network classification predictions using `imsls_f_mlff_pattern_classification`.

NUMBER OF EPOCHS AND EPOCH SIZE: To ensure that a globally optimum network results from the training, several training sessions are conducted and compared. Each session is referred to as an epoch. The training for each epoch is conducted using all of the training patterns or a random sample of all available patterns.

Both the number of epochs and epoch size can be set using the `IMSLS_STAGE_I` option. By default the number of epochs during Stage I training is 15 and the epoch size is equal to the total number of training patterns. Increasing the number of epochs increases the training time, but it can result in a more accurate classification network.

During Stage I training, the network entropy is calculated after each epoch. If that value is smaller than `tolerance` Stage I training will stop since it is assumed that a network with entropy that low is acceptably accurate, and it is not necessary to continue training. The value for `tolerance` can be set using the `IMSLS_TOLERANCE` option. Setting this to a larger value, such as 0.001, is useful for initially evaluating alternate network architectures.

NETWORK SIZE AND VALIDATION: The network architecture, the number of perceptrons and network layers, also play a key role in network training. Larger networks with many inputs and perceptrons have a larger number of weights. Large networks can provide very accurate classifications, driving the misclassification error rate for the training patterns to zero. However networks with too many weights can take too long to train, and can be inaccurate for classifying patterns not adequately represented among the training patterns.

A starting point is to ensure the total number of network weights is approximately equal to the number of training patterns. A trained network of this size typically has a low misclassification error rate when calculated for the training patterns. That is, it is able to accurately reproduce the training data. However, it might be inaccurate for classifying other patterns.

One approach to this validation is to split the total number of training patterns into two or more subsets then train the network using only one of the subsets and classify the remaining data using the trained network. The misclassification error rate for the data not used in training will be a better estimate of the true classification error rate for this network.

However, this approach to validation is only possible when the number of training patterns is large.

Output

Output from `imsls_f_mlff_classification_trainer` consists of classification probabilities calculated for each training pattern, a classification error array for these patterns, predicted classifications, weights and their associated gradients for the trained network, and the training statistics. The `imsls_f_NN_Network` structure is automatically updated with the weights, gradients and bias values for use as input to `imsls_f_mlff_pattern_classification`.

The trained network can be saved and retrieved using `imsls_f_mlff_network_write` and `imsls_f_mlff_network_read`. For more details about the weights and bias values, see [Table 50](#). These functions allow the functions of network training and classification to be implemented in different languages. Networks trained in CNL can be transferred into other IMSL libraries, such as JMSL and C# Numerical Library, for pattern classification.

Examples

Example 1

This example trains a three-layer network using 48 training patterns with two nominal and two continuous input attributes. The first nominal attribute has three classifications and the second has four. Classifications for the nominal attributes are encoded using `imsls_f_unsupervised_nominal_filter`. This function uses binary encoding, generating a total of 7 input attributes to represent the two nominal attributes. The two additional continuous attributes increase the total number of network inputs to 9.

In this example, the target classification is binary, either zero or one. The continuous input attribute was scaled to the interval [0,1].

The structure of the network consists of nine input attributes in the input layer and three other layers. There are three perceptrons in the 1st hidden layer, and two in the 2nd. Since the classification target in this example is binary, there is only one perceptron in the output layer.

All perceptrons use the logistic function for activation, including the output perceptron. Since logistic activation values are always between 0 and 1, the output from this network can be interpreted directly as the estimated probability, $P(0)$, that a pattern belongs to target classification 0.

The following figure illustrates this structure:

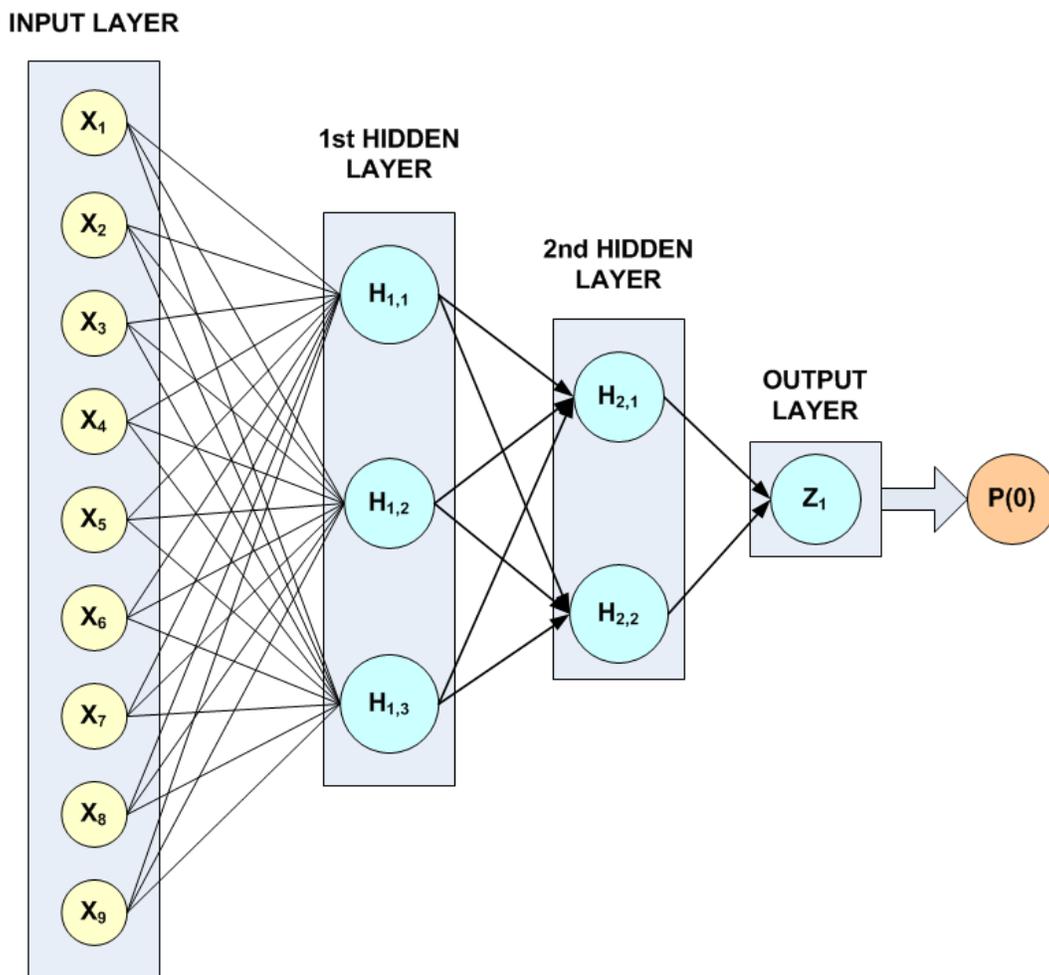


Figure 36, A Binary 3-layer, Classification Network with 7 Inputs and 6 Perceptrons

There are a total of 41 weights in this network. Six are bias weights and the remaining 35 are the weights for the input links to every perceptron, e.g. $35 = 9 \cdot 3 + 3 \cdot 2 + 2$.

Printing is turned on to show progress during the training session.

```

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

int main(){
    float *trainStats;
    int i, j, k, m;
    int n_patterns =48; /* # of training patterns */
    int n_inputs =9; /* 2 nominal (7 classes) and 2 continuous */
    int n_nominal =7; /* 2 attributes with 3 and 4 classes each */
    int n_continuous =2; /* 2 continuous input attributes */
    int n_outputs =1; /* binary classification */
    int classification[48];
    int nominalAtt[48*7];
    int n_cat = 2;
    int nomTempIn[48], *nomTempOut, nClass;
    float inputData[5*48] =
    {
        0.00, 0.00, 0, 0, 0, 0.02, 0.02, 0, 1, 0, 0.04, 0.04, 0, 2, 0,
        0.06, 0.06, 0, 3, 0, 0.08, 0.08, 1, 0, 0, 0.10, 0.10, 1, 1, 0,
        0.12, 0.12, 1, 2, 0, 0.14, 0.14, 1, 3, 0, 0.16, 0.16, 2, 0, 0,
        0.18, 0.18, 2, 1, 0, 0.20, 0.20, 2, 2, 0, 0.22, 0.22, 2, 3, 0,
        0.24, 0.28, 0, 0, 0, 0.26, 0.30, 0, 1, 0, 0.28, 0.32, 0, 2, 0,
        0.30, 0.34, 0, 3, 0, 0.32, 0.36, 1, 0, 0, 0.34, 0.38, 1, 1, 0,
        0.36, 0.40, 1, 2, 0, 0.38, 0.42, 1, 3, 0, 0.40, 0.44, 2, 0, 0,
        0.42, 0.46, 2, 1, 0, 0.44, 0.48, 2, 2, 0, 0.46, 0.50, 2, 3, 0,
        0.52, 0.48, 0, 0, 0, 0.54, 0.50, 0, 1, 1, 0.56, 0.52, 0, 2, 1,
        0.58, 0.54, 0, 3, 1, 0.60, 0.56, 1, 0, 1, 0.62, 0.58, 1, 1, 1,
        0.64, 0.60, 1, 2, 1, 0.66, 0.62, 1, 3, 1, 0.68, 0.64, 2, 0, 0,
        0.70, 0.66, 2, 1, 0, 0.72, 0.68, 2, 2, 0, 0.74, 0.70, 2, 3, 0,
        0.76, 0.76, 0, 0, 1, 0.78, 0.78, 0, 1, 1, 0.80, 0.80, 0, 2, 1,
        0.82, 0.82, 0, 3, 1, 0.84, 0.84, 1, 0, 1, 0.86, 0.86, 1, 1, 1,
        0.88, 0.88, 1, 2, 1, 0.90, 0.90, 1, 3, 1, 0.92, 0.92, 2, 0, 0,
        0.94, 0.94, 2, 1, 0, 0.96, 0.96, 2, 2, 0, 0.98, 0.98, 2, 3, 0
    };
};
float contAtt[2*48];
float *classProb;
char *colLabels[] = {"Pattern", "Class=0", "Class=1"};
Imsls_f_NN_Network *network;

printf("\n*****\n");
printf("** BINARY CLASSIFICATION EXAMPLE *\n");
printf("*****\n\n");

imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

/* Setup Continuous Input Attributes and
 * Classification Target Arrays
 */
for(i=0; i<n_patterns; i++){
    /* Assign input to array for continuous input attributes */
    contAtt[2*i] = inputData[i*5];
    contAtt[2*i+1] = inputData[i*5+1];
    /* Assign input to classification target array*/
    classification[i] = (int) inputData[i*5+4];
}

/* Setup Nominal Input Attributes Using Binary Encoding */

```

```

m=0;
for(i=0; i<n_cat; i++){
    for(j=0; j<n_patterns; j++){
        nomTempIn[j] = (int) inputData[j*5+n_continuous+i]+1;
    }
    nomTempOut = imsls_unsupervised_nominal_filter(n_patterns,
        &nClass, nomTempIn, 0);
    for(k=0; k<nClass; k++){
        for(j=0; j<n_patterns; j++){
            nominalAtt[j*n_nominal+m] = nomTempOut[j*nClass+k];
        }
        m++;
    }
    imsls_free(nomTempOut);
}

printf("\t TRAINING PATTERNS\n");
printf("\tY  N1 N2  Z1  Z2 \n");
for(i=0; i<n_patterns; i++){
    j = (int) inputData[i*5+2];
    k = (int) inputData[i*5+3];
    printf("\t%d  %d  %d  %g  %g \n", classification[i], j, k,
        contAtt[i*2], contAtt[i*2+1]);
}

printf("\n");
network = imsls_f_mlff_network_init(n_inputs, n_outputs);
imsls_f_mlff_network(network,
    IMSLS_CREATE_HIDDEN_LAYER, 3,
    0);
imsls_f_mlff_network(network,
    IMSLS_CREATE_HIDDEN_LAYER, 2,
    IMSLS_LINK_ALL,
    0);

/* Note the following statement is for repeatable output */
imsls_random_seed_set(5555);
/* Train Classification Network */
printf("STARTING NETWORK TRAINING\n");
trainStats = imsls_f_mlff_classification_trainer (network,
    n_patterns, n_nominal, n_continuous, classification,
    nominalAtt, contAtt,
    IMSLS_PRINT,
    IMSLS_PREDICTED_CLASS_PROB, &classProb,
    0);

/* Print class predictions*/
imsls_f_write_matrix("Predicted Classification Probabilities",
    n_patterns, n_outputs , classProb,
    IMSLS_ROW_NUMBER,
    IMSLS_COL_LABELS, colLabels,
    0);
}

```

Output

Notice that although by default the maximum number of epoch iterations in Stage I training is 15, in this case Stage I optimization is halted after the first epoch. This occurs because the minimum entropy for that epoch is less than the default tolerance.

```
*****
* BINARY CLASSIFICATION EXAMPLE *
*****
```

```

      TRAINING PATTERNS
      Y  N1 N2  Z1  Z2
0  0  0  0  0  0
0  0  1  0.02  0.02
0  0  2  0.04  0.04
0  0  3  0.06  0.06
0  1  0  0.08  0.08
0  1  1  0.1  0.1
0  1  2  0.12  0.12
0  1  3  0.14  0.14
0  2  0  0.16  0.16
0  2  1  0.18  0.18
0  2  2  0.2  0.2
0  2  3  0.22  0.22
0  0  0  0.24  0.28
0  0  1  0.26  0.3
0  0  2  0.28  0.32
0  0  3  0.3  0.34
0  1  0  0.32  0.36
0  1  1  0.34  0.38
0  1  2  0.36  0.4
0  1  3  0.38  0.42
0  2  0  0.4  0.44
0  2  1  0.42  0.46
0  2  2  0.44  0.48
0  2  3  0.46  0.5
0  0  0  0.52  0.48
1  0  1  0.54  0.5
1  0  2  0.56  0.52
1  0  3  0.58  0.54
1  1  0  0.6  0.56
1  1  1  0.62  0.58
1  1  2  0.64  0.6
1  1  3  0.66  0.62
0  2  0  0.68  0.64
0  2  1  0.7  0.66
0  2  2  0.72  0.68
0  2  3  0.74  0.7
1  0  0  0.76  0.76
1  0  1  0.78  0.78
1  0  2  0.8  0.8
1  0  3  0.82  0.82
1  1  0  0.84  0.84
1  1  1  0.86  0.86
1  1  2  0.88  0.88
1  1  3  0.9  0.9
0  2  0  0.92  0.92
0  2  1  0.94  0.94

```

```
0 2 2 0.96 0.96
0 2 3 0.98 0.98
```

STARTING NETWORK TRAINING

TRAINING PARAMETERS:

```
Stage II Opt. = 1
n_epochs     = 15
epoch_size   = 48
maxIterations = 1000
maxFunctionEval = 1000
maxStep      = 10.000000
functionTol  = 2.42218e-005
gradientTol  = 0.000345267
accuracy     = 0.000345267
n_inputs     = 9
n_continuous = 2
n_nominal    = 7
n_classes    = 2
n_outputs    = 1
n_patterns   = 48
n_layers     = 3
n_perceptrons = 6
n_weights    = 41
```

STAGE I TRAINING STARTING

```
Stage I: Epoch 1 - Cross-Entropy Error = 1.03973e-005 (Iterations=58)
(CPU Min.=0.000260)
Stage I Training Converged at Epoch = 1
```

STAGE I FINAL CROSS-ENTROPY ERROR = 0.000010 (CPU Min.=0.000260)

OPTIMUM WEIGHTS AFTER STAGE I TRAINING:

```
weight[0] = -0.634574    weight[1] = 3.10432
weight[2] = 0.753153    weight[3] = 3.26621
weight[4] = 0.873874    weight[5] = 0.564623
weight[6] = 0.574684    weight[7] = 2.03545
weight[8] = 2.11041     weight[9] = 4.53693
weight[10] = 3.17946    weight[11] = -10.908
weight[12] = -1.12353   weight[13] = 0.915452
weight[14] = -0.737025  weight[15] = -0.44498
weight[16] = 11.1242    weight[17] = 8.99682
weight[18] = 2.17726    weight[19] = 1.82712
weight[20] = -8.49784   weight[21] = -3.14366
weight[22] = -0.0180527 weight[23] = 0.618158
weight[24] = -0.243258  weight[25] = 13.5949
weight[26] = 11.9534    weight[27] = -4.83385
weight[28] = -18.9217   weight[29] = -9.0807
weight[30] = -7.01863   weight[31] = -15.0597
weight[32] = -17.7305   weight[33] = 23.6268
weight[34] = 11.2716    weight[35] = -6.76151
weight[36] = -13.0134   weight[37] = -14.1513
weight[38] = 17.8283    weight[39] = 21.7138
weight[40] = -19.8484
```

STAGE I TRAINING CONVERGED

STAGE I CROSS-ENTROPY ERROR = 0.000010

0 PATTERNS OUT OF 48 INCORRECTLY CLASSIFIED

GRADIENT AT THE OPTIMUM WEIGHTS

-->g[0] =	0.000000	weight[0] =	-0.634574
-->g[1] =	-0.000001	weight[1] =	3.104319
-->g[2] =	0.000000	weight[2] =	0.753153
-->g[3] =	-0.000001	weight[3] =	3.266206
-->g[4] =	0.000000	weight[4] =	0.873874
-->g[5] =	0.000000	weight[5] =	0.564623
-->g[6] =	0.000000	weight[6] =	0.574684
-->g[7] =	-0.000000	weight[7] =	2.035449
-->g[8] =	-0.000000	weight[8] =	2.110413
-->g[9] =	0.000000	weight[9] =	4.536931
-->g[10] =	-0.000005	weight[10] =	3.179461
-->g[11] =	0.000000	weight[11] =	-10.908046
-->g[12] =	-0.000005	weight[12] =	-1.123529
-->g[13] =	0.000000	weight[13] =	0.915452
-->g[14] =	0.000000	weight[14] =	-0.737025
-->g[15] =	0.000000	weight[15] =	-0.444980
-->g[16] =	-0.000003	weight[16] =	11.124193
-->g[17] =	-0.000003	weight[17] =	8.996821
-->g[18] =	0.000000	weight[18] =	2.177265
-->g[19] =	-0.000004	weight[19] =	1.827117
-->g[20] =	0.000000	weight[20] =	-8.497839
-->g[21] =	-0.000004	weight[21] =	-3.143664
-->g[22] =	0.000000	weight[22] =	-0.018053
-->g[23] =	0.000000	weight[23] =	0.618158
-->g[24] =	0.000000	weight[24] =	-0.243258
-->g[25] =	-0.000002	weight[25] =	13.594892
-->g[26] =	-0.000002	weight[26] =	11.953360
-->g[27] =	0.000001	weight[27] =	-4.833848
-->g[28] =	0.000000	weight[28] =	-18.921690
-->g[29] =	0.000000	weight[29] =	-9.080699
-->g[30] =	0.000001	weight[30] =	-7.018632
-->g[31] =	0.000001	weight[31] =	-15.059658
-->g[32] =	0.000000	weight[32] =	-17.730463
-->g[33] =	-0.000010	weight[33] =	23.626806
-->g[34] =	-0.000010	weight[34] =	11.271611
-->g[35] =	-0.000001	weight[35] =	-6.761511
-->g[36] =	-0.000005	weight[36] =	-13.013445
-->g[37] =	-0.000004	weight[37] =	-14.151301
-->g[38] =	0.000001	weight[38] =	17.828314
-->g[39] =	0.000001	weight[39] =	21.713770
-->g[40] =	-0.000010	weight[40] =	-19.848421

Training Completed - leaving training engine (CPU Min.=0.000260)

Predicted Classification Probabilities

Pattern	Class=0
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

11	1
12	1
13	1
14	1
15	1
16	1
17	1
18	1
19	1
20	1
21	1
22	1
23	1
24	1
25	1
26	0
27	0
28	0
29	0
30	0
31	0
32	0
33	1
34	1
35	1
36	1
37	0
38	0
39	0
40	0
41	0
42	0
43	0
44	0
45	1
46	1
47	1
48	1

Example 2

Fisher's (1936) Iris data is often used for benchmarking discriminant analysis and classification solutions. It is part of the IMSL data sets and consists of the following continuous input attributes and classification target:

Continuous Attributes – X_1 (sepal length), X_2 (sepal width), X_3 (petal length), and X_4 (petal width)

Classification Target (Iris Type) – Setosa, Versicolour or Virginica.

These data consist of 150 patterns. Since all pattern input attributes are continuous, linear discriminant analysis can be used for classifying these patterns, see [Example 1](#) of `imsls_f_discriminant_analysis`. Linear discriminant analysis is able to correctly classify 98% of the training patterns. In this example, the simple neural network illustrated in the following figure is able to achieve 100% accuracy.

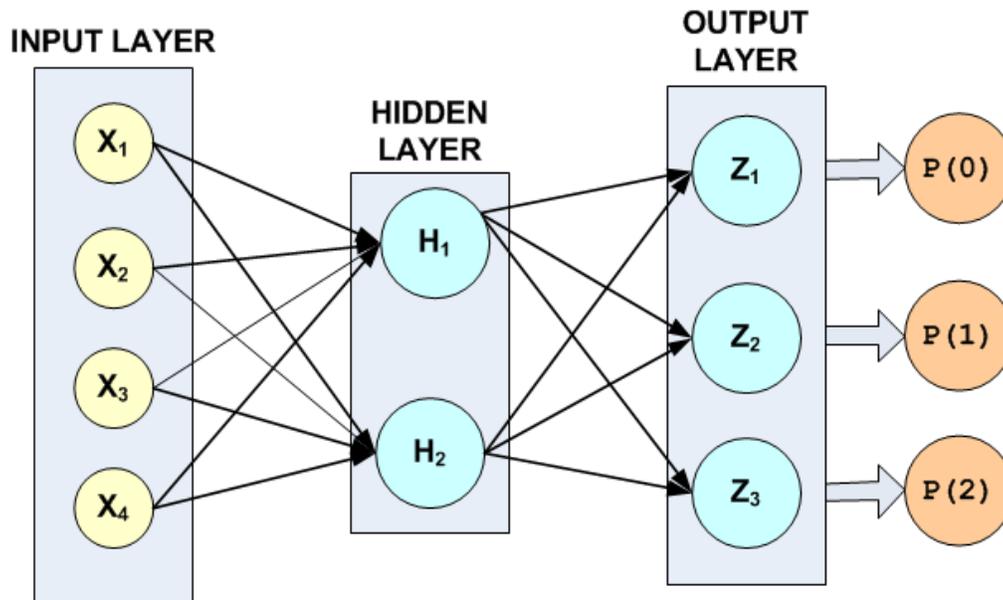


Figure 37, A 2-layer, Classification Network with 4 Inputs 5 Perceptrons and a Target Classification with 3 Classes

The hidden layer in this example consists of only two perceptrons with logistic activation. Since the target attribute in this example has three classes, the network output layer consists of three perceptrons, one for each class.

There are a total of 19 weights in this network. Fourteen of the weights are assigned to the input links, i.e., $4 \times 2 + 2 \times 3 = 14$. The last five weights are the bias weights for each of the five network perceptrons. All weights were initialized using principal components, i.e. `method = IMSLS_PRINCIPAL_COMPONENTS`.

Although in these data the continuous attributes have similar ranges, they were scaled using z-score scaling to make network training more efficient. For more details, see [imsls_f_scale_filter](#).

For non-binary classification problems, `imsls_f_mlff_classification_trainer` uses softmax activation for output perceptrons. This ensures that the estimates of the classification probabilities sum to one, i.e.

$$\sum_{i=0}^2 P(\text{class} = i) = 1$$

Note that the default setting for `IMSLS_MAX_STEP` was changed from 10 to 1000. The default setting converged to a network with 100% classification accuracy. However, the following warning message appeared:

```

*** WARNING Error IMSLS_UNBOUNDED from imsls_d_mlff_classification_trainer.
*** Five consecutive steps of length "max_step" have been taken;
  
```

```

*** either the function is unbounded below, or has a finite
*** asymptote in some direction or the maximum allowable step size
*** "max_step" is too small.

```

In addition, the number of iterations used for each epoch were well below the default maximum (1000), and the gradients at the optimum solution for this network were not zero.

```

STAGE I TRAINING STARTING
Stage I: Epoch 1 - Cross-Entropy Error = 5.50552 (Iterations=40) (CPU Min.=0.000260)
Stage I: Epoch 2 - Cross-Entropy Error = 5.65875 (Iterations=69) (CPU Min.=0.000260)
Stage I: Epoch 3 - Cross-Entropy Error = 4.83886 (Iterations=81) (CPU Min.=0.000260)
Stage I: Epoch 4 - Cross-Entropy Error = 5.94979 (Iterations=108) (CPU Min.=0.000521)
Stage I: Epoch 5 - Cross-Entropy Error = 5.54461 (Iterations=47) (CPU Min.=0.000260)
Stage I: Epoch 6 - Cross-Entropy Error = 6.04163 (Iterations=51) (CPU Min.=0.000260)
Stage I: Epoch 7 - Cross-Entropy Error = 5.95148 (Iterations=151) (CPU Min.=0.000521)
Stage I: Epoch 8 - Cross-Entropy Error = 5.5646 (Iterations=55) (CPU Min.=0.000260)
Stage I: Epoch 9 - Cross-Entropy Error = 5.94914 (Iterations=442) (CPU Min.=0.001563)
Stage I: Epoch 10 - Cross-Entropy Error = 5.94381 (Iterations=271) (CPU Min.=0.001302)
Stage I: Epoch 11 - Cross-Entropy Error = 5.41955 (Iterations=35) (CPU Min.=0.000000)
Stage I: Epoch 12 - Cross-Entropy Error = 6.01766 (Iterations=48) (CPU Min.=0.000260)
Stage I: Epoch 13 - Cross-Entropy Error = 4.20551 (Iterations=112) (CPU Min.=0.000521)
Stage I: Epoch 14 - Cross-Entropy Error = 5.95085 (Iterations=103) (CPU Min.=0.000260)
Stage I: Epoch 15 - Cross-Entropy Error = 5.9596 (Iterations=55) (CPU Min.=0.000260)
Stage I: Epoch 16 - Cross-Entropy Error = 5.96131 (Iterations=59) (CPU Min.=0.000260)
Stage I: Epoch 17 - Cross-Entropy Error = 4.83231 (Iterations=74) (CPU Min.=0.000260)
Stage I: Epoch 18 - Cross-Entropy Error = 17.1345 (Iterations=30) (CPU Min.=0.000260)
Stage I: Epoch 19 - Cross-Entropy Error = 5.95569 (Iterations=92) (CPU Min.=0.000260)
Stage I: Epoch 20 - Cross-Entropy Error = 3.15336 (Iterations=46) (CPU Min.=0.000260)
GRADIENT AT THE OPTIMUM WEIGHTS
-->g[0] = 0.675632 weight[0] = 0.075861
-->g[1] = -0.953480 weight[1] = -0.078585
-->g[2] = 1.065184 weight[2] = 2.841074
-->g[3] = 0.535531 weight[3] = -0.941049
-->g[4] = -0.019011 weight[4] = -10.638772
-->g[5] = 0.001459 weight[5] = -14.573394
-->g[6] = -0.031098 weight[6] = 6.037813
-->g[7] = -0.035305 weight[7] = 72.382073
-->g[8] = 0.011015 weight[8] = -73.564433
-->g[9] = 0.000000 weight[9] = -14.853988
-->g[10] = -0.074332 weight[10] = 2.057743
-->g[11] = 0.000522 weight[11] = -39.952435
-->g[12] = 0.063316 weight[12] = 73.164141
-->g[13] = -0.000522 weight[13] = 57.065975
-->g[14] = 1.279914 weight[14] = -0.661036
-->g[15] = -0.043097 weight[15] = -61.171894
-->g[16] = 0.003227 weight[16] = 24.236722
-->g[17] = -0.108146 weight[17] = 14.968424
-->g[18] = 0.104919 weight[18] = -39.079343

```

Combined, this information suggests that either the default tolerances were too high or the maximum step size was too small. As shown in the output below, when the maximum step size was changed to 1000, the number of iterations increased, the gradients went to zero and the warning message for step size disappeared.

```

#include <stdio.h>
#include <imsls.h>
/* *****
* Two Layer Feed-Forward Network with 4 inputs, all
* continuous, and 3 classification categories.
*

```

```

* This is a well known database to be found in the pattern
* recognition literature. Fisher's paper is often cited.
* The data set contains 3 classes of 50 instances each,
* where each class refers to a type of iris plant. One class is
* linearly separable from the other 2; the latter are NOT linearly
* separable from each other.
*
* Predicted attribute: class of iris plant.
* 1=Iris Setosa, 2=Iris Versicolour, and 3=Iris Virginica
*
* Input Attributes (4 Continuous Attributes)
* X1: Sepal length,
* X2: Sepal width,
* X3: Petal length,
* and X4: Petal width
*
*****/
int main(){
    float *predicted_class_prob, *class_error, *trainStats;
    int i, j;
    int n_patterns = 150;
    int n_inputs = 4; /* all continuous inputs*/
    int n_nominal = 0; /* no nominal */
    int n_continuous = 4;
    int n_outputs = 3;
    int *nominalAtt=NULL;
    int *predicted_class;
    int act_fcn[3] = {1, 1, 1};
    int classification[150];
    float unscaledX[150], scaledX[150];
    float contAtt[4*150];
    float *irisData;
    float mean[4], s[4], center, spread;
    double startTime, endTime;
    char *colLabels[] = {"Pattern", "Class=0", "Class=1", "Class=2"};
    char prtLabel[] =
        "Predicted_Class | P(0) P(1) P(2) | Class_Error";
    char dashes[] =
        "-----";
    char *filename = "iris_classification.txt";
    /* Declare mlff network structure */
    Imsls_f_NN_Network *network;

    printf("*****\n");
    printf("* IRIS CLASSIFICATION EXAMPLE *\n");
    printf("*****\n");

    imsls_omp_options(IMSL5_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    /*
    * irisData[]: The raw data matrix. This is a 2-D matrix with 150
    * rows and 5 columns. The last 4 columns are the
    * continuous input attributes and the 1st column is
    * the classification category (1-3). These data
    * contain no nominal input attributes.
    */
    irisData = imsls_f_data_sets(3,0);

    /*
    * Setup the continuous attribute input array, contAtt[], and the

```

```

* network target classification array, classification[], using the
* above raw data matrix.
*/
for(i=0; i < n_patterns; i++){
    classification[i] = (int)irisData[i*5]-1;
    for(j=1; j<5; j++){
        contAtt[i*4+j-1] = irisData[i*5+j];
    }
}
/* Scale continuous input attributes using z-score method */
for(j=0; j<n_continuous; j++){
    for(i=0; i<n_patterns; i++)
        unscaledX[i] = contAtt[i*n_continuous+j];
    imsls_f_scale_filter(n_patterns, unscaledX, 2,
        IMSLS_RETURN_USER, scaledX,
        IMSLS_RETURN_CENTER_SPREAD, &center, &spread, 0);
    for(i=0; i<n_patterns; i++)
        contAtt[i*n_continuous+j] = scaledX[i];
    mean[j] = center;
    s[j] = spread;
}
printf("Scale Parameters: \n");
for(j=0; j<n_continuous; j++){
    printf("Var %d Mean = %f S = %f \n",j+1, mean[j], s[j]);
}

network = imsls_f_mlff_network_init(n_inputs, n_outputs);
imsls_f_mlff_network(network, IMSLS_CREATE_HIDDEN_LAYER, 2,
    IMSLS_LINK_ALL, 0);

/* Note the following statement is for repeatable output */
imsls_random_seed_set(5555);
/* Train Classification Network */
startTime = imsls_ctime();
trainStats = imsls_f_mlff_classification_trainer (network,
    n_patterns,
    n_nominal,
    n_continuous,
    classification,
    nominalAtt,
    contAtt,
    IMSLS_PRINT,
    IMSLS_STAGE_I, 20, 150,
    IMSLS_WEIGHT_INITIALIZATION_METHOD, IMSLS_PRINCIPAL_COMPONENTS,
    IMSLS_MAX_STEP, 1000.0,
    IMSLS_PREDICTED_CLASS, &predicted_class,
    IMSLS_PREDICTED_CLASS_PROB, &predicted_class_prob,
    IMSLS_CLASS_ERROR, &class_error,
    0);
endTime = imsls_ctime();
printf("%s\n", dashes);
printf("Minimum Cross-Entropy Error: %g\n", trainStats[0]);
printf("Classification Error Rate: %f\n", trainStats[5]);
printf("Execution Time (Sec.): %f\n\n", (endTime-startTime));

printf("%s\n",prtLabel);
printf("%s\n",dashes);
for(i=0; i<n_patterns; i++){
    printf(" %d ", predicted_class[i]);
    printf(" | %f %f %f | %f\n", predicted_class_prob[i*3],

```

```

        predicted_class_prob[i*3+1], predicted_class_prob[i*3+2],
        class_error[i]);
    if(i==49 || i==99){
        printf("%s\n",prtLabel);
        printf("%s\n",dashes);
    }
}
    imsls_f_mlff_network_write(network, filename, IMSLS_PRINT, 0);
}

```

Output

Note that the misclassification error rate is zero and Stage I training halts automatically at the 16th epoch because the cross-entropy error after the 16th epoch is below the default tolerance.

```

*****
* IRIS CLASSIFICATION EXAMPLE *
*****
Scale Parameters:
Var 1 Mean = 5.843334 S = 0.828065
Var 2 Mean = 3.057333 S = 0.435866
Var 3 Mean = 3.758000 S = 1.765298
Var 4 Mean = 1.199333 S = 0.762238

TRAINING PARAMETERS:
Stage II Opt. = 1
n_epochs = 20
epoch_size = 150
maxIterations = 1000
maxFunctionEval = 1000
maxStep = 1000.000000
functionTol = 2.42218e-005
gradientTol = 0.000345267
accuracy = 0.000345267
n_inputs = 4
n_continuous = 4
n_nominal = 0
n_classes = 3
n_outputs = 3
n_patterns = 150
n_layers = 2
n_perceptrons = 5
n_weights = 19

STAGE I TRAINING STARTING
Stage I: Epoch 1 - Cross-Entropy Error = 4.92197 (Iterations=77)
(CPU Min.=0.000260)
Stage I: Epoch 2 - Cross-Entropy Error = 5.95334 (Iterations=234)
(CPU Min.=0.001042)
Stage I: Epoch 3 - Cross-Entropy Error = 5.95312 (Iterations=237)
(CPU Min.=0.000781)
Stage I: Epoch 4 - Cross-Entropy Error = 74.9249 (Iterations=30)
(CPU Min.=0.000260)
Stage I: Epoch 5 - Cross-Entropy Error = 4.92196 (Iterations=130)
(CPU Min.=0.000260)
Stage I: Epoch 6 - Cross-Entropy Error = 5.9565 (Iterations=208)
(CPU Min.=0.000781)

```

```
Stage I: Epoch 7 - Cross-Entropy Error = 4.92199 (Iterations=99)
(CPU Min.=0.000521)
Stage I: Epoch 8 - Cross-Entropy Error = 4.92197 (Iterations=117)
(CPU Min.=0.000260)
Stage I: Epoch 9 - Cross-Entropy Error = 5.06757 (Iterations=500)
(CPU Min.=0.001302)
Stage I: Epoch 10 - Cross-Entropy Error = 5.94276 (Iterations=136)
(CPU Min.=0.000260)
Stage I: Epoch 11 - Cross-Entropy Error = 4.92198 (Iterations=80)
(CPU Min.=0.000260)
Stage I: Epoch 12 - Cross-Entropy Error = 4.92199 (Iterations=100)
(CPU Min.=0.000260)
Stage I: Epoch 13 - Cross-Entropy Error = 4.92199 (Iterations=87)
(CPU Min.=0.000260)
Stage I: Epoch 14 - Cross-Entropy Error = 5.95085 (Iterations=245)
(CPU Min.=0.000781)
Stage I: Epoch 15 - Cross-Entropy Error = 5.95099 (Iterations=165)
(CPU Min.=0.001042)
Stage I: Epoch 16 - Cross-Entropy Error = 2.5034e-005 (Iterations=134)
(CPU Min.=0.000521)
Stage I Training Converged at Epoch = 16
```

```
STAGE I FINAL CROSS-ENTROPY ERROR = 0.000025 (CPU Min.=0.008854)
```

```
OPTIMUM WEIGHTS AFTER STAGE I TRAINING:
```

```
weight[0] = 0.3079      weight[1] = -0.12877
weight[2] = 4.51303    weight[3] = -1.90144
weight[4] = -14.3699   weight[5] = -519.855
weight[6] = 1317.12    weight[7] = 2756.14
weight[8] = -3454.11   weight[9] = -193.738
weight[10] = 116.785   weight[11] = -1310.03
weight[12] = 3339.35   weight[13] = 1505.56
weight[14] = 1.01363   weight[15] = -3010.6
weight[16] = 1785.05   weight[17] = 682.292
weight[18] = -2467.38
```

```
STAGE I TRAINING CONVERGED
```

```
STAGE I CROSS-ENTROPY ERROR = 0.000025
```

```
0 PATTERNS OUT OF 150 INCORRECTLY CLASSIFIED
```

```
GRADIENT AT THE OPTIMUM WEIGHTS
```

```
-->g[0] = 0.001389      weight[0] = 0.307900
-->g[1] = 0.001205      weight[1] = -0.128770
-->g[2] = 0.000713      weight[2] = 4.513030
-->g[3] = 0.000269      weight[3] = -1.901438
-->g[4] = 0.000000      weight[4] = -14.369907
-->g[5] = 0.000000      weight[5] = -519.854919
-->g[6] = 0.000000      weight[6] = 1317.116821
-->g[7] = 0.000000      weight[7] = 2756.140625
-->g[8] = 0.000000      weight[8] = -3454.105713
-->g[9] = 0.000000      weight[9] = -193.738205
-->g[10] = -0.000002     weight[10] = 116.785263
-->g[11] = 0.000000      weight[11] = -1310.028076
-->g[12] = 0.000002     weight[12] = 3339.346680
-->g[13] = 0.000000      weight[13] = 1505.561646
-->g[14] = -0.000807     weight[14] = 1.013626
-->g[15] = 0.000000      weight[15] = -3010.596680
-->g[16] = 0.000002     weight[16] = 1785.052979
```

mlff_pattern_classification

Calculates classifications for trained multilayered feedforward neural networks.

Synopsis

```
#include <imsls.h>

float *imsls_f_mlff_pattern_classification (Imsls_f_NN_Network *network,
    int n_patterns, int n_nominal, int n_continuous, int nominal[ ], float continuous[ ], ...,
    0)
```

The type *double* function is `imsls_d_mlff_pattern_classification`.

Required Arguments

Imsls_f_NN_Network *network (Input)
Pointer to a structure of type *Imsls_f_NN_Network* containing the trained feedforward network. See [imsls_f_mlff_network](#).

int n_patterns (Input)
Number of patterns to classify.

int n_nominal (Input)
Number of nominal input attributes.

int n_continuous (Input)
Number of continuous attributes, including ordinal attributes encoded using cumulative percentage.

int nominal[] (Input)
Array of size n_patterns by n_nominal containing the nominal input variables.

float continuous[] (Input)
Array of size n_patterns by n_continuous containing the continuous and scaled ordinal input variables.

Return Value

Pointer to an array of size `n_patterns` by `n_classes` containing the predicted class probabilities associated with each input pattern, where `n_classes` is the number of possible target classifications.

`n_classes = network->n_outputs` for non-binary classification categories. For binary classification, `n_classes = 2`. This space can be released by using the `imsls_free` function.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_mlff_pattern_classification (Imsls_f_NN_Network *network,
    int n_patterns, int n_nominal, int n_continuous, int nominal[ ], float continuous[ ],
    IMSLS_LOGISTIC_TABLE,
    IMSLS_PREDICTED_CLASS, int **pred_class,
    IMSLS_PREDICTED_CLASS_USER, int pred_class[ ],
    IMSLS_RETURN_USER, float class_prob[ ],
    0)
```

Optional Arguments

`IMSLS_LOGISTIC_TABLE`, (Input)

This option specifies that all logistic activation functions are calculated using the table lookup approximation. This is only needed when a network is trained with this option and Stage II training is bypassed. If Stage II training was not bypassed during network training, weights were based upon the optimum network from Stage II which never uses a table lookup approximation to calculate logistic activations.

`IMSLS_PREDICTED_CLASS, int **pred_class` (Output)

The address of a pointer to an array of size `n_patterns` containing the predicted classification for each pattern.

`IMSLS_PREDICTED_CLASS_USER, int pred_class[]` (Output)

Storage for array `pred_class` provided by user. See `IMSLS_PREDICTED_CLASS`.

`IMSLS_RETURN_USER, float class_prob[]` (Output)

If specified, the classification probabilities for the input patterns are stored in the two-dimensional matrix `class_prob` of size `n_patterns` by `n_classes`, where `n_classes` is the number of target classes used to train the network. For binary classification problems, `n_classes = 2`. For all others, `n_classes = n_outputs = network->n_outputs`.

Description

Function `imsls_f_mlff_pattern_classification` calculates classification probabilities from 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 function. The weights are stored in the `Imsls_f_NN_Network` structure.

In addition, two two-dimensional arrays are used to describe the values of the nominal and continuous attributes that are to be used as network inputs for calculating classification probabilities. Optionally, it can also return the predicted classifications in `pred_class`. The predicted classification is the target class with the highest probability, `class_prob`.

Function `imsls_f_mlff_pattern_classification` returns classification probabilities for the network input patterns.

Pattern Classification Attributes

Neural network classification inputs consist of the following types of attributes:

1. nominal input attributes, and
2. continuous attributes, including ordinal attributes encoded to cumulative percentages.

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. The function `imsls_f_unsupervised_nominal_filter` can be used for this encoding.

When only one nominal attribute is used for input, then the number of binary encoded columns is equal to the number of classes for that attribute. If more nominal attributes appear in the data, then each nominal attribute is associated with several columns, one for each of its classes. Each column consists of zeros and ones. The column value is zero if that classification is not associated with this pattern; otherwise, it is equal to one if it is assigned to this pattern.

Consider an example with one nominal variable and two classes: *male* and *female* and the following five patterns: male, male, female, male, female. With binary encoding, the following 5 by 2 matrix is sent to the pattern classification to request classification probabilities for these patterns:

$$\text{nominal} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The second category of input attributes corresponds to continuous attributes. They are passed to this classification function via the floating point array `continuous`. The number of rows in this matrix is `n_patterns`, and the number of columns is `n_continuous`, corresponding to the number of continuous input attributes.

Ordinal input attributes, if used, are typically encoded to cumulative percentages. Since these are floating point values, they are placed into a column of the `continuous` array and `n_continuous` is set equal to the number of columns in this array.

In some cases, one of these types of input attributes may not exist. In that case, either `n_nominal` = 0 or `n_continuous` = 0 and their corresponding input matrix is ignored.

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 `lmsls_f_NN_Network`. See [lmsls_f_mlff_network](#). For binary problems there is only a single output since the probability $P(\text{class} = 0)$ is equal to $1 - P(\text{class} = 1)$. For other classification problems, however, `n_outputs` = `n_classes` and $P(\text{class} = j)$ is equal to the classification probabilities in the `j + 1` column of `class_prob[]`.

Classification Probabilities

Classification probabilities are calculated from the input attributes, network structure and weights provided in `network`.

Classification probabilities are returned in a two-dimensional array, `class_prob`, with `n_patterns` rows and `n_classes` columns. The values in the *i*-th column are estimated probabilities for the `class = (i-1)`.

Examples

Example 1

Fisher’s (1936) Iris data is often used for benchmarking discriminant analysis and classification solutions. It is part of the IMSL data sets and consists of the following continuous input attributes and classification target:

Continuous Attributes – X_1 (sepal length), X_2 (sepal width), X_3 (petal length), and X_4 (petal width)

Classification Target (Iris Type) – Setosa, Versicolour or Virginica.

The input attributes were scaled to z-scores using `imsls_f_scale_filter`. The hidden layer contained only 2 perceptrons and the output layer consisted of three perceptrons, one for each classification target.

[Example 2](#) for `imsls_f_mlff_classification_trainer` used the following network structure for the 150 patterns in these data:

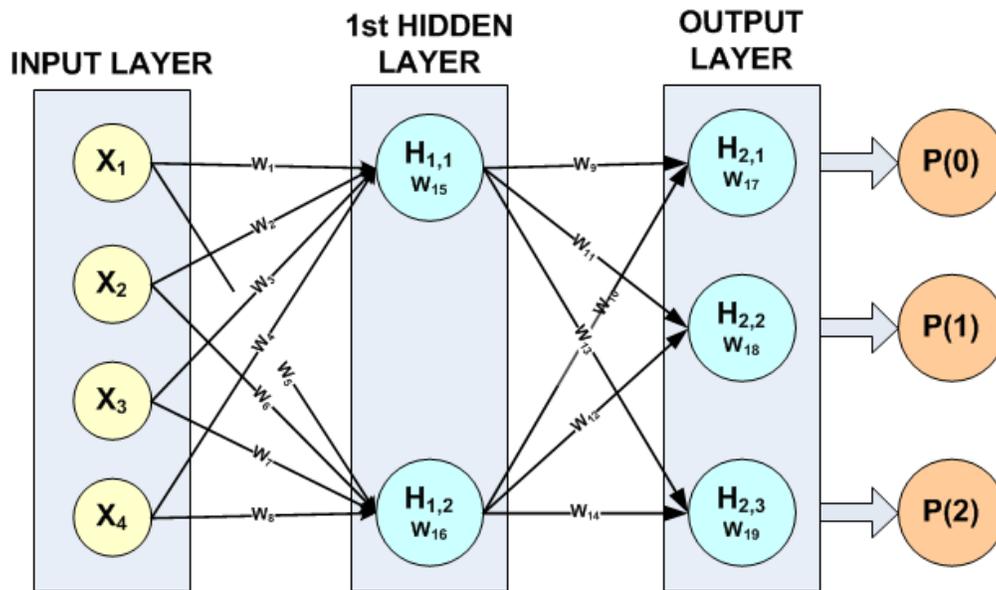


Figure 38, A 2-layer, Classification Network with 4 Inputs 5 Perceptrons and a Target Classification with 3 Classes

`imsls_f_mlff_classification_trainer` found the following 19 weights for this network:

$W_1 = -0.109866$	$W_2 = -0.0534655$	$W_3 = 4.92944$	$W_4 = -2.04734$
$W_5 = 10.2339$	$W_6 = -1495.09$	$W_7 = 3336.49$	$W_8 = 7372.98$
$W_9 = -9143.53$	$W_{10} = 48.8937$	$W_{11} = 240.958$	$W_{12} = -3386.21$

$W_{13} = 8904.6$	$W_{14} = 3339.1$	$W_{15} = 0.874638$	$W_{16} = -7978.42$
$W_{17} = 4586.22$	$W_{18} = 1931.89$	$W_{19} = -6518.14$	

The association of these weights with the calculation of the potentials for each perceptron is described in the following table:

Table 52 – Association of Network Weights with Perceptron Calculations

PERCEPTRON	POTENTIAL	ACTIVATION
$H_{1,1}$	$W_{15} + X_1W_1 + X_2W_2 + X_3W_3 + X_4W_4$	LOGISTIC
$H_{1,2}$	$W_{16} + X_1W_5 + X_2W_6 + X_3W_7 + X_4W_8$	LOGISTIC
$H_{2,1}$	$W_{17} + H_{1,1}W_9 + H_{1,2}W_{10}$	SOFTMAX
$H_{2,2}$	$W_{18} + H_{1,1}W_{11} + H_{1,2}W_{12}$	SOFTMAX
$H_{2,3}$	$W_{19} + H_{1,1}W_{13} + H_{1,2}W_{14}$	SOFTMAX

The potential calculations for each perceptron are activated using the assigned activation function. In this example, default activations were used, e.g. logistic for $H_{1,1}$ and $H_{1,2}$ and softmax for the output perceptrons $H_{2,1}$, $H_{2,2}$ and $H_{2,3}$.

Note that in this case the network weights were retrieved from a file named `iris_classification.txt` using `imsls_f_mlff_network_read`. This retrieves the trained network from `mlff_classification_trainer` described in [Example 2](#). These were passed directly to `imsls_f_mlff_pattern_classification` in the `Imsls_f_NN_Network` structure.

```
#include <stdio.h>
#include <imsls.h>
/* *****
* Three Layer Feed-Forward Network with 4 inputs, all
* continuous, and 3 classification categories.
*
* This is perhaps the best known database to be found in the pattern
* recognition literature. Fisher's paper is a classic in the
* field. The data set contains 3 classes of 50 instances each,
* where each class refers to a type of iris plant. One class is
* linearly separable from the other 2; the latter are NOT linearly
* separable from each other.
*
* Predicted attribute: class of iris plant.
* 1=Iris Setosa, 2=Iris Versicolour, and 3=Iris Virginica
*
* Input Attributes (4 Continuous Attributes)
* X1: Sepal length,
* X2: Sepal width,
* X3: Petal length,
* and X4: Petal width
*****/
int main(){
    float *classProb;
    int i, j;
```

```

int n_patterns      =150;
int n_inputs       =4; /* four inputs, all continuous */
int n_nominal      =0; /* no nominal input attributes */
int n_continuous   =4; /* one continuous input attribute */
int n_outputs      =3; /* total number of output perceptrons */
int *predicted_class;
int act_fcn[3] = {1, 1, 1};
int classification[150];
float unscaledX[150], scaledX[150];
float contAtt[4*150];
float *irisData;
float mean[4], s[4], center, spread;
char *colLabels[] = {"Pattern", "Class=0", "Class=1", "Class=2"};
char filename[] = "iris_classification.txt";
char prtLabel[] = "\nPredicted_Class | P(0) P(1) P(2)";
char dashes[] = "-----";
/* Declare mlff network structure */
Imsls_f_NN_Network *network;

printf("*****\n");
printf(" IRIS CLASSIFICATION EXAMPLE - PATTERN CLASSIFICATION \n");
printf("*****\n");
irisData = imsls_f_data_sets(3,0);

/*
 * Setup the continuous attribute input array, contAtt[], and the
 * network target classification array, classification[], using
 * the above raw data matrix.
 */
for(i=0; i < n_patterns; i++){
    classification[i] = (int)irisData[i*5]-1;
    for(j=1; j<5; j++){
        contAtt[i*4+j-1] = irisData[i*5+j];
    }
}
/* Scale continuous input attributes using z-score method */
for(j=0; j<n_continuous; j++){
    for(i=0; i<n_patterns; i++)
        unscaledX[i] = contAtt[i*n_continuous+j];
    imsls_f_scale_filter(n_patterns, unscaledX, 2,
        IMSLS_RETURN_USER, scaledX,
        IMSLS_RETURN_CENTER_SPREAD, &center, &spread, 0);
    for(i=0; i<n_patterns; i++)
        contAtt[i*n_continuous+j] = scaledX[i];
    mean[j] = center;
    s[j] = spread;
}
printf("Scale Parameters: \n");
for(j=0; j<n_continuous; j++){
    printf("Var %d Mean = %f S = %f \n",j+1, mean[j], s[j]);
}

network = imsls_f_mlff_network_read(filename, IMSLS_PRINT,0);

/* Use pattern classification routine to classify training
 * patterns using trained network.
 */
classProb = imsls_f_mlff_pattern_classification (network,
    n_patterns, n_nominal, n_continuous, NULL, contAtt,
    IMSLS_PREDICTED_CLASS, &predicted_class, 0);

```

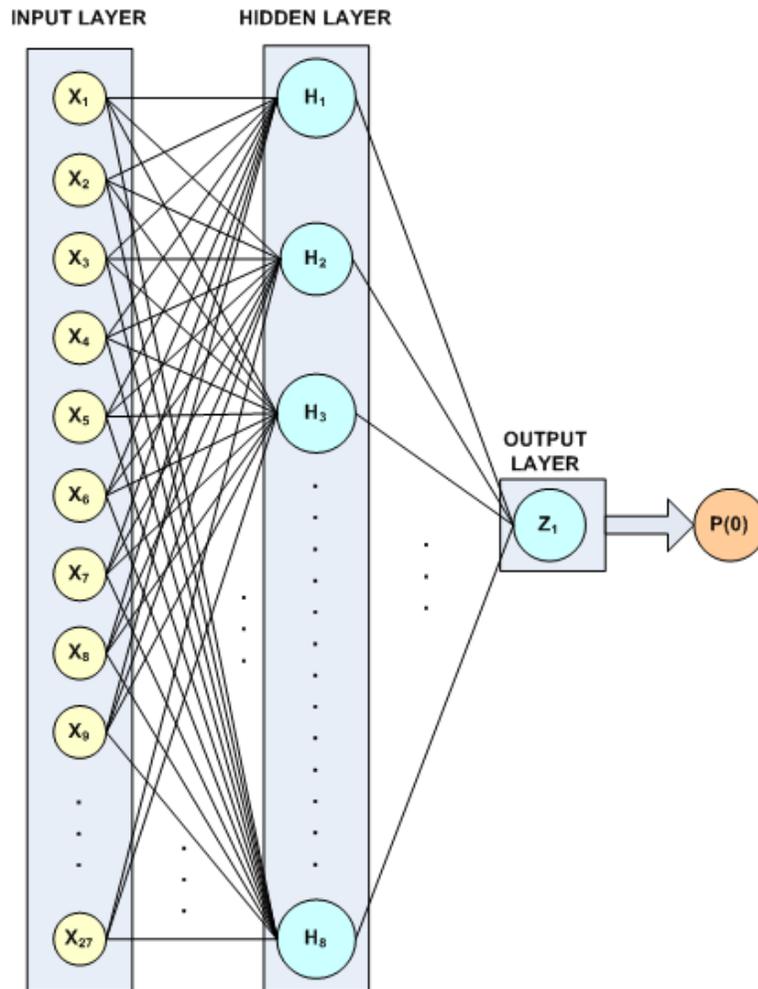



Figure 39, A 2-layer, Binary Classification Network for Playing Tic-Tac-Toe

All hidden layer perceptrons used the default activation, logistic, and since the classification target is binary only one perceptron with logistic activation is used to calculate the probability of a loss for X , i.e. $P(\text{class} = 0)$. All logistic activations are calculated using the `IMSLS_LOGISTIC_TABLE` option, which can reduce Stage I training time. Since Stage II training is bypassed, this option must also be used with the `imsls_f_mlff_pattern_classification` routine. This is the only time this option is used. If Stage II training was part of the network training, the final network weights would have been calculated without using the logistic table to approximate the calculations.

This structure results in a network with $27 \times 8 + 8 + 9 = 233$ weights. It is surprising that with this small a number of weights relative to the number of training patterns, the trained network achieves 100% classification accuracy.

Unlike Example 1 in which the network was trained previously and retrieved using [imsls_f_mlff_network_read](#), this example first trains the network and then passes the network structure into `imsls_f_mlff_pattern_classification`.

```

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

int main (){
    int i, j, k, m, n_patterns, n_var;
    int n_cat =9; /* 9 nominal input attributes */
    int n_categorical =27; /* 9 Encoded = 27 categorical inputs */
    int n_classes =2; /* positive or negative */
    float *classProb;
    float *trainStats;
    int *predictedClass;
    int classification[958];
    float *inputData;
    int *categoricalAtt, *nomTempIn, *nomTempOut, nClass;
    Imsls_f_NN_Network *network;

    /* get tic tac toe data */
    inputData = imsls_f_data_sets (10,
        IMSLS_N_OBSERVATIONS, &n_patterns,
        IMSLS_N_VARIABLES, &n_var,
        0);

    printf("\n\n");
    printf("*****\n");
    printf("* TIC-TAC-TOE BINARY CLASSIFICATION EXAMPLE *\n");
    printf("*****\n");

    /* allocate memory for categoricalATT array */
    categoricalAtt = (int*) malloc(958 * n_categorical * sizeof(int));

    /* populate categorical Att from catAtt using binary encoding */
    nomTempIn = (int*) malloc(n_patterns * sizeof(int));
    m=0;

    for(i=0; i<n_cat; i++)
    {
        for(j = 0; j < n_patterns; j++)
            nomTempIn[j] = (int) inputData[j * n_var + i] + 1;

        nomTempOut = imsls_unsupervised_nominal_filter(n_patterns,
            &nClass, nomTempIn,
            0);

        for(k = 0; k < nClass; k++)
        {
            for(j = 0; j < n_patterns; j++)
                categoricalAtt[j * n_categorical + m] =
                    nomTempOut[j * nClass + k];
            m++;
        }
        imsls_free(nomTempOut);
    }
    free(nomTempIn);
}

```

```

/* Setup the classification array, classification[] */
for(i = 0; i < n_patterns; i++)
    classification[i] = (int)inputData[(i * n_var) + n_var - 1];

network = imsls_f_mlff_network_init(27, 1);

imsls_f_mlff_network(network,
    IMSLS_CREATE_HIDDEN_LAYER, 8,
    IMSLS_LINK_ALL,
    0);

imsls_random_seed_set(5555);

/* Train Classification Network */
trainStats = imsls_f_mlff_classification_trainer (network,
    n_patterns, n_categorical, 0, classification, categoricalAtt,
    NULL,
    IMSLS_STAGE_I, 30, n_patterns,
    IMSLS_NO_STAGE_II,
    IMSLS_LOGISTIC_TABLE,
    IMSLS_WEIGHT_INITIALIZATION_METHOD,
    IMSLS_EQUAL,
    0);

/* Use pattern classification routine to classify training patterns
 * using trained network. This will reproduce the results returned
 * in predicted_class[] */
classProb = imsls_f_mlff_pattern_classification (network,
    n_patterns, n_categorical, 0, categoricalAtt, NULL,
    IMSLS_LOGISTIC_TABLE,
    IMSLS_PREDICTED_CLASS, &predictedClass,
    0);

/* Printing Classification Predictions */
printf("*****\n");
printf("Classification Minimum Cross-Entropy Error: %f\n",
    trainStats[0]);
printf("Classification Error Rate: %f \n", trainStats[5]);
printf("*****\n");
printf("\nPRINTING FIRST TEN PREDICTIONS FOR EACH TARGET CLASS\n");
printf("*****\n");
printf("      |TARGET|PREDICTED|      |      *\n");
printf("PATTERN |CLASS | CLASS | P(class=0) | P(class=1) *\n");
printf("*****\n");

for(k = 0; k < 2; k++){
    for(i = k * 627; i < k * 627 + 10; i++){
        printf(" %d\t| %d | %d | ", i+1,
            classification[i], predictedClass[i]);
        printf("%f | %f \n", classProb[i * n_classes],
            classProb[i * n_classes + 1]);
    }
    printf("\n");
}

k=0;

for(i = 0; i < n_patterns; i++)
    if(classification[i] != predictedClass[i])

```

```

        k++;

    if(k==0) {
        printf("All %d predicted classifications agree ", n_patterns);
        printf("with target classifications\n");
    }
}

```

Output

The output for this example demonstrates how `imsls_f_mlff_pattern_classification` reproduces the 100% classification accuracy found during network training.

```

*****
* TIC-TAC-TOE BINARY CLASSIFICATION EXAMPLE *
*****
Classification Minimum Cross-Entropy Error: 0.000022
Classification Error Rate: 0.000000
*****

```

PRINTING FIRST TEN PREDICTIONS FOR EACH TARGET CLASS

```

*****
PATTERN | TARGET | PREDICTED | P(class=0) | P(class=1) |
*****

```

PATTERN	TARGET CLASS	PREDICTED CLASS	P(class=0)	P(class=1)
1	1	1	0.000000	1.000000
2	1	1	0.000000	1.000000
3	1	1	0.000000	1.000000
4	1	1	0.000000	1.000000
5	1	1	0.000000	1.000000
6	1	1	0.000000	1.000000
7	1	1	0.000000	1.000000
8	1	1	0.000000	1.000000
9	1	1	0.000000	1.000000
10	1	1	0.000000	1.000000

628	0	0	1.000000	0.000000
629	0	0	1.000000	0.000000
630	0	0	1.000000	0.000000
631	0	0	1.000000	0.000000
632	0	0	1.000000	0.000000
633	0	0	1.000000	0.000000
634	0	0	1.000000	0.000000
635	0	0	1.000000	0.000000
636	0	0	1.000000	0.000000
637	0	0	1.000000	0.000000

All 958 predicted classifications agree with target classifications.

scale_filter

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.

Synopsis

```
#include <imspls.h>
```

```
float *imspls_f_scale_filter (int n_patterns, float x[], int method, ..., 0)
```

The type *double* function is `imspls_d_scale_filter`.

Required Arguments

int `n_patterns` (Input)

Number of observations.

float `x[]` (Input)

An array of length `n_patterns`. 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 [imspls_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.

<code>method</code>	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_patterns` 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 <imspls.h>

float *imspls_f_scale_filter (int n_patterns, 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_patterns` 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:

$$\tilde{s} = \frac{MAD}{0.6745}, \text{ where MAD is the Mean Absolute Deviation}$$

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

Again, the values of a and b can be overridden using the optional argument `IMSLI_S_SUPPLY_CENTER_SPREAD`.

Method ±1: Bounded Scaling and Unscaling

If `method=1`, then the optional argument `IMSLI_SCALE_LIMITS` is required and a scaling operation is conducted using the scale limits for \mathbf{x} using the following calculation:

$$z[i] = r(x[i] - \text{real_min}) + \text{target_min}$$

where

$$r = \frac{\text{target_max} - \text{target_min}}{\text{real_max} - \text{real_min}}$$

If `method=-1`, then optional argument `IMSLI_SCALE_LIMITS` is required and an unscaling operation is conducted by inverting the following calculation:

$$x[i] = \frac{(z[i] - \text{target_min})}{r} + \text{real_min}$$

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 \mathbf{x} using a z-score calculation:

$$z[i] = \frac{(x[i] - \text{center})}{\text{spread}}$$

If either `center` or `spread` are missing, (a NaN), then appropriate values are calculated from the non-missing values of \mathbf{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 `spread` 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.*"

$$x[i] = \text{spread} \cdot z[i] + \text{center}$$

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:

$$z[i] = \frac{r \cdot (x[i] - \text{center})}{\text{spread}} - r \cdot \text{real_min} + \text{target_min}$$

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 `center` 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.

$$x[i] = \frac{\text{spread} \cdot (z[i] - \text{target_min})}{r} + \text{spread} \cdot \text{real_min} + \text{center}$$

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 arithmetic 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`.

Example

In this example two data sets are filtered using bounded z-score scaling.

```
#include <imsls.h>
#include <stdio.h>
int main()
{
    int n_patterns=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_patterns, 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_patterns, 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_patterns, 1, z1, 0);
    printf("Center = %f\nSpread = %f\n", center1, spread1);
    imsls_f_write_matrix("z2", n_patterns, 1, z2, 0);
    printf("Center = %f\nSpread = %f\n", center2, spread2);

    /* Un-scale z1 and z2. */
    y1 = imsls_f_scale_filter(n_patterns, 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_patterns, 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_patterns, 1, y1, 0);
    imsls_f_write_matrix("y2", n_patterns, 1, y2, 0);
}
```

Output

```
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_patterns, int n_var, int max_lag, float x[ ], ..., 0)
```

The type *double* function is `imsls_d_time_series_filter`.

Required Arguments

int `n_patterns` (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` \geq 1.

int `max_lag` (Input)

The number of lags. The number of lags must be one or greater, `max_lag` \geq 1 and less than or equal to `n_patterns`.

float `x[]` (Input)

An array of size `n_patterns` by `n_var`. All data must be sorted in chronological order from most recent to oldest observation.

Return Value

A pointer to an internally allocated array of size $(n_patterns - max_lag)$ by $n_var \times (max_lag + 1)$. If errors are encountered, `NULL` is returned.

Synopsis with Optional Arguments

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

```
float *imsls_f_time_series_filter (int n_patterns, int n_var, int max_lag, float x[ ],
```

```
IMSL_S_RETURN_USER, float z[ ],
0)
```

Optional Arguments

IMSL_S_RETURN_USER, float z[] (Output)

User supplied array of size $(n_patterns - \max_lag)$ by $n_var \times (\max_lag + 1)$ 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, \mathbf{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 \mathbf{z} , the total number of lags is $n_lags = \max_lag + 1$.

The output data array, \mathbf{z} , can be represented symbolically as:

$$\mathbf{z} = |x(0) : x(1) : x(2) : \dots : x(\max_lag)|,$$

where $x(i)$ is the i -th lag of the incoming data matrix, \mathbf{x} . For example, if $\mathbf{x} = \{1, 2, 3, 4, 5\}$ and $n_var = 1$, then $n_patterns = 5$, and $x(0) = x$, $x(1) = \{2, 3, 4, 5\}$, $x(2) = \{3, 4, 5\}$, etc.

Consider, an example in which $n_patterns = 2$ and $n_var = 2$ with all variables having 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 $n_patterns - \max_lag = 5 - 1 = 4$:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 \\ 2 & 7 & 3 & 8 \\ 3 & 8 & 4 & 9 \\ 4 & 9 & 5 & 10 \end{bmatrix}$$

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 $n_patterns - \max_lag = 5 - 2 = 3$:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \\ 2 & 7 & 3 & 8 & 4 & 9 \\ 3 & 8 & 4 & 9 & 5 & 10 \end{bmatrix}$$

Example

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 `(n_patterns-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>
int main()
{
#define N_PATTERNS 5
#define N_VAR 2
#define MAX_LAG 2
float x[N_PATTERNS*N_VAR] = {1, 6,
                             2, 7,
                             3, 8,
                             4, 9,
                             5, 10};

float *z;

z = imsls_f_time_series_filter(N_PATTERNS, N_VAR, MAX_LAG, (float*)x, 0);
imsls_f_write_matrix("X", N_PATTERNS, N_VAR, (float*)x, 0);
imsls_f_write_matrix("Z", N_PATTERNS-MAX_LAG, N_VAR*(MAX_LAG+1), z, 0);
}
```

Output

	X			Z			
	1	2					
1	1	6		3	4	5	6
2	2	7		2	7	3	8
3	3	8		3	8	4	9
4	4	9		4	9	5	10
5	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_patterns, 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_patterns** (Input)

Number of observations. The number of observations must be greater than `max_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_patterns**. 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_patterns**. 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_patterns** 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_patterns, int n_lags, int n_classes,
    int i_class[], float x[],
    IMSLS_LAGS, int lag[],
    IMSLS_RETURN_USER, float z[],
    0)
```

The type *double* function is `imsls_d_time_series_class_filter`.

Optional Arguments

`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`

`IMSLS_RETURN_USER, float z[]` (Output)

A user-supplied array of size `n_patterns` 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]`.

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:

$$z = |x(0) : x(1) : x(2) : \dots : x(n_lags-1)|,$$

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 `lag[]` is given, the highest lag is `max_lags`. If `n_lags =2` and the optional input `lag[]` is not given, then the output array contains the lags 0, 1.

Consider, an example in which `n_patterns=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:

$$z = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & 6 \\ 5 & 7 \\ 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & \text{NaN} \\ 10 & \text{NaN} \end{bmatrix}$$

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:

$$z = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & \text{NaN} \\ 5 & \text{NaN} \\ 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & \text{NaN} \\ 10 & \text{NaN} \end{bmatrix}$$

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

Suppose that the training data to the neural network is represented by 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, $n_patterns=8$ and $n_classes=2$. If we wanted to lag the 3rd column by 2 time lags, i.e., $n_lags=2$,

$$lag^T = \{0, 1\}$$

$$i_class^T = \{1, 1, 1, 1, 2, 2, 2, 2\}$$

$$x^T = \{2.1, 2.3, 2.4, 2.5, 1.1, 1.2, 1.3, 1.4\}$$

The resulting data matrix would have 8 rows and 2 columns:

$$z = \begin{bmatrix} x(0) & x(1) \end{bmatrix} = \begin{bmatrix} 2.1 & 2.3 \\ 2.3 & 2.4 \\ 2.4 & 2.5 \\ 2.5 & \text{NaN} \\ 1.1 & 1.2 \\ 1.2 & 1.3 \\ 1.3 & 1.4 \\ 1.4 & \text{NaN} \end{bmatrix}$$

```
#include <imsls.h>

#define N_PATTERNS 8
#define N_LAGS 2

int main()
{
    float x[N_PATTERNS] = {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_PATTERNS, N_LAGS, n_classes,  
    i_class, x,  
    0);  
imsls_f_write_matrix("z", N_PATTERNS, N_LAGS, (float*)z,  
    0);  
return 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_patterns, int n_classes, int x[ ], ..., 0)
```

Required Arguments

int n_patterns (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_patterns* 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_patterns* 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_patterns, int n_classes, int x[ ],  
                                         IMSLS_ENCODE, or  
                                         IMSLS_DECODE,  
                                         IMSLS_RETURN_USER, int z[ ],  
                                         0)
```

Optional Arguments

`IMSLS_ENCODE`, (Input)

Specifies binary encoding. Classes must be numbered sequentially from 1 to `n_classes`. Optional Arguments `IMSLS_ENCODE` and `IMSLS_DECODE` are mutually exclusive.

Default: `IMSLS_ENCODE`.

or

`IMSLS_DECODE`, (Input)

Specifies that `x` will 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. Optional Arguments `IMSLS_ENCODE` and `IMSLS_DECODE` are mutually exclusive.

Default: `IMSLS_ENCODE`.

`IMSLS_RETURN_USER, int z[]` (Output)

A user-supplied array of size `n_patterns` by `n_classes`. If `IMSLS_DECODE` is specified, then `z` should be length `n_patterns`. 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 \mathbf{z} is equal to the number of distinct classes in \mathbf{x} . The number of rows in \mathbf{z} is equal to the length of \mathbf{x} .

Binary Decoding: IMSLS_DECODE

Binary decoding takes each column in $\mathbf{x}[\]$, and returns the appropriate class in $\mathbf{z}[\]$.

For example, if $\mathbf{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 $\mathbf{z}[\]$ would be returned as $\mathbf{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 $\mathbf{x}[\]$ is associated with the i -th class in the output array.

Example

This example illustrates nominal binary encoding and decoding for $x = \{3, 3, 1, 2, 2, 1, 2\}$.

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

int main ()
{
#define N_PATTERNS 7

    int x[N_PATTERNS] = {3, 3, 1, 2, 2, 1, 2};
    int *x2;
    int *z, n_classes;

    /* Binary Filtering. */
    z = imsls_unsupervised_nominal_filter(N_PATTERNS, &n_classes, x,
        0);

    printf("n_classes = %d\n",n_classes);
```

```
    imsls_i_write_matrix("X", N_PATTERNS, 1, (int*)x,
        0);
    imsls_i_write_matrix("Z", N_PATTERNS, n_classes, z,
        0);

    /* Binary Unfiltering. */
    x2 = imsls_unsupervised_nominal_filter(N_PATTERNS, &n_classes, z,
        IMSLS_DECODE,
        0);

    imsls_i_write_matrix("Unfiltering result", N_PATTERNS, 1, x2,
        0);
}
```

Output

```
7   n_classes = 3
8
9   X
10  1   3
11  2   3
12  3   1
13  4   2
14  5   2
15  6   1
16  7   2
17
18   Z
19   1   2   3
20  1   0   0   1
21  2   0   0   1
22  3   1   0   0
23  4   0   1   0
24  5   0   1   0
25  6   1   0   0
26  7   0   1   0
27
28  Unfiltering result
29   1   3
30   2   3
31   3   1
32   4   2
33   5   2
34   6   1
35   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_patterns, int x[ ], float z[ ], ..., 0)
```

The type *double* function is `imsls_d_unsupervised_ordinal_filter`.

Required Arguments

int `n_patterns` (Input)

Number of observations.

int `x[]` (Input/Output)

An array of length `n_patterns` 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_patterns` 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_patterns, int x[ ], float z[ ],
```

```
    IMSLS_ENCODE or
```

```
    IMSLS_DECODE,
```

```
    IMSLS_NO_TRANSFORM, or
```

```
    IMSLS_SQUARE_ROOT, or
```

```
    IMSLS_ARC_SIN,
```

```
IMSL_S_N_CLASSES, int *n_classes,  
0)
```

The type *double* function is `imsls_d_unsupervised_ordinal_filter`.

Optional Arguments

`IMSL_S_ENCODE`, (Input)

Specifies **z** as an output array and **x** 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). Optional Arguments `IMSL_S_ENCODE` and `IMSL_S_DECODE` are mutually exclusive.

Default: `IMSL_S_ENCODE`.

or

`IMSL_S_DECODE`, (Input)

Specifies **x** as an output array and **z** 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. Optional Arguments `IMSL_S_ENCODE` and `IMSL_S_DECODE` are mutually exclusive.

Default: `IMSL_S_ENCODE`.

`IMSL_S_NO_TRANSFORM`, (Input)

Indicates that the cumulative proportions used to encode the ordinal variable are not transformed. Optional Arguments `IMSL_S_NO_TRANSFORM`, `IMSL_S_SQUARE_ROOT`, and `IMSL_S_ARC_SIN` are mutually exclusive.

Default: `IMSL_S_NO_TRANSFORM`.

or

`IMSL_S_SQUARE_ROOT`, (Input)

Indicates cumulative proportions are transformed using the square root transformation. Optional Arguments `IMSL_S_NO_TRANSFORM`, `IMSL_S_SQUARE_ROOT`, and `IMSL_S_ARC_SIN` are mutually exclusive.

Default: `IMSL_S_NO_TRANSFORM`.

or

`IMSL_S_ARC_SIN`, (Input)

Indicates cumulative proportions are transformed using the arcsin of the square root of the cumulative proportions. Optional Arguments `IMSL_S_NO_TRANSFORM`, `IMSL_S_SQUARE_ROOT`, and `IMSL_S_ARC_SIN` are mutually exclusive.

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_patterns=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

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>
#include <stdio.h>
```

```
int main () {
#define N_PATTERNS 5
  int x[N_PATTERNS] = {2,3,4,1,4};
  int x2[N_PATTERNS], n_classes;
  float z[N_PATTERNS];

  /* Filtering. */
  imsls_f_unsupervised_ordinal_filter(N_PATTERNS, x, z,
    IMSLS_N_CLASSES, &n_classes,
    0);
  printf("n_classes = %d\n", n_classes);

  imsls_i_write_matrix("x", N_PATTERNS, 1, x, 0);
  imsls_f_write_matrix("z", N_PATTERNS, 1, z, 0);

  /* Unfiltering. */
  imsls_f_unsupervised_ordinal_filter(N_PATTERNS, x2, z,
    IMSLS_DECODE,
    IMSLS_N_CLASSES, &n_classes,
    0);
  printf("\nn_classes = %d\n", n_classes);

  imsls_i_write_matrix("x-unfiltered", N_PATTERNS, 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
```

kohonenSOM_trainer



[more...](#)

Trains a Kohonen network.

Synopsis

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

```
imsls_f_kohonenSOM *imsls_f_kohonenSOM_trainer (float fcn( ), float lcn( ), int dim,
int nrow, int ncol, int nobs, float data[ ], ..., 0)
```

The type *double* function is `imsls_d_kohonenSOM_trainer`.

Required Arguments

float fcn (*int nrow*, *int ncol*, *int total_iter*, *int t*, *float d*) (Input/Output)

User-supplied neighborhood function. In the simplest form, the neighborhood function $h(d, t)$ is 1 for all nodes closest to the BMU and 0 for others, but a Gaussian function is also commonly used. For example:

$$h(d, t) = e^{-d^2/2r^2} \text{ where } r \text{ represents the neighborhood radius at index } t$$

Arguments

int nrow (Input)

The number of rows in the node grid.

int ncol (Input)

The number of columns in the node grid.

int total_iter (Input)

The number of iterations for training.

int t (Input)

The current iteration of the training.

float d (Input)

The lattice distance between the best matching node and the current node.

Return Value

The computed neighborhood function value.

float lcn (*int nrow*, *int ncol*, *int total_iter*, *int t*) (Input/Output)

User supplied learning coefficient function. The monotonically decreasing learning coefficient function $\alpha(t)$ is a scalar factor that defines the size of the update correction. The value of $\alpha(t)$ decreases with the step index t . Typical forms are linear, power, and inverse time/step. For example:

power:

$$\alpha(t) = \alpha_0 \left(\frac{\alpha_T}{\alpha_0} \right)^{t/T}$$

where $t=t$, $T=\text{total_iter}$, α_0 = initial learning coefficient, α_T = final learning coefficient

inverse time:

$$\alpha(t) = \frac{A}{t+B}$$

where A and B are user determined constants

Arguments

int nrow (Input)

The number of rows in the node grid.

int ncol (Input)

The number of columns in the node grid.

int total_iter (Input)

The number of iterations for training.

int t (Input)

The current iteration of the training.

Return Value

The computed learning coefficient.

int dim (Input)

The number of weights for each node in the node grid. **dim** must be greater than zero.

int nrow (Input)

The number of rows in the node grid. **nrow** must be greater than zero.

int ncol (Input)

The number of columns in the node grid. **ncol** must be greater than zero.

int nobs (Input)

The number of observations in **data**. **nobs** must be greater than zero.

float data[] (Input)

An **nobs** × **dim** array containing the data to be used for training the Kohonen network.

Return Value

A pointer to a `Imsls_f_kohonenSOM` data structure containing the trained Kohonen network. This space can be released by using the `imsls_free` function. Please see [Data Structures](#) for a description of this data structure.

Synopsis with Optional Arguments

```
#include <imsls.h>

Imsls_f_kohonenSOM imsls_f_kohonenSOM_trainer (float fcn( ), float lcn( ), int dim, int nrow,
int ncol, int nobs, float data[ ],

    IMSLS_RECTANGULAR, or

    IMSLS_HEXAGONAL,

    IMSLS_VON_NEUMANN, or

    IMSLS_MOORE,

    IMSLS_WRAP_AROUND,

    IMSLS_RANDOM_SEED, int seed,

    IMSLS_ITERATIONS, int total_iter,

    IMSLS_INITIAL_WEIGHTS, float weights[ ],

    IMSL_FCN_W_DATA, float fcn( ), void *data,

    IMSL_LCN_W_DATA, float lcn( ), void *data,

    IMSL_RECONSTRUCTION_ERROR, float *error,

    0)
```

Optional Arguments

IMSLS_RECTANGULAR, (Input)

Specifies a rectangular grid should be used. Optional Arguments **IMSLS_RECTANGULAR** and **IMSLS_HEXAGONAL** are mutually exclusive.

Argument	Action
IMSLS_RECTANGULAR	Use a rectangular grid.
IMSLS_HEXAGONAL	Use a hexagonal grid.

Default: A rectangular grid is used.

or

IMSLS_HEXAGONAL

Specifies a hexagonal grid should be used. Optional Arguments **IMSLS_RECTANGULAR** and **IMSLS_HEXAGONAL** are mutually exclusive.

Argument	Action
IMSLS_RECTANGULAR	Use a rectangular grid.
IMSLS_HEXAGONAL	Use a hexagonal grid.

Default: A rectangular grid is used.

IMSLS_VON_NEUMANN, (Input)

Use the Von Neumann neighborhood type. Optional Arguments **IMSLS_VON_NEUMAN** and **IMSLS_MOORE** are mutually exclusive.

Argument	Action
IMSLS_VON_NEUMANN	Use the Von Neumann neighborhood type.
IMSLS_MOORE	Use the Moore neighborhood type.

Default: The Von Neumann neighborhood type is used.

or

IMSLS_MOORE

Use the Moore neighborhood type. Optional Arguments IMSLS_VON_NEUMAN and IMSLS_MOORE are mutually exclusive.

Argument	Action
IMSLS_VON_NEUMANN	Use the Von Neumann neighborhood type.
IMSLS_MOORE	Use the Moore neighborhood type.

Default: The Von Neumann neighborhood type is used.

IMSLS_WRAP_AROUND, (Input)

Wrap around the opposite edges. A hexagonal grid must have an even number of rows to wrap around.

Default: Do not wrap around the opposite edges.

IMSLS_RANDOM_SEED, *int seed* (Input)

The seed of the random number generator used in generating the initial weights. If *seed* is 0, a value is computed using the system clock; hence, the results may be different between different calls with the same input.

Default: *seed* = 0.

IMSLS_ITERATIONS, *int total_iter* (Input)

The number of iterations to be used for training.

Default: *total_iter* = 100.

IMSLS_INITIAL_WEIGHTS, *float weights[]* (Input)

The initial weights of the nodes.

Default: Initial weights are generated internally using random uniform number generator.

IMSL_FCN_W_DATA, *float fcn (int nrow, int ncol, int total_iter, int t, float d, void *data), void *data* (Input)

*float fcn (int nrow, int ncol, int total_iter, int t, float d, void *data)* (Input)

User supplied neighborhood 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.

Arguments

int nrow (Input)

The number of rows in the node grid.

int ncol (Input)

The number of columns in the node grid.

int total_iter (Input)

The number of iterations for training.

int t (Input)

The current iteration of the training.

float d (Input)

The lattice distance between the best matching node and the current node.

*void *data* (Input)

A pointer to the data to be passed to the user-supplied function.

Return Value

The computed neighborhood function value.

*void *data* (Input)

A pointer to the data to be passed to the user-supplied function.

IMSL_LCN_W_DATA, float lcn (*int nrow, int ncol, int total_iter, int t, void *data*), *void *data*
(Input)

float lcn (*int nrow, int ncol, int total_iter, int t, void *data*) (Input)

User supplied learning coefficient 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.

Arguments

int nrow (Input)

The number of rows in the node grid.

int ncol (Input)

The number of columns in the node grid.

int total_iter (Input)

The number of iterations for training.

int t (Input)

The current iteration of the training.

*void *data* (Input)

A pointer to the data to be passed to the user-supplied function.

Return Value

The computed learning coefficient.

*void *data* (Input)

A pointer to the data to be passed to the user-supplied function.

*IMSLS_RECONSTRUCTION_ERROR, float *error* (Output)

The sum of the Euclidean distance between the input, *data*, and the nodes in the trained Kohonen network.

Description

A self-organizing map (SOM), also known as a Kohonen map or Kohonen SOM, is a technique for gathering high-dimensional data into clusters that are constrained to lie in low dimensional space, usually two dimensions. A Kohonen map is a widely used technique for the purpose of feature extraction and visualization for very high dimensional data in situations where classifications are not known beforehand. The Kohonen SOM is equivalent to an artificial neural network having inputs linked to every node in the network. Self-organizing maps use a neighborhood function to preserve the topological properties of the input space.

In a Kohonen map, nodes are arranged in a rectangular or hexagonal grid or lattice. The input is connected to each node, and the output of the Kohonen map is the zero-based (i, j) index of the node that is closest to the input. A Kohonen map involves two steps: training and forecasting. Training builds the map using input examples (vectors), and forecasting classifies a new input.

During training, an input vector is fed to the network. The input's Euclidean distance from all the nodes is calculated. The node with the shortest distance is identified and is called the Best Matching Unit, or BMU. After identifying the BMU, the weights of the BMU and the nodes closest to it in the SOM lattice are updated towards the input vector. The magnitude of the update decreases with time and with distance (within the lattice) from the BMU. The weights of the nodes surrounding the BMU are updated according to:

$$W_{t+1} = W_t + \alpha(t) * h(d,t) * (D_t - W_t)$$

where W_t represents the node weights, $\alpha(t)$ is the monotonically decreasing learning coefficient function, $h(d,t)$ is the neighborhood function, d is the lattice distance between the node and the BMU, and D_t is the input vector.

The monotonically decreasing learning coefficient function $\alpha(t)$ is a scalar factor that defines the size of the update correction. The value of $\alpha(t)$ decreases with the step index t .

The neighborhood function $h(d,t)$ depends on the lattice distance d between the node and the BMU, and represents the strength of the coupling between the node and BMU. In the simplest form, the value of $h(d,t)$ is 1 for all nodes closest to the BMU and 0 for others, but a Gaussian function is also commonly used. Regardless of the functional form, the neighborhood function shrinks with time (Hollmén, 15.2.1996). Early on, when the neighborhood is broad, the self-organizing takes place on the global scale. When the neighborhood has shrunk to just a couple of nodes, the weights converge to local estimates.

Note that in a rectangular grid, the BMU has four closest nodes for the Von Neumann neighborhood type, or eight closest nodes for the Moore neighborhood type. In a hexagonal grid, the BMU has six closest nodes.

During training, this process is repeated for a number of iterations on all input vectors.

During forecasting, the node with the shortest Euclidean distance is the winning node, and its (i, j) index is the output.

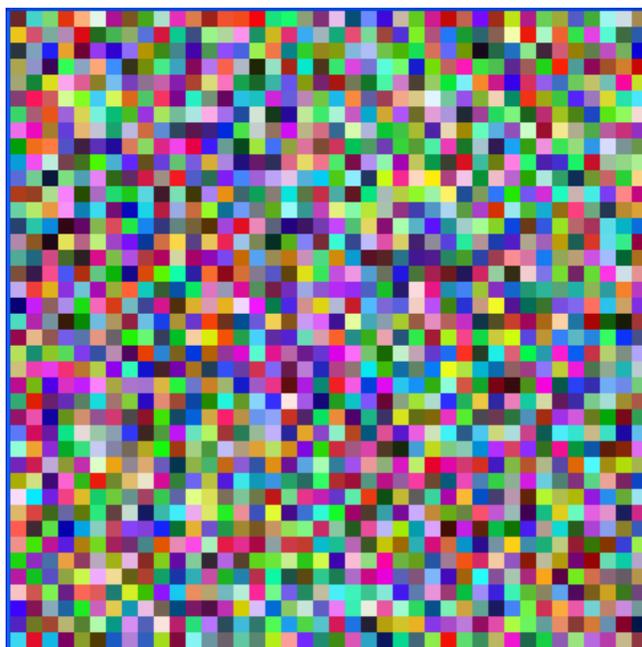
Data Structures

Table 53 – The data structure `ims1s_f_kohonenSOM`

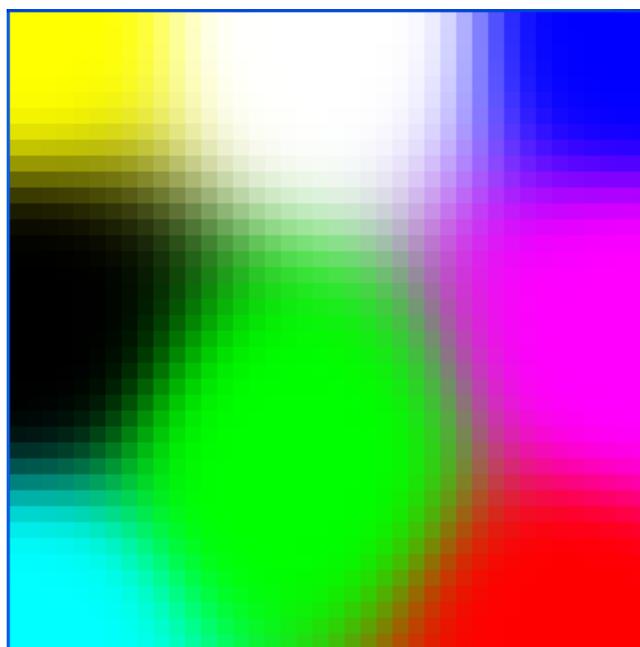
Field	Description
<code>int grid</code>	0 = rectangular grid. Otherwise, hexagonal grid.
<code>int type</code>	0 = Von Neumann neighborhood type. Otherwise, Moore neighborhood type.
<code>int wrap</code>	0 = do not wrap-around node edges. Otherwise, wrap-around node edges.
<code>int dim</code>	Number of weights in each node.
<code>int nrow</code>	Number of rows in the node grid.
<code>int ncol</code>	Number of columns in the node grid.
<code>float *weights</code>	Array of length <code>nrow x ncol x dim</code> containing the weights of the nodes.

Example

This example creates a Kohonen network with 40×40 nodes. Each node has three weights, representing the RGB values of a color. This network is trained with eight colors using 500 iterations. Then, the example prints out a forecast result. Initially, the image of the nodes is:



After the training, the image is:



```
#include <stdio.h>
#include <math.h>
#include <imsls.h>
float fcn(int nrow, int ncol, int total_iter, int t, float d);
float lcn(int nrow, int ncol, int total_iter, int t);

int main() {
    Imsls_f_kohonenSOM *kohonen=NULL;
    int dim=3, nrow=40, ncol=40, nobs=8;
    float data[8][3] = {
        {1.0, 0.0, 0.0},
        {0.0, 1.0, 0.0},
        {0.0, 0.0, 1.0},
        {1.0, 1.0, 0.0},
        {1.0, 0.0, 1.0},
        {0.0, 1.0, 1.0},
        {0.0, 0.0, 0.0},
        {1.0, 1.0, 1.0}
    };
    int *forecasts = NULL;
    float fdata[1][3] = {
        {0.25, 0.5, 0.75}
    };
    float error;

    kohonen = imsls_f_kohonenSOM_trainer(fcn, lcn, dim, nrow, ncol,
        nobs, &data[0][0],
        IMSLS_RANDOM_SEED, 123457,
        IMSLS_ITERATIONS, 500,
        IMSLS_RECONSTRUCTION_ERROR, &error,
        0);

    forecasts =
        imsls_f_kohonenSOM_forecast(kohonen, 1, &fdata[0][0], 0);

    printf("The output node is at (%d, %d).\n",
        forecasts[0], forecasts[1]);

    printf("Reconstruction error is %f.\n", error);

    /* Free up memory. */
    imsls_free(kohonen->weights);
    imsls_free(kohonen);
    imsls_free(forecasts);
}

float fcn(int nrow, int ncol, int total_iter, int t, float d) {
    float factor, c;
    int max;

    max = nrow > ncol ? nrow : ncol;

    /* A Gaussian function. */
    factor = max / 4.0;
    c = (float) (total_iter - t) / ((float) total_iter / factor);

    return exp(-(d * d) / (2.0 * c * c));
}
```

```
float lcn(int nrow, int ncol, int total_iter, int t) {  
    float initialLearning = 0.07;  
  
    return initialLearning * exp(-(float) t / (float) total_iter);  
}
```

Output

```
The output node is at (25, 11).  
Reconstruction error is 13589.462891.
```

kohonenSOM_forecast

Calculates forecasts using a trained Kohonen network.

Synopsis

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

```
int *imsls_f_kohonenSOM_forecast (Imsls_f_kohonenSOM *kohonen, int nobs, float data[ ],  
..., 0)
```

The type *double* function is `imsls_d_kohonenSOM_forecast`.

Required Arguments

Imsls_f_kohonenSOM *kohonen (Input)

Pointer to a structure of type *Imsls_f_kohonenSOM* containing the trained Kohonen network. See the Remarks section of `imsls_f_kohonenSOM_trainer` for a description of this structure.

int nobs (Input)

The number of observations in `data`. `nobs` must be greater than zero.

float data[] (Input)

An `nobs` × `dim` array containing the input data for forecasts, where `dim` is the number of weights for each node in the node grid specified during training.

Return Value

An `nobs` × 2 array containing the (i, j) index of the winning nodes for the input data. This space can be released by using the `imsls_free` function.

Synopsis with Optional Arguments

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

```
int *imsls_f_kohonenSOM_forecast (Imsls_f_kohonenSOM *kohonen, int nobs, float data[ ],  
IMSLS_RETURN_USER, int forecasts[ ],
```

0)

Optional Arguments

`IMSLS_RETURN_USER`, *int forecasts* [] (Output)

If specified, the forecasts are returned in the user-supplied array `forecasts`.

Description

Function `imsls_f_kohonenSOM_forecast` calculates forecasts for a previously trained Kohonen network from `imsls_f_kohonenSOM_trainer`. The structure `Imsls_f_kohonenSOM` 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 function. The weights are stored in the `Imsls_f_kohonenSOM` structure.

Example

See `imsls_f_kohonenSOM_trainer`.

Output

See `imsls_f_kohonenSOM_trainer`.

Support Vector Machines – An Overview

Support Vector Machines (SVM) are a class of learning algorithms for classification, regression, and distribution estimation motivated by results of statistical learning theory (Vapnik, 1995). Classification problems are characterized by separating data into training and testing sets. Each pattern, or instance in the training set, contains one “target classification value” (i.e. one of the class values) and several “attributes” (i.e. the features or observed variables). The goal of SVM is to produce a model based on the training data that predicts the target values of the test data.

If (\mathbf{x}_i, y_i) are the instance-label pairs for a given training set, and $i = 1, \dots, l$, where l is the number of training patterns, $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{1, -1\}$, the support vector machine (SVM) (Boser et al., 1992; Cortes and Vapnik, 1995) solves the following primal optimization problem:

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^l \xi_i \\ \text{subject to} \quad & y_i (\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \quad i = 1, \dots, l \end{aligned}$$

The quantities \mathbf{w} and b are the weight vector and bias. $C > 0$ is the penalty parameter of the error term ξ_i . The training vectors \mathbf{x}_i are mapped into a higher (maybe infinite) dimensional space by the function $\phi(\mathbf{x}_i)$, called the input features. SVM finds a linear separating hyperplane of maximal margin in this higher dimensional space. Rather than applying SVM using the original input attributes \mathbf{x}_i , the new features $\phi(\mathbf{x}_i)$ are passed to the learning algorithm. $K(x, y)$, an inner product defined as $K(\mathbf{x}_i, \mathbf{x}_j) \equiv \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$, is the kernel function. Often, even though $\phi(\mathbf{x})$ itself may be very expensive to calculate (perhaps because it is an extremely high-dimensional vector), $K(\mathbf{x}_i, \mathbf{x}_j)$ may be very inexpensive to calculate. Without ever having to explicitly find or represent vectors $\phi(\mathbf{x})$, using $K(\mathbf{x}_i, \mathbf{x}_j)$ is an efficient way for SVMs to learn in the high-dimensional feature space. Four popular kernels for classification and regression are:

- linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$
- polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i^T \mathbf{x}_j + r)^d, \quad \gamma > 0$
- radial basis function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2), \quad \gamma > 0$
- sigmoid: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i^T \mathbf{x}_j + r)$

where γ , r , and d are kernel parameters.

SVM classification algorithms determine an optimal large-margin linear decision boundary for the given training data. The SVM formulation for classification could be for either two-class or multi-class classifications. Multiple binary classifiers are combined for multi-class classification. If the class information is not provided for the training data then the distribution estimation algorithm one-class SVM is used to estimate the support vectors of a high-dimensional distribution. The support vector methodology can also be applied to the regression problem by seeking to optimize the generalization bounds for regression which rely on defining a loss function that ignores errors within a certain distance of the true value. The following classification algorithms are supported.

- SVC (Support Vector Classification): Two-class and multi-class. This is the standard SVM algorithm used to classify two-class or multi-class data.
- One-class SVM: This algorithm assumes that the data is available from only one class. For example, the data comes from some unknown underlying probability distribution, P .
- SVR (Support Vector Regression): This algorithm applies the features of the SVM algorithm to the regression problem.

A typical use involves these steps:

- Scale the data. Typically, the data is linearly scaled to the range $[-1, 1]$ or $[0, 1]$. The same scaling parameters must be used on both the training data and the test data. You may find [`imsls_f_scale_filter`](#) useful for this step.
- Apply the trainer to the scaled training data set using one of the available kernel types to obtain a model. The RBF kernel is a good kernel type to start with.
- Use cross-validation to find the best model parameters.
- Use the resulting model with the best model parameters to predict information about the scaled testing data set.

For SVC and SVR, the classifier can calculate probability estimates. Function [`imsls_f_support_vector_trainer`](#) is used to train a classifier from a set of training patterns with values of both the input attributes and target classes. This function stores the trained classifier model into an *imsls_f_svm_model* data structure.

Unknown classifications of new patterns can be predicted by passing the trained classifier model data structure, *imsls_f_svm_model*, to [`imsls_f_support_vector_classification`](#). If necessary, memory allocated to the trained classifier model can be released using [`imsls_f_svm_classifier_free`](#).

support_vector_trainer

Trains a Support Vector Machines (SVM) classifier.

Synopsis

```
#include <ims1s.h>
```

```
ims1s_f_svm_model *ims1s_f_support_vector_trainer(int n_patterns, int n_classes,  
int n_attributes, float classification[], float x[], ..., 0)
```

The type *double* function is `ims1s_d_support_vector_trainer`.

Required Arguments

int n_patterns (Input)
Number of training patterns.

int n_classes (Input)
Number of unique target classification values.

int n_attributes (Input)
Number of attributes.

float classification[] (Input)
Array of length `n_patterns` containing the target classification values for each of the training patterns.

float x[] (Input)
Array of length `n_patterns` by `n_attributes` containing the training data matrix.

Return Value

A pointer to a structure of type `ims1s_f_svm_model` containing the trained support vector classifier model. If training is unsuccessful, `NULL` is returned. To release this space, use `ims1s_f_svm_classifier_free`.

Synopsis with Optional Arguments

```
#include <ims1s.h>
```

```

lmsls_f_svm_model *lmsls_f_support_vector_trainer (int n_patterns, int n_classes,
int n_attributes, float classification[], float x[],
IMSLS_SVM_C_SVC_TYPE, float C, int nr_weight, float weight_class[],
float weight[], or
IMSLS_SVM_NU_SVC_TYPE, float nu, or
IMSLS_SVM_ONE_CLASS_TYPE, float nu, or
IMSLS_SVM_EPSILON_SVR_TYPE, float C, float p, or
IMSLS_SVM_NU_SVR_TYPE, float C, float nu,
IMSLS_SVM_WORK_ARRAY_SIZE, float work_size,
IMSLS_SVM_EPSILON, float epsilon,
IMSLS_SVM_NO_SHRINKING,
IMSLS_SVM_TRAIN_ESTIMATE_PROB,
IMSLS_SVM_KERNEL_LINEAR, or
IMSLS_SVM_KERNEL_POLYNOMIAL, int degree, float gamma, float coef0, or
IMSLS_SVM_KERNEL_RADIAL_BASIS, float gamma, or
IMSLS_SVM_KERNEL_SIGMOID, float gamma, float coef0, or
IMSLS_SVM_KERNEL_PRECOMPUTED, float kernel_values[],
IMSLS_SVM_CROSS_VALIDATION, int n_folds, float **target, float *result
IMSLS_SVM_CROSS_VALIDATION_USER, int n_folds, float target[], float *result,
0)

```

Optional Arguments

IMSLS_SVM_C_SVC_TYPE, float C, int nr_weight, float weight_class[], float weight[]

(Input)

Specifies that the C-support vector classification (C-SVC) algorithm is to be used to create the classification model. This is the default type of SVM used.

float C (Input)

The regularization parameter. *C* must be greater than 0. By default, the penalty parameters are set to the regularization parameter *C*. The penalty parameters can be changed by scaling *C* by the values specified in **weight** below.

int nr_weight (Input)

The number of elements in **weight** and **weight_class** used to change the penalty parameters.

float weight_class[] (Input)

An array of length **nr_weight** containing the target classification values that are to be weighted.

float weight[] (Input)

An array of length **nr_weight** containing the weights corresponding to the target classification values in **weight_class** to be used to change the penalty parameters.

Default: C-SVC is the default SVM type used with **C** = 5.0, **nr_weight** = 0, **weight_class** = NULL, and **weight** = NULL.

or

IMSLS_SVM_NU_SVC_TYPE, *float nu* (Input)

Specifies that the **v**-support vector classification (**v**-SVC) algorithm is to be used to create the classification model.

float nu (Input)

The parameter **nu** controls the number of support vectors and $nu \in (0,1]$.

or

IMSLS_SVM_ONE_CLASS_TYPE, *float nu* (Input)

Specifies that the distribution estimation (one-class SVM) algorithm is to be used to create the classification model.

float nu (Input)

The parameter **nu** controls the number of support vectors and $nu \in (0,1]$.

or

`IMSLS_SVM_EPSILON_SVR_TYPE`, *float* `C`, *float* `p` (Input)

Specifies that the ϵ -support vector regression (ϵ -SVR) algorithm is to be used to create the classification model.

float `C` (Input)

The regularization parameter. `C` must be greater than 0.

float `p` (Input)

The insensitivity band parameter `p` must be positive.

or

`IMSLS_SVM_NU_SVR_TYPE`, *float* `C`, *float* `nu` (Input)

Specifies that the ν -support vector regression (ν -SVR) algorithm is to be used to create the classification model.

float `C` (Input)

The regularization parameter. `C` must be greater than 0.

float `nu` (Input)

The parameter `nu` controls the number of support vectors and $\text{nu} \in (0,1]$.

`IMSLS_SVM_WORK_ARRAY_SIZE`, *float* `work_size` (Input)

This work array size argument sets the number of megabytes allocated for the work array used during the decomposition method. A larger work array size can reduce the computational time of the decomposition method.

Default: `work_size` = 1.0.

`IMSLS_SVM_EPSILON`, *float* `epsilon` (Input)

The absolute accuracy tolerance for termination criterion. The algorithm uses the SMO algorithm in solving the optimization problem. When the Lagrange multipliers used in the SMO algorithm satisfy the Karush-Kuhn-Tucker (KKT) conditions within `epsilon`, convergence is assumed.

Default: `epsilon` = 0.001.

`IMSLS_SVM_NO_SHRINKING`, (Input)

Use of this argument specifies that the shrinking technique is not to be used in the SMO algorithm. The shrinking technique tries to identify and remove some bounded elements during the application of the SMO algorithm, so a smaller optimization problem is solved.

Default: Shrinking is performed.

`IMSLS_SVM_TRAIN_ESTIMATE_PROB`, (Input)

Instructs the trainer to include information in the resultant classifier model to enable you to obtain probability estimates when invoking `imsls_f_support_vector_classification`.

Default: Information necessary to obtain probability estimates is **not** included in the model.

IMSL_SVM_KERNEL_LINEAR, (Input)

This argument specifies that the inner-product kernel type

$$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$$

is to be used. This kernel type is best used when the relation between the target classification values and attributes is linear or when the number of attributes is large (for example, 1000 attributes).

or

IMSL_SVM_KERNEL_POLYNOMIAL, *int degree*, *float gamma*, *float coef0* (Input)

This argument specifies that the polynomial kernel type

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma \mathbf{x}_i^T \mathbf{x}_j + r)^d$$

is to be used. Use this argument when the data are not linearly separable.

int degree (Input)

Parameter **degree** specifies the order of the polynomial kernel. **degree** = d in the equation above.

float gamma (Input)

Parameter **gamma** must be greater than 0. **gamma** = γ in the equation above.

float coef0 (Input)

Parameter **coef0** corresponds to r in the equation above.

or

IMSL_SVM_KERNEL_RADIAL_BASIS, *float gamma* (Input)

This argument specifies that the radial basis function kernel type

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

is to be used. Use this kernel type when the relation between the class labels and attributes is nonlinear, although it can also be used when the relation between the target classification values and attributes is linear. This kernel type exhibits fewer numerical difficulties. If no kernel type is specified, this is the kernel type used.

float gamma (Input)

Parameter **gamma** must be greater than 0. **gamma** = γ in the equation above.

or

IMSL_SVM_KERNEL_SIGMOID, *float gamma*, *float coef0* (Input)

This argument specifies that the sigmoid kernel type

$$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i^T \mathbf{x}_j + r)$$

is to be used.

float gamma (Input)

Parameter `gamma` = γ in the equation above.

float coef0 (Input)

Parameter `coef0` corresponds to r in the equation above.

or

`IMSLS_SVM_KERNEL_PRECOMPUTED`, *float kernel_values[]* (Input)

Use of this argument indicates that the kernel function values have been precomputed for the training and testing data sets. If `IMSLS_SVM_KERNEL_PRECOMPUTED` is used, the required argument `x` is ignored.

float kernel_values[] (Input)

An array of length `n_patterns` by `n_patterns` containing the precomputed kernel function values. Assume there are L training instances x_1, x_2, \dots, x_L and let $K(x,y)$ be the kernel function value of two instances x and y . Row i of the testing or training data set would be represented by $K(x_i, x_1) K(x_i, x_2) \dots K(x_i, x_L)$. All kernel function values, including zeros, must be provided.

Default: `IMSLS_SVM_KERNEL_RADIAL_BASIS`, `gamma = 1.0/n_attributes`

`IMSLS_SVM_CROSS_VALIDATION`, *int n_folds*, *float **target*, *float *result* (Input/Output)

Conducts cross validation on `n_folds` folds of the data.

`imsls_f_random_uniform_discrete` is used during the cross validation step. See the [Description](#) section for more information on cross validation. See the [Usage Notes](#) in Chapter 12, "Random Number Generation" for instructions on setting the seed to the random number generator if different seeds are desired.

int n_folds (Input)

The number of folds of the data to be used in cross validation. `n_folds` must be greater than 1 and less than `n_patterns`.

*float **target* (Output)

The address of a pointer to an array of length `n_patterns` containing the predicted labels

`float *result` (Output)

If the SVM type used is SVR, `result` contains the mean squared error. For all other SVM types `result` contains the accuracy percentage.

Default: Cross validation is **not** performed.

`IMSLS_SVM_CROSS_VALIDATION_USER`, `int n_folds`, `float target[]`, `float *result`

(Input/Output)

Storage for array `target` is provided by the user. See `IMSLS_SVM_CROSS_VALIDATION` for a description.

Description

Function `imsls_f_support_vector_trainer` trains an SVM classifier for classifying data into one of `n_classes` target classes. There are several SVM formulations that are supported through the optional arguments for classification, regression, and distribution estimation. The C-support vector classification (C-SVC) is the fundamental algorithm for the SVM optimization problem and its primal form is given as

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^l \xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \quad i = 1, \dots, l \end{aligned}$$

Where (\mathbf{x}_i, y_i) are the instance-label pairs for a given training set, where l is the number of training examples, and $\mathbf{x}_i \in R^n$ and $y_i \in \{1, -1\}$. ξ_i are the slack variables in optimization and is an upper bound on the number of errors. The regularization parameter $C > 0$ acts as a tradeoff parameter between error and margin. This is the default algorithm used and can be controlled through the use of the `IMSLS_SVM_C_SVC_TYPE` optional argument.

The ν -support vector classification (ν -SVC) algorithm presents a new parameter $\nu \in (0, 1]$ which acts as an upper bound on the fraction of training errors and a lower bound on the fraction of support vectors. The use of this algorithm is triggered through the use of the `IMSLS_SVM_NU_SVC_TYPE` optional argument. The primal optimization problem for the binary variable $y \in \{1, -1\}$ is

$$\begin{aligned} \min_{w,b,\xi,\rho} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} - \nu \rho + \frac{1}{l} \sum_{i=1}^l \xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + b) \geq \rho - \xi_i, \\ & \xi_i \geq 0, \quad i = 1, \dots, l, \quad \rho \geq 0 \end{aligned}$$

The one-class SVM algorithm estimates the support of a high-dimensional distribution without any class information. Control of this algorithm is through the use of the `IMSLS_SVM_ONE_CLASS_TYPE` optional argument. The primal problem of one-class SVM is

$$\begin{aligned} \min_{w, \xi, \rho} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} - \rho + \frac{1}{vl} \sum_{i=1}^l \xi_i \\ \text{subject to} \quad & \mathbf{w}^T \phi(\mathbf{x}_i) \geq \rho - \xi_i, \\ & \xi_i \geq 0, \quad i = 1, \dots, l \end{aligned}$$

If z_i is the target output and given the parameters $C > 0$, $\varepsilon > 0$, the standard form of ε -support vector regression (ε -SVR) is

$$\begin{aligned} \min_{w, b, \xi, \xi^*} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^l \xi_i + C \sum_{i=1}^l \xi_i^* \\ \text{subject to} \quad & \mathbf{w}^T \phi(\mathbf{x}_i) + b - z_i \leq \varepsilon + \xi_i, \\ & z_i - \mathbf{w}^T \phi(\mathbf{x}_i) - b \leq \varepsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geq 0, \quad i = 1, \dots, l \end{aligned}$$

where the two slack variables ξ_i and ξ_i^* are introduced, one for exceeding the target value by more than ε and the other for being more than ε below the target. The use of this algorithm is triggered through the use of the `IMSLS_SVM_EPSILON_SVR_TYPE` optional argument.

Similar to ν -SVC, in ν -support vector regression (ν -SVR) the parameter $\nu \in (0, 1]$ controls the number of support vectors. Use `IMSLS_SVM_NU_SVR_TYPE` to trigger this algorithm. The ν -SVR primal problem is

$$\begin{aligned} \min_{w, b, \xi, \xi^*, \varepsilon} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C(\nu \varepsilon + \frac{1}{l} \sum_{i=1}^l (\xi_i + \xi_i^*)) \\ \text{subject to} \quad & (\mathbf{w}^T \phi(\mathbf{x}_i) + b) - z_i \leq \varepsilon + \xi_i, \\ & z_i - (\mathbf{w}^T \phi(\mathbf{x}_i) + b) \leq \varepsilon + \xi_i^*, \\ & \xi_i, \xi_i^* \geq 0, \quad i = 1, \dots, l, \quad \varepsilon \geq 0 \end{aligned}$$

The decomposition method used to solve the dual formulation of these primal problems is an SMO-type (sequential minimal optimization) decomposition method proposed by [Fan et. al. \(2005\)](#).

The `IMSL_SVM_CROSS_VALIDATION` optional argument allows one to estimate how accurately the resulting training model will perform in practice. The cross validation technique partitions the training data into `n_folds` complementary subsets. Each of the subsets is subsequently used in training and validated against the remaining subsets. The validation results of the rounds are then averaged. The result is usually a good indicator of how the trained model will perform on unclassified data.

Function `imsls_f_support_vector_trainer` is based on LIBSVM, Copyright (c) 2000-2013, with permission from the authors, Chih-Chung Chang and Chih-Jen Lin, with the following disclaimer:

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Examples

Example 1

In this example, we use a subset of the Fisher Iris data to train the classifier. The default values of `imsls_f_support_vector_trainer` are used in the training. The resultant classifier model, stored in `svm_classifier`, is then used as input to `imsls_f_support_vector_classification` to classify all of the patterns in the Fisher Iris data set. Results of the classification are then printed. In the Fisher Iris data set, the first column is the target classification value, 1=Setosa, 2=Versicolour, and 3=Virginica. Columns 2 through 5 contain the attributes sepal length, sepal width, petal length, and petal width.

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

int main()
{
    int i, ii, j, jj, k, kk, method=1;
    int n_patterns      =150; /* 150 total patterns */
    int n_patterns_train =30; /* 30 training patterns */
    int n_attributes     =4;  /* four attributes */
    int n_classes       =3;  /* three classification categories */
    int *class_errors=NULL;
    float classification[150], *predictedClass=NULL, *xx=NULL;
    float x[150*4], training_data[30*4], training_classification[150];
    float *irisData=NULL; /* Fishers Iris Data */
    float real_min=0.0, real_max=10.0, target_min=0.0, target_max=1.0;
    char *classLabel[] = {"Setosa", "Versicolour", "Virginica"};
    char dashes[] = {
```

```

    "-----"
};
char wspace[] = {" "};
imsls_f_svm_model *svm_classifier=NULL;

/* irisData[]: The raw data matrix. This is a 2-D matrix with 150 */
/*             rows and 5 columns. The first column is the target */
/*             classification value (1-3), and the last 4 columns */
/*             are the continuous input attributes. These data */
/*             contain no categorical input attributes. */
irisData = imsls_f_data_sets(3,0);

/* Data corrections described in the KDD data mining archive */
irisData[5*34+4] = 0.1;
irisData[5*37+2] = 3.1;
irisData[5*37+3] = 1.5;

/* Set up the required input arrays from the data matrix */
for(i=0; i<n_patterns; i++){
    classification[i] = irisData[i*5];
    for(j=1; j<=n_attributes; j++) {
        x[i*n_attributes+j-1] = irisData[i*5+j];
    }
}

/* Scale the data */
xx = imsls_f_scale_filter(n_attributes*n_patterns, x, method,
    IMSLS_SCALE_LIMITS, real_min, real_max, target_min, target_max,
    0);

/* Use a subset of the data for training */
ii = 0;
jj = 0;

printf("    The Input Classification and Training Data \n\n\n");
printf("Classification    Sepal    Sepal    Petal    Petal\n");
printf("    Value            Length    Width    Length    Width\n\n");

for(i=0;i<3;i++){
    kk = 0;
    for(j=0;j<10;j++){
        training_classification[ii] = classification[(i*50)+j];
        printf("    %8.4f        ", training_classification[ii]);
        ii++;
        for(k=0; k<4; k++){
            training_data[jj] = xx[(i*200)+kk++];
            printf("%8.4f ", training_data[jj]);
            jj++;
        }
        printf("\n");
    }
}

/* Train with the training data */
svm_classifier = imsls_f_support_vector_trainer(
    n_patterns_train, n_classes, n_attributes,
    training_classification, training_data, 0);

/* Classify the entire test set */
predictedClass = imsls_f_support_vector_classification(

```

```

svm_classifier, n_patterns, xx,
IMSL_CLASS_ERROR, classification, &class_errors,
0);

printf("\n\n\n    Some Output Classifications\n\n\n");
printf("Pattern      Predicted      Actual\n");
printf("Number      Classification      Classification\n\n");

for (i = 0; i < 10; i++) {
    printf("%2s%d%10s%8.4f%11s%8.4f\n\n",wspace,i,wspace,
        predictedClass[i],wspace,classification[i]);
}

printf("\n\n    Iris Classification Error Rates\n");
printf("%s\n",dashes);
printf("    Setosa  Versicolour  Virginica    |    TOTAL\n");
printf("    %d/%d    %d/%d    %d/%d    |    %d/%d\n",
    class_errors[0], class_errors[1],
    class_errors[2], class_errors[3], class_errors[4],
    class_errors[5], class_errors[6], class_errors[7]);
printf("%s\n\n", dashes);

if (svm_classifier) imsls_f_svm_classifier_free(svm_classifier);
if (predictedClass) imsls_free(predictedClass);
if (class_errors) imsls_free(class_errors);
if (irisData) imsls_free(irisData);
if (xx) imsls_free(xx);
}

```

Output

The Input Classification and Training Data

Classification Value	Sepal Length	Sepal Width	Petal Length	Petal Width
1.0000	0.5100	0.3500	0.1400	0.0200
1.0000	0.4900	0.3000	0.1400	0.0200
1.0000	0.4700	0.3200	0.1300	0.0200
1.0000	0.4600	0.3100	0.1500	0.0200
1.0000	0.5000	0.3600	0.1400	0.0200
1.0000	0.5400	0.3900	0.1700	0.0400
1.0000	0.4600	0.3400	0.1400	0.0300
1.0000	0.5000	0.3400	0.1500	0.0200
1.0000	0.4400	0.2900	0.1400	0.0200
1.0000	0.4900	0.3100	0.1500	0.0100
2.0000	0.7000	0.3200	0.4700	0.1400
2.0000	0.6400	0.3200	0.4500	0.1500
2.0000	0.6900	0.3100	0.4900	0.1500
2.0000	0.5500	0.2300	0.4000	0.1300
2.0000	0.6500	0.2800	0.4600	0.1500
2.0000	0.5700	0.2800	0.4500	0.1300
2.0000	0.6300	0.3300	0.4700	0.1600
2.0000	0.4900	0.2400	0.3300	0.1000
2.0000	0.6600	0.2900	0.4600	0.1300
2.0000	0.5200	0.2700	0.3900	0.1400
3.0000	0.6300	0.3300	0.6000	0.2500
3.0000	0.5800	0.2700	0.5100	0.1900
3.0000	0.7100	0.3000	0.5900	0.2100

3.0000	0.6300	0.2900	0.5600	0.1800
3.0000	0.6500	0.3000	0.5800	0.2200
3.0000	0.7600	0.3000	0.6600	0.2100
3.0000	0.4900	0.2500	0.4500	0.1700
3.0000	0.7300	0.2900	0.6300	0.1800
3.0000	0.6700	0.2500	0.5800	0.1800
3.0000	0.7200	0.3600	0.6100	0.2500

Some Output Classifications

Pattern Number	Predicted Classification	Actual Classification
0	1.0000	1.0000
1	1.0000	1.0000
2	1.0000	1.0000
3	1.0000	1.0000
4	1.0000	1.0000
5	1.0000	1.0000
6	1.0000	1.0000
7	1.0000	1.0000
8	1.0000	1.0000
9	1.0000	1.0000

Iris Classification Error Rates

Setosa	Versicolour	Virginica	TOTAL
0/50	4/50	5/50	9/150

Example 2

In this example we use a subset of the Fisher Iris data to train the classifier and use the cross-validation option with various combinations of `C` and `gamma` to find a combination which yields the best results on the training data. The best combination of `C` and `gamma` are then used to get the classification model, stored in `svm_classifier`. This model is then used as input to `imsls_f_support_vector_classification` to classify all of the patterns in the Fisher Iris data set. Results of the classification are then printed.

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

int main()
{
    int i, ii, j, jj, k, kk, method=1;
    int n_patterns_train    =30; /* 30 training patterns          */
    int n_patterns         =150; /* 150 total patterns      */
    int n_attributes       =4;  /* four attributes         */
    int n_classes          =3;  /* three classification categories */
    int nr_weight          =0;
    int n_folds            =3;
    int degree=0;
    int *class_errors=NULL;
    float C, gamma, coef0=0.0, best_accuracy, best_C, best_gamma, result;
    float *weight_class=NULL;
```

```

float *weight=NULL;
float classification[150], *predictedClass=NULL, *xx=NULL;
float x[150*4], training_data[150*4];
float training_classification[30];
float *irisData=NULL;      /* Fishers Iris Data */
float *target=NULL;
float real_min=0.0, real_max=10.0, target_min=0.0, target_max=1.0;
char *classLabel[] = {"Setosa", "Versicolour", "Virginica"};
char dashes[] = {
    "-----"
};
Imsls_f_svm_model *svm_classifier=NULL;

/* irisData[]: The raw data matrix. This is a 2-D matrix with 150 */
/*              rows and 5 columns. The first column is the target */
/*              classification value (1-3), and the last 4 columns */
/*              are the continuous input attributes. These data */
/*              contain no categorical input attributes. */
irisData = imsls_f_data_sets(3,0);

/* Data corrections described in the KDD data mining archive */
irisData[5*34+4] = 0.1;
irisData[5*37+2] = 3.1;
irisData[5*37+3] = 1.5;

/* Set up the required input arrays from the data matrix */
for(i=0; i<n_patterns; i++){
    classification[i] = irisData[i*5];
    for(j=1; j<n_attributes; j++) {
        x[i*n_attributes+j-1] = irisData[i*5+j];
    }
}

/* Scale the data */
xx = imsls_f_scale_filter(n_attributes*n_patterns, x, method,
    IMSLS_SCALE_LIMITS, real_min, real_max, target_min, target_max,
    0);

/* Use a subset of the data for training */
ii = 0;
jj = 0;
for(i=0; i<3; i++){
    kk = 0;
    for(j=0; j<10; j++){
        training_classification[ii++] = classification[(i*50)+j];
        for(k=0; k<4; k++){
            training_data[jj] = xx[(i*200)+kk];
            kk++;
            jj++;
        }
    }
}
C = 2.0;

/* Try different combinations of C and gamma to settle on model parameters */
best_accuracy = 0.0;
best_C = 0.0;
best_gamma = 0.0;
for(i=0; i<10; i++){
    gamma = .1;

```

```

    for(j=0;j<5;j++){
        svm_classifier = imsls_f_support_vector_trainer(
            n_patterns_train, n_classes, n_attributes, training_classification,
            training_data,
            IMSLS_SVM_C_SVC_TYPE, C, nr_weight, weight_class, weight,
            IMSLS_SVM_KERNEL_RADIAL_BASIS, gamma,
            IMSLS_SVM_CROSS_VALIDATION, n_folds, &target, &result,
            0);
        if(result > best_accuracy){
            best_accuracy = result;
            best_C = C;
            best_gamma = gamma;
        }
        gamma = gamma*2.0;
        imsls_f_svm_classifier_free(svm_classifier);
        if(target) imsls_free(target);
    }
    C = C*2.0;
}

/* Train with the best resultant parameters */
svm_classifier = imsls_f_support_vector_trainer(
    n_patterns_train, n_classes, n_attributes, training_classification,
    training_data,
    IMSLS_SVM_C_SVC_TYPE, best_C, nr_weight, weight_class, weight,
    IMSLS_SVM_KERNEL_RADIAL_BASIS, best_gamma,
    0);

/* Call SUPPORT_VECTOR_CLASSIFICATION on the entire test set */
predictedClass = imsls_f_support_vector_classification(
    svm_classifier, n_patterns, xx,
    IMSLS_CLASS_ERROR, classification, &class_errors,
    0);

printf("    Iris Classification Error Rates\n");
printf("%s\n", dashes);
printf("    Setosa Versicolour Virginica |    TOTAL\n");
printf("    %d/%d    %d/%d    %d/%d    |    %d/%d\n",
    class_errors[0], class_errors[1],
    class_errors[2], class_errors[3], class_errors[4],
    class_errors[5], class_errors[6], class_errors[7]);
printf("%s\n\n", dashes);

if (svm_classifier) imsls_f_svm_classifier_free(svm_classifier);
if (predictedClass) imsls_free(predictedClass);
if (class_errors) imsls_free(class_errors);
if (irisData) imsls_free(irisData);
if (xx) imsls_free(xx);
}

```

Output

Iris Classification Error Rates

```

-----
Setosa Versicolour Virginica | TOTAL
0/50    1/50    3/50    | 4/150

```

Example 3

One thousand uniform deviates from a uniform distribution are used in the training data set of this example. `IMSL_SVM_ONE_CLASS_TYPE` is used to produce the model during training. A test data set of one hundred uniform deviates is produced and contaminated with ten normal deviates.

`imsls_f_support_vector_classification` is then called in an attempt to pick out the contaminated data in the test data set. The suspect observations are printed.

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

#define N_PATTERNS_TRAIN 1000
#define N_PATTERNS_TEST 100
#define N_PATTERNS_TEN 10
#define N_CLASSES 1
#define N_ATTRIBUTES 1

int main()
{
    int i;
    float *target=NULL;
    float classification_train[N_PATTERNS_TRAIN];
    float classification_test[N_PATTERNS_TEST];
    float *x_train;
    float *x_test;
    float *x_test_contaminant;

    Imsls_f_svm_model *svm_classifier=NULL;

    /* Create the training set from a uniform distribution */

    imsls_random_seed_set(123457);
    x_train = imsls_f_random_uniform(N_PATTERNS_TRAIN, 0);
    for(i=0;i<N_PATTERNS_TRAIN;i++)
        classification_train[i] = 1.0;

    svm_classifier = imsls_f_support_vector_trainer(N_PATTERNS_TRAIN, N_CLASSES,
        N_ATTRIBUTES, classification_train, x_train,
        IMSLS_SVM_ONE_CLASS_TYPE, .001,
        0);

    /* Create a testing set from a uniform distribution */

    x_test = imsls_f_random_uniform(N_PATTERNS_TEST, 0);
    for(i=0;i<N_PATTERNS_TEST;i++)
        classification_test[i] = 1.0;

    /* Contaminate the testing set with deviates from a normal distribution */

    x_test_contaminant = imsls_f_random_normal(N_PATTERNS_TEN,
        IMSLS_MEAN, .1,
        IMSLS_VARIANCE, .2,
        0);
    for(i=0;i<N_PATTERNS_TEN;i++)
        x_test[i*10] = x_test_contaminant[i];
}
```

```

target = imsls_f_support_vector_classification(svm_classifier,
      N_PATTERNS_TEST, x_test, 0);

printf("\n\n\n          Classification Results \n\n\n");
for(i=0; i<N_PATTERNS_TEST; i++){
    if (target[i]!=1.0){
        printf("The %d-th observation may not to belong to the",i);
        printf(" target distribution.\n");
    }
}

if (svm_classifier) imsls_f_svm_classifier_free(svm_classifier);
if (target) imsls_free(target);
if (x_train) imsls_free(x_train);
if (x_test) imsls_free(x_test);
if (x_test_contaminant) imsls_free(x_test_contaminant);
}

```

Output

```

          Classification Results

The 0-th observation may not belong to the target distribution.

The 20-th observation may not belong to the target distribution.

The 30-th observation may not belong to the target distribution.

The 40-th observation may not belong to the target distribution.

The 60-th observation may not belong to the target distribution.

The 70-th observation may not belong to the target distribution.

```

Example 4

This example uses `IMSLS_SVM_NU_SVR_TYPE` to create a regression model which is used by `imsls_f_support_vector_classification` in an attempt to predict values in the test data set. The predicted values are printed.

```

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

#define N_PATTERNS_TRAIN  10
#define N_PATTERNS_TEST   4
#define N_CLASSES         2
#define N_ATTRIBUTES      2

int main()
{
    int    i;
    float  C=50., nu=.01, diff, mse=0.0, *target=NULL;
    float  classification_train[] = {1.0,1.0,1.0,1.0,1.0,2.0,2.0,2.0,2.0,2.0};
    float  classification_test[]  = {1.0,1.0,2.0,2.0};

```

```

float x_train[] = { 0.19, 0.61,
  0.156, 0.564,
  0.224, 0.528,
  0.178, 0.51,
  0.234, 0.578,
  0.394, 0.296,
  0.478, 0.254,
  0.454, 0.294,
  0.48, 0.358,
  0.398, 0.336};
float x_test[] = {
  0.316, 0.556,
  0.278, 0.622,
  0.562, 0.336,
  0.522, 0.412};

Imsls_f_svm_model *svm_classifier=NULL;

svm_classifier = imsls_f_support_vector_trainer(N_PATTERNS_TRAIN,
  N_CLASSES, N_ATTRIBUTES, classification_train, x_train,
  IMSLS_SVM_NU_SVR_TYPE, C, nu,
  0);

target = imsls_f_support_vector_classification(svm_classifier,
  N_PATTERNS_TEST, x_test, 0);

mse = 0.0;

printf("Predicted    Actual    Difference \n");
for(i=0;i<N_PATTERNS_TEST;i++){
  diff = (target[i] - classification_test[i]);
  printf("%f    %f    %f \n",target[i], classification_test[i], diff);
  mse = mse + (diff*diff);
}
mse = mse/N_PATTERNS_TEST;

printf("\n The Mean squared error for the predicted values is %f \n",
  mse);
if (svm_classifier) imsls_f_svm_classifier_free(svm_classifier);
if (target) imsls_free(target);
}

```

Output

Predicted	Actual	Difference
1.443569	1.000000	0.443569
1.397248	1.000000	0.397248
1.648531	2.000000	-0.351469
1.598311	2.000000	-0.401689

The Mean squared error for the predicted values is 0.159861

Warning Errors

IMSLS_OPTION_NOT_SUPPORTED	The optional argument # is not supported for #.
IMSLS_LABEL_NOT_FOUND	The class label # specified in "weight" not found.
IMSLS_INADEQUATE_MODEL	The model used contains inadequate information to compute the requested probability.
IMSLS_TWO_CLASS_LINE_SEARCH	The line search failed in a two-class probability estimation while performing cross validation.
IMSLS_VALIDATION_MAX_ITERATIONS	The maximum number of iterations was reached in a #-class probability estimation while performing cross validation.

support_vector_classification

Classifies unknown patterns using a previously trained Support Vector Machines (SVM) model computed by [imsls_f_support_vector_trainer](#).

Synopsis

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

```
float *imsls_f_support_vector_classification( imsls_f_svm_model *svm_classifier,  
int n_patterns, float x[ ], ..., 0)
```

The type *double* function is `imsls_d_support_vector_classification`.

Required Arguments

imsls_f_svm_model *svm_classifier (Input)

Pointer to a structure of type *imsls_f_svm_model* from `imsls_f_support_vector_trainer`.

int n_patterns (Input)

Number of patterns to classify.

float x[] (Input)

An array of length `n_patterns` by `n_attributes` containing the data matrix where `n_attributes` is the number of attributes as specified in `imsls_f_support_vector_trainer`.

Return Value

Pointer to an array of length `n_patterns` containing the predicted classification associated with each input pattern for a classification model, or the calculated function value for a regression model. If classification is unsuccessful, `NULL` is returned. To release this space, use [imsls_free](#).

Synopsis with Optional Arguments

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

```

float *imsls_f_support_vector_classification(Imsls_f_svm_model *svm_classifier,
int n_patterns, float x[],

    IMSLS_SVM_KERNEL_PRECOMPUTED, float kernel_values[],

    IMSLS_PREDICTED_CLASS_PROB, float **pred_class_prob,

    IMSLS_PREDICTED_CLASS_PROB_USER, float pred_class_prob[],

    IMSLS_SVR_PROBABILITY, float *svr_probability,

    IMSLS_CLASS_ERROR, float classification[], int **class_errors,

    IMSLS_CLASS_ERROR_USER, float *classification, int class_errors[],

    IMSLS_DECISION_VALUES, int i, float **dec_values,

    IMSLS_DECISION_VALUES_USER, int i, float dec_values[],

    IMSLS_RETURN_USER, float predicted_labels[],

    0)

```

Optional Arguments

`IMSLS_SVM_KERNEL_PRECOMPUTED, float kernel_values[]` (Input)

Use of this argument indicates that the kernel function values have been precomputed for the training and testing data sets. If `IMSLS_SVM_KERNEL_PRECOMPUTED` is used, the required argument `x` is ignored.

`float kernel_values[]` (Input)

An array of length `n_patterns` by `n_patterns` containing the precomputed kernel function values. Assume there are L testing instances x_1, x_2, \dots, x_L and let $K(x, y)$ be the kernel function value of two instances x and y . Row i of the testing or training data set would be represented by $K(x_i, x_1) K(x_i, x_2) \dots K(x_i, x_L)$. All kernel function values, including zeros, must be provided.

`IMSLS_PREDICTED_CLASS_PROB, float **pred_class_prob` (Output)

The address of a pointer to an array of length `n_patterns` by `n_classes`, where `n_classes` is the number of target classifications. The values in the i -th row are the predicted classification probabilities associated with the target classes. `pred_class_prob[i*n_classes+j]` is the estimated probability that the i -th pattern belongs to the j -th target class.

For regression and one-class SVMs, the array `pred_class_prob` is `NULL`.

To release this space, use `imsls_free`.

`IMSLS_PREDICTED_CLASS_PROB_USER`, *float pred_class_prob* [] (Output)

Storage for array `pred_class_prob` is provided by the user. See `IMSLS_PREDICTED_CLASS_PROB` for a description.

`IMSLS_SVR_PROBABILITY`, *float *svr_probability* (Output)

For a regression model with probability information, this option outputs a value $\sigma > 0$. For the test data, we consider the probability model as

$$\text{target value} = \text{predicted value} + z$$

where z is distributed according to Laplace with zero-mean density function $p(x) = e^{-|x|/\sigma}/2\sigma$.

`svr_probability` contains σ on output. If the model is not for SVR or does not contain the required probability information, 0 is returned.

`IMSLS_CLASS_ERROR`, *float classification* [], *int **class_errors* (Output)

Returns classification error counts.

float classification [] (Input)

An array of length `n_patterns` containing the known classifications for each of the patterns.

*int **class_errors* (Output)

The address of a pointer to an array of length `(n_classes+1)` by 2 containing the number of classification errors and the number of non-missing classifications for each target classification, plus the overall totals for these errors. For $i < \text{n_classes}$, the i -th row contains the number of classification errors for the i -th class and the number of patterns with non-missing classifications for that class. The last row contains the number of classification errors totaled over all target classifications, and the total number of patterns with non-missing target classifications. To release this space, use `imsls_free`.

`IMSLS_CLASS_ERROR_USER`, *float classification* [], *int class_errors* [] (Output)

Storage for `class_errors` is provided by the user. See `IMSLS_CLASS_ERROR` for a description.

`IMSLS_DECISION_VALUES`, *int i*, *float **dec_values* (Output)

Gives decision values on row i of data matrix \mathbf{x} . The decision values are used to predict the target classification value. For binary class problems, decision values are signed values used to determine which side of the decision boundary the observation in question lays. Recall that SVM combines multiple binary classifiers for multi-class classification. The magnitudes of the decision values are essentially meaningless as only the sign is used to determine whether the point is more likely to

belong to $class[i]$ versus $class[j]$ (a positive value) or less likely to belong to $class[i]$ versus $class[j]$ (a negative value). See the [Description](#) section for more on how the decision values are used to predict the target classification value.

For a classification model with $n_classes$ classes, this function gives $n_classes*(n_classes-1)/2$ decision values in the array `dec_values`. The order is $class[0]$ vs. $class[1]$, ..., $class[0]$ vs. $class[n_classes-1]$, $class[1]$ vs. $class[2]$, ..., $class[1]$ vs. $class[n_classes-1]$, ..., $class[n_classes-2]$ vs. $class[n_classes-1]$, where $class$ contains the target classification values.

For a regression model, only 1 value is returned. `dec_values[0]` is the function value of row i of x calculated using the model. For a one-class model, `dec_values[0]` is the decision value of row i of x . To release this space, use `imsls_free`

`IMSLS_DECISION_VALUES_USER`, *int* i , *float* `dec_values[]` (Output)

Storage for array `dec_values` is provided by the user. See `IMSLS_DECISION_VALUES` for a description.

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

Storage for the return values is provided by the user in array `predicted_labels`. `predicted_labels` must be at least of length `n_patterns`.

Description

Function `imsls_f_support_vector_classification` estimates classification probabilities from a previously trained SVM model. This function does classification or regression on a test vector x using the SVM model output from `imsls_f_support_vector_trainer`. For a classification model, the predicted class for x is returned. For a regression model, the function value of x calculated using the model is returned. For a one-class model, +1 or -1 is returned signifying that the observation belongs to (+1) or does not belong to (-1) the expected class. The predicted classification returned by this function is the class with the largest estimated classification probability. The classification probability estimates for each pattern can be obtained using the optional argument `IMSLS_PREDICTED_CLASS_PROB`.

The decision values which are returned through the use of optional argument `IMSLS_DECISION_VALUES` are used to cast votes for the target classification value. Only the sign of the decision value is used in casting votes. For a decision value for $class[i]$ versus $class[j]$, a positive value is interpreted as $class[i]$ being more likely than $class[j]$, so $class[i]$ would get a vote. A negative value is interpreted as $class[i]$ being less likely than $class[j]$, so $class[j]$ would get a vote. The votes along a row of decision values are tallied to elect the class with the most votes as the target classification value. In case of a tie, the first class encountered with the highest number of votes in a row of decision values is the predicted class for that row.

Function `imsls_f_support_vector_classification` is based on LIBSVM, Copyright (c) 2000-2013, with permission from the authors, Chih-Chung Chang and Chih-Jen Lin with the following disclaimer:

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Examples

See [Examples](#) in `imsls_f_support_vector_trainer`.

Warning Errors

`IMSL_S_INADEQUATE_MODEL`

The model used contains inadequate information to compute the requested probability.

svm_classifier_free

Frees memory allocated to an *ImSls_f_svm_model* data structure.

Synopsis

```
#include <imsls.h>

void imsls_f_svm_classifier_free (ImSls_f_svm_model *svm_classifier)
```

The type *double* function is `imsls_d_svm_classifier_free`.

Required Arguments

ImSls_f_svm_model *svm_classifier (Input)
Pointer to a structure of the type *ImSls_f_svm_model* from [imsls_f_support_vector_trainer](#).

Description

An *ImSls_f_svm_model* data structure is created by `imsls_f_support_vector_trainer`. Function `imsls_f_svm_classifier_free` is used to free the memory allocated to this structure when the structure is no longer needed.

Printing Functions

Functions

Print a matrix or vector	write_matrix	1798
Set the page width and length	page	1805
Set the printing options	write_options	1807

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, (Input)
 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.

Default: `IMSLS_PRINT_ALL`.

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 con-

version 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, (Input)

Indicates that no row labels are used.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., **nra** are used for the row labels whenever **nra** > 1. If **nra** = 1, the default is no row labels.

or

IMSLS_ROW_NUMBER, (Input)

Indicates the numbers 1, 2, 3, ..., **nra** are used for the row labels.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., **nra** are used for the row labels whenever **nra** > 1. If **nra** = 1, the default is no row labels.

or

IMSLS_ROW_NUMBER_ZERO, (Input)

Indicates the numbers 1, 2, 3, ..., **nra**-1 are used for the row labels.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., **nra** are used for the row labels whenever **nra** > 1. If **nra** = 1, the default is no row labels.

or

IMSLS_ROW_LABELS, *char *rlabel[]* (Input)

Indicates **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.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., **nra** are used for the row labels whenever **nra** > 1. If **nra** = 1, the default is no row labels.

IMSLS_NO_COL_LABELS, (Input)

Indicates that no column labels are used.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., **nca** are used for the column labels whenever **nca** > 1. If **nca** = 1, the default is no column labels.

or

IMSLS_COL_NUMBER, (Input)

Indicates the numbers 1, 2, 3, ..., **nca** are used for the column labels.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., `nca` are used for the column labels whenever `nca` > 1. If `nca` = 1, the default is no column labels.

or

`IMSLS_COL_NUMBER_ZERO`, (Input)

Indicates the numbers 1, 2, 3, ..., `nca`-1 are used for the column labels.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., `nca` are used for the column labels whenever `nca` > 1. If `nca` = 1, the default is no column labels.

or

`IMSLS_COL_LABELS`, *char *clabel[]* (Input)

Indicates `clabel` is a vector of length `nca` + 1 containing pointers to the character strings comprising the column headings. The heading for the column labels is `clabel` [0]; `clabel` [*i*], *i* = 1, ..., `nca`, is the heading for the *i*-th column. Use `\n` within a label to create a new line. Long labels are automatically wrapped.

Default: If none of these optional arguments is used, the numbers 1, 2, 3, ..., `nca` are used for the column labels 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

int 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

int main()
{
    int        i, j;
    float      a[NRA][NCA];
    char       *fmt = "%10.6W";
    char       *rlabel[] = {"row 1", "row 2", "row 3"};
    char       *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};
}
```

```

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }
    }

    /* Write matrix */
    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

int main()
{
    int        i;
    float      a[NCA];
    char       *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0; i < NCA; i++) {
        a[i] = i + 1;
    }

    /* Write matrix */
    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. Possible values are shown 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
int main()
{
    int    i, j, page_attribute;
    float  a[NRA][NCA];
```

```
    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
    page_attribute = 40;
    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		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 <code>imsls_write_matrix</code> begins on a new page, and paging occurs within each invocation as is needed.
	0	Paging is on. The first invocation of an <code>imsls_f_write_f_matrix</code> function begins on a new page, and subsequent paging occurs as is needed. Paging occurs in the second and all subsequent calls to an <code>imsls_f_write_matrix</code> 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 <code>imsls_write_matrix</code> 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

int main()
{
    int      i, j, option_value;
    float    a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }

    /* Activate centering option */
    option_value = 1;
    imsls_write_options (IMSL_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 (IMSL_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	

Utilities

Functions

Set Output Files

Sets output files	output_file	1813
Gets library version and license number	version	1817

Error Handling

Error message options.	error_options	1819
Gets error code	error_code	1826
Gets error type.	error_type	1828
Gets error message.	error_message	1829
Initializes error handling system	initialize_error_handler	1831
Stops the current algorithm and returns to the calling programset_user_fcn_return_flag	1834

C Runtime Library

Frees memory	free	1839
Opens a file	fopen	1841
Closes a file	fclose	1843
Reads ASCII files.	ascii_read	1844

OpenMP

OpenMP options	omp_options	1863
--------------------------	-----------------------------	------

Constants

Integer machine constants.	machine (integer)	1865
Float machine constantsmachine (float)	1868
Common data sets.data_sets	1871

Mathematical Support

Matrix-vector, matrix-matrix, vector-vector productsmat_mul_rect	1875
Rearrange elements of vector	permute_vector	1880
Interchange rows and columns of matrices	permute_matrix	1882
Locate and optionally replace dependent variable missing values with nearest neighbor estimates	impute_missing	1885
Evaluate the binomial coefficientbinomial_coefficient	1895
Evaluate the complete beta function	beta	1897
Evaluate the real incomplete beta function	beta_incomplete	1900

Evaluate the log of the real beta function	log_beta	1902
Evaluate the real gamma function	gamma	1904
Evaluate the incomplete gamma function	gamma_incomplete	1907
Evaluate the logarithm of the absolute value of the gamma function	log_gamma	1910
Return the number of CPU seconds used	ctime	1913

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.

Examples

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

extern FILE* imsls_fopen(char* filename, char* mode);
extern int imsls_fclose(FILE* file);

int main()
{
    FILE *ofile;
    float x[] = {3.0, 2.0, 1.0};

    imsls_f_write_matrix ("x (default file)", 1, 3, x,
        0);

    ofile = imsls_fopen("myfile", "w");

    imsls_output_file(IMSLs_SET_OUTPUT_FILE, ofile,
        0);
    imsls_f_write_matrix ("x (myfile)", 1, 3, x,
        0);

    imsls_fclose(ofile);
}
```

Output

```
      x (default file)
      1          2          3
      3          2          1
```

File myfile

```
x (myfile)
1          2          3
3          2          1
```

Example 2

This example illustrates how to direct output from IMSL routines that run in separate threads to different files. First, two threads are created, each calling a different IMSL function, then the results are printed by calling `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 <stdlib.h>
#include <imsls.h>

void *ex1(void* arg);
void *ex2(void* arg);

extern FILE* imsls_fopen(char* filename, char* mode);
extern int imsls_fclose(FILE* file);

int main()
{
    pthread_t    thread1;
    pthread_t    thread2;

    /* Create two threads. */
    if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
        perror("pthread_create"), exit(1);
    if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
        perror("pthread_create"), exit(1);

    /* Wait for threads to finish. */
    if (pthread_join(thread1, NULL) != 0)
        perror("pthread_join"),exit(1);
    if (pthread_join(thread2, NULL) != 0)
        perror("pthread_join"),exit(1);
}

void *ex1(void *arg)
{
    float *rand_nums = NULL;
    FILE *file_ptr;

    /* Open a file to write the result in. */
    file_ptr = imsls_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)
        imsls_free(rand_nums);

    imsls_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 = imsls_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)
        imsls_free(table);

    imsls_fclose(file_ptr);
}

```

Output

The content of the file ex1.out is shown below.

```

Random Numbers
1      0.4919
2      0.3909
3      0.2645
4      0.1814
5      0.7546

```

The content of the file ex2.out is shown below.

```

                                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 `IMSL_S_LIBRARY_VERSION`, `IMSL_S_OS_VERSION`, `IMSL_S_COMPILER_VERSION`, or `IMSL_S_LICENSE_NUMBER`.

Return Value

The requested value is returned. If `code` is out of range, then `NULL` is returned. Use `imsls_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>
#include <stdio.h>

int 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);  
return 0;  
}
```

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,
    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.

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. See [Example 3](#) and [Example 4](#) for multithreaded examples.

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>

int 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("IMSL_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.

```

```

IMSL_TERMINAL error print setting = 1

```

Example 2

In this example, IMSL's error printing function has been substituted for the standard function. Only the first four lines are printed below.

```

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

void      print_proc(Imsls_error, long, char*, char*);

int main()
{
    /* Turn off tracebacks on IMSLS_TERMINAL */
    /* error messages and use a custom */
    /* print function */
    imsls_error_options(IMSL_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. 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);
int main()
{
    pthread_t    thread1;
    pthread_t    thread2;

    /* Create two threads. */
    if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
        perror("pthread_create"), exit(1);
    if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
        perror("pthread_create"), exit(1);

    /* Wait for threads to finish. */
    if (pthread_join(thread1, NULL) != 0)
        perror("pthread_join"),exit(1);
    if (pthread_join(thread2, NULL) != 0)
        perror("pthread_join"),exit(1);
}

void *ex1(void* arg)
{
    float res;
    /*
     * Call 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];

    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>

int 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
```

error_type

Gets the type corresponding to the error message from the last function called.

Synopsis

```
#include <imsls.h>

Imsls_error imsls_error_type ( )
```

Return Value

An *Imsls_error* enum value is returned.

Description

The *Imsls_error* enum type has seven values: `IMSLS_NOTE`, `IMSLS_ALERT`, `IMSLS_WARNING`, `IMSLS_FATAL`, `IMSLS_TERMINAL`, `IMSLS_WARNING_IMMEDIATE` and `IMSLS_FATAL_IMMEDIATE`. See [Kinds of Errors and Default Actions](#) for more details.

Example

See [error_message](#) for an example.

error_message

Gets the text of the error message from the last function called.

Synopsis

```
#include <imsls.h>

char *imsls_error_message ()
```

Return Value

Returns the current error message.

Description

If the current error type is positive then the last error message set is returned. It does not matter if the error message was printed or not. The current error type is the number returned by [imsls_error_type](#). If the current error type is zero then `NULL` is returned.

The returned string can be freed using [imsls_free](#).

Example

This example retrieves the error message from a call to [imsls_f_wilcoxon_rank_sum](#) with an illegal argument.

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

int main(void)
{
    char *msg;
    float x[] = {0, 1, 2};
    float y[] = {0, 1, 2};
    float p;

    p = imsls_f_wilcoxon_rank_sum (3, x, 3, y, 0);
    msg = imsls_error_message();
    printf("type = %d\ncode = %d\nmsg = %s\n",
           imsls_error_type(), imsls_error_code(), msg);
    imsls_free(msg);
}
```

Output

```
*** WARNING Error IMSLS_AT_LEAST_ONE_TIE from imsls_f_wilcoxon_rank_sum.  
***           At least one tie is detected between the samples.
```

```
type = 3  
code = 11123  
msg = At least one tie is detected between the samples.
```

initialize_error_handler

Initializes the IMSL C Stat Library error handling system.

Synopsis

```
#include <imsls.h>

int imsls_initialize_error_handler ()
```

Return Value

If the initialization succeeds, zero is returned. If there is an error, a nonzero value is returned.

Description

This function is used to initialize the IMSL C Stat Library error handling system for the current thread. It is not required, but is always allowed.

Use of this function is advised if the possibility of low heap memory exists when calling IMSL C Stat Library for the first time in the current thread. A successful return from `imsls_initialize_error_handler` confirms that the IMSL C Stat Library error handling system has been initialized and is operational. The effects of calling `imsls_initialize_error_handler` are limited to the calling thread only.

If `imsls_initialize_error_handler` is not called and initialization of the error handling system fails, an error message is printed to `stderr`, and execution is stopped.

Example

In this example, the IMSL C Stat Library error handler is initialized prior to calling multiple other IMSL C Stat Library functions. Often this is not required, but is advised if the possibility of low heap memory exists. Even if not required, the initialization call is always allowed.

The computations performed in this example are based on [Example 1](#) for `imsls_f_regression_prediction`.

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

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

    int          status;
    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};

    /* Initialize the IMSL C Math Library error handler. */
    status = imsls_initialize_error_handler();

    /*
     * Verify successful error handler initialization before
     * continuing.
     */
    if (status == 0) {

        /* 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);

    } else {
        printf("Unable to initialize IMSL C Math Library error handler.\n");
    }
}
```

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					

set_user_fcn_return_flag

Indicates a condition has occurred in a user-supplied function necessitating a return to the calling function.

Synopsis

```
#include <imsls.h>

void imsls_set_user_fcn_return_flag(int code)
```

Required Arguments

int code (Input)
A user-defined number that indicates the reason for the return from the user-supplied function.

Description

Given a certain condition in a user-supplied function, `imsls_set_user_fcn_return_flag` stops executing any IMSL algorithm that has called the function, and returns to the calling function or main program. On invocation of `imsls_set_user_fcn_return_flag`, a flag is set in the IMSL error handler. On returning from the user-supplied function, the error `IMSL_STOP_USER_FCN` is issued with severity `IMSL_FATAL`. Typically, if you use this function, you would disable stopping on `IMSL C STAT` errors, thus gaining greater control in situations where you need to prematurely return from an algorithm. (See Programming Notes.)

Programming Notes

1. Since the default behavior of IMSL error handling is to stop execution on `IMSL_TERMINAL` and `IMSL_FATAL` errors, execution of the main program stops when the `IMSL_STOP_USER_FCN` error message is issued unless you alter this behavior by turning stopping off using `imsls_error_options`.
2. In a user-supplied function, the user is responsible for checking error conditions such as memory allocation, return status for any function calls, valid return values, etc.
3. Use of this function is valid only if called from within a user-supplied function.

Examples

Example 1

This example is based on [imsls_f_kolmogorov_one](#). In this example, the user, for any hypothetical reason, wants to stop the evaluation of the user-supplied function, `cdf` when `x` is greater 0.5.

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

float cdf(float);

int main()
{
    float *statistics = NULL, *diffs = NULL, *x = NULL;
    int nobs = 100, nmiss;

    imsls_random_seed_set(123457);

    x = imsls_f_random_uniform(nobs, 0);

    /* Turn off stopping on IMSLS_FATAL errors. */
    imsls_error_options(IMSLs_SET_STOP, IMSLS_FATAL, 0, 0);

    statistics = imsls_f_kolmogorov_one(cdf, nobs, x,
        IMSLS_N_MISSING, &nmiss,
        IMSLS_DIFFERENCES, &diffs,
        0);

    /* The following lines will be executed because
       stopping is turned off. */
    if (statistics) {
        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);
    } else {
        printf("\"statistics\" is NULL.\n");
    }
}

float cdf(float x)
{
    float mean = .5, std = .2886751, z, result;

    /* For a hypothetical reason, stop execution when x > 0.5. */
    if (x > 0.5) {
        imsls_set_user_fcn_return_flag(1);
        return 0;
    }

    z = (x-mean)/std;
    result = imsls_f_normal_cdf(z);
}
```

```

    return result;
}

```

Output

```

*** FATAL   Error IMSLS_STOP_USER_FCN from imsls_f_kolmogorov_one. Request
***          from user supplied function to stop algorithm. User flag = "1".

"statistics" is NULL.

```

Example 2

This example is based on `imsls_f_chi_squared_test`, Example 3. This example demonstrates how to handle the error condition if the user-supplied function calls a C Stat Library function. In this example, `THETA` is set to 0 to force an error condition in calling the `imsls_f_poisson_cdf` function in the user-supplied function.

```

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

#define SEED                123457
#define N_CATEGORIES        10
#define N_PARAMETERS_ESTIMATED 0
#define N_NUMBERS           1000
#define THETA                0.0

float      user_proc_cdf(float);

int main()
{
    int      i, *poisson;
    float    cell_statistics[3][N_CATEGORIES];
    float    chi_squared_statistics[3], x[N_NUMBERS];
    float    cutpoints[] = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5,
        7.5, 8.5, 9.5};
    char     *cell_row_labels[] = {"count", "expected count",
        "cell chi-squared"};
    char     *cell_col_labels[] = {"Poisson value", "0", "1", "2",
        "3", "4", "5", "6", "7",
        "8", "9"};
    char     *stat_row_labels[] = {"chi-squared",
        "degrees of freedom", "p-value"};

    /* Turn off stopping on IMSLS_FATAL errors. */
    imsls_error_options(IMSL_SET_STOP, IMSLS_FATAL, 0, 0);

    imsls_random_seed_set(SEED);

    /* Generate the data */
    poisson = imsls_random_poisson(N_NUMBERS, 5.0, 0);

    /* Copy data to a floating point vector*/
    for (i = 0; i < N_NUMBERS; i++)
        x[i] = poisson[i];

    chi_squared_statistics[2] =
        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);

/* The following lines will be executed because
stopping is turned off. */
if (chi_squared_statistics[2] != chi_squared_statistics[2]) {
    printf("p-value = NaN\n");
} else {
    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;
    int        setting;

    /* The user is responsible for checking error conditions in the
user-supplied function, even if the user-supplied function
is calling an IMSL function.

For theta = 0.0 (an invalid input), imsls_f_poisson_cdf issues
an IMSLS_TERMINAL error. Thus, stopping is turned off on
IMLS_TERMINAL error. */

    /* Get the current terminal error stopping setting which will be
used for restoring the setting later. */
    imsls_error_options(IMSL_GET_STOP, IMSLS_TERMINAL, &setting, 0);

    /* Disable stopping on terminal error. */
    imsls_error_options(IMSL_SET_STOP, IMSLS_TERMINAL, 0, 0);

    cdf_v = imsls_f_poisson_cdf ((int) k, THETA);

    /* If there is terminal error, stop and return to main. */
    if (imsls_error_type() == IMSLS_TERMINAL) {
        imsls_set_user_fcn_return_flag(1);
        return 0;
    }

    /* Restore stopping setting */
    imsls_error_options(IMSL_SET_STOP, IMSLS_TERMINAL, setting, 0);
}

```

```
    return cdf_v;  
}
```

Output

```
*** TERMINAL Error from imsls_f_poisson_cdf. The mean of the Poisson  
***          distribution, "theta" = 0.000000e+000, must be positive.
```

```
*** FATAL    Error IMSLS_STOP_USER_FCN from imsls_f_chi_squared_test.  
***          Request from user supplied function to stop algorithm. User  
***          flag = "1".
```

```
p-value = NaN
```

free

Frees memory returned from an IMSL C Stat Library function.

Synopsis

```
#include <imsls.h>

void imsls_free(void *data)
```

Required Arguments

*void *data* (Input)
A pointer to data returned from an IMSL C Stat Library function.

Description

The function `imsls_free` frees memory using the C runtime library used by the IMSL C Stat Library for allocation. It is a wrapper around the standard C runtime function `free`.

Function `imsls_free` can always be used to free memory allocated by the IMSL C Stat Library, but is required if an application has linked to multiple copies of the C runtime library, with each copy having its own set of heap allocation structures. In this situation, using the C runtime function `free` can result in memory being allocated with one copy of the C runtime library and freed with a different copy, which may cause abnormal termination. Using `imsls_free` ensures that the same C runtime library is used for both allocation and freeing.

Note that `imsls_free` should be used only to free memory that was allocated by IMSL C Stat Library.

Example

This example computes a set of random numbers, prints them, and then frees the array returned from the random number generation function.

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

int main()
{
    int seed = 123457;
```

```
int n_random = 5;
float *r;

imsls_random_seed_set (seed);
r = imsls_f_random_normal(n_random, 0);
printf("%s: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
       "Standard normal random deviates",
       r[0], r[1], r[2], r[3], r[4]);

imsls_free(r);
}
```

Output

```
Standard normal random deviates:  1.8279 -0.6412 0.7266 0.1747 1.0145
```

fopen

Opens a file using the C runtime library used by the IMSL C Stat Library.

Synopsis

```
#include <imsls.h>

#include <stdio.h>

FILE *imsls_fopen(char *filename, char *mode)
```

Required Arguments

`char *filename` (Input)
The name of the file to be opened.

`char *mode` (Input)
The type of access to be permitted to the file. This string is passed to the C runtime function `fopen`, which determines the valid `mode` values.

Return Value

A pointer to the file structure, `FILE`, defined in `stdio.h`. To close the file, use `imsls_fclose`.

Description

The function `imsls_fopen` opens a file using the C runtime library used by the IMSL C Stat Library. It is a wrapper around the standard C runtime function `fopen`.

Function `imsls_fopen` can always be used to open a file which will be used by the IMSL C Stat Library, but is required if an application has linked to multiple copies of the C runtime library, with each copy having its own set of file instructions. In this situation, using the C runtime function `fopen` can result in a file being opened with one copy of the C runtime library and reading or writing to it with a different copy, which may cause abnormal behav-

ior or termination. Using `imsls_fopen` ensures that the same C runtime library is used for both the open operation and reading and writing within an IMSL C Stat Library function to which the file pointer has been passed as an input argument.

Note: The function `imsls_fopen` should only be used to open a file whose file pointer will be input to an IMSL C Stat Library function. Use `imsls_fclose` to close files opened with `imsls_fopen`.

Note: This function is not prototyped in `imsls.h`. This is to avoid including `stdio.h` within `imsls.h`. An extern declaration should be explicitly used to assure compatibility with linkers.

Example

This example writes a matrix to the file `matrix.txt`. The function `imsls_fopen` is used to open a file. This function returns a file pointer, which is passed to `imsls_output_file`. The matrix is written by `imsls_f_write_matrix`, which uses the file pointer from `imsls_output_file`. The function `imsls_fclose` is then used to close the file.

```
#include <imsls.h>
#include <stdio.h>
extern FILE* imsls_fopen(char* filename, char* mode);
extern void imsls_fclose(FILE* file);

int main()
{
    FILE *file;
    float a[] = {
        1.1, 2.4, 3.6,
        4.3, 5.1, 6.7,
        7.2, 8.9, 9.3
    };

    file = imsls_fopen("matrix.txt", "w");

    imsls_output_file(IMSL_SET_OUTPUT_FILE, file,
        0);
    imsls_f_write_matrix("Matrix written matrix.txt",
        3, 3, a, 0);

    imsls_fclose(file);
}
```

Output

The content below is stored in the `matrix.txt` file.

```
Matrix written to matrix.txt
  1      2      3
1    1.1    2.4    3.6
2    4.3    5.1    6.7
3    7.2    8.9    9.3
```

fclose

Closes a file opened by `imsls_fopen`.

Synopsis

```
#include <imsls.h>

#include <stdio.h>

int imsls_fclose(FILE *file)
```

Required Arguments

`FILE *file` (Input/Output)
A file pointer returned from `imsls_fopen`.

Return Value

The return value is zero if the file is successfully closed. If there is an error, `EOF` is returned. `EOF` is defined in `stdio.h`.

Description

The function `imsls_fclose` is a wrapper around the standard C runtime function `fclose`. It is used to close files opened with `imsls_fopen`.

Note: The function `imsls_fopen` should only be used to open a file whose file pointer will be input to an IMSL C Stat Library function. Use `imsls_fclose` to close files opened with `imsls_fopen`.

Note: This function is not prototyped in `imsls.h`. This is to avoid including `stdio.h` within `imsls.h`. An extern declaration should be explicitly used to assure compatibility with linkers.

Example

See [imsls_fopen](#) for an example of its use.

ascii_read

Reads freely-formatted ASCII files.

Synopsis

```
#include <imsls.h>

int imsls_ascii_read(char *filename, ..., 0)
```

Required Arguments

`char *filename` (Input)
A string containing the pathname and filename of the file containing the data.

NOTE: In addition to `filename`, at least one data type, array length, and array name triplet optional argument is required. A Terminal error will be issued when the triplet is missing. See [Optional Arguments](#) for more information.

Return Value

The return value is the status of the read operation. The return value is set to 0 for success and 1 for failure.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_ascii_read(char *filename,

    IMSLS_SHORT, long *n_vals, short **var,

    IMSLS_SHORT_USER, long *n_vals, short var[ ],

    IMSLS_INT, long *n_vals, int **var,

    IMSLS_INT_USER, long *n_vals, int var[ ],

    IMSLS_LONG, long *n_vals, long **var,

    IMSLS_LONG_USER, long *n_vals, long var[ ],
```

```
IMSLS_FLOAT, long *n_vals, float **var,
IMSLS_FLOAT_USER, long *n_vals, float var[ ],
IMSLS_DOUBLE, long *n_vals, double **var,
IMSLS_DOUBLE_USER, long *n_vals, double var[ ],
IMSLS_CHAR, long *n_vals, char **var,
IMSLS_CHAR_USER, long *n_vals, char var[ ][ ],
IMSLS_DATETIME, long *n_vals, struct tm**var,
IMSLS_DATETIME_USER, long *n_vals, struct tmvar[ ],
IMSLS_TIME_VAL, long *n_vals, lmsls_time_val **var,
IMSLS_TIME_VAL_USER, long *n_vals, lmsls_time_val var[ ],
IMSLS_COLUMNS, or
IMSLS_ROWS,
IMSLS_DELIM, int n_delimiters, char delimiters[ ],
IMSLS_DATETIME_FORMAT, int n_formats, int datetime_format[ ],
IMSLS_FILTERS, int n_filters, char filters[ ],
IMSLS_COLUMN_INDEX, int n_columns, int columns[ ],
IMSLS_IGNORE, int n_ignore, char *ignore[ ],
IMSLS_IGNORE_BAD_DATETIME,
IMSLS_REPLACEMENT_NUMBERS, int n_repl_nums, char *miss_vals[ ],
    float repl_nums[ ],
IMSLS_REPLACEMENT_STRINGS, int n_repl_strs,
    char *miss_vals[ ], char *repl_strs[ ],
IMSLS_NRECS, int n_recs,
IMSLS_NSkip, int n_skip,
IMSLS_VALS_PER_REC, int vals_per_rec,
IMSLS_NO_BINARY_CHECK,
IMSLS_ALLOW_CHARS, int n_allow, char allow_chars[ ],
```

0)

Optional Arguments

`IMSL_S_SHORT`, *long *n_vals*, *short **var* (Output)

`IMSL_S_INT`, *long *n_vals*, *int **var* (Output)

`IMSL_S_LONG`, *long *n_vals*, *long **var* (Output)

`IMSL_S_FLOAT`, *long *n_vals*, *float **var* (Output)

`IMSL_S_DOUBLE`, *long *n_vals*, *double **var* (Output)

`IMSL_S_CHAR`, *long *n_vals*, *char **var* (Output)

`IMSL_S_DATETIME`, *long *n_vals*, *struct tm **var* (Output)

`IMSL_S_TIME_VAL`, *long *n_vals*, *lmsls_time_val **var* (Output)

The above optional argument triplets are used to read data from an ASCII file. Each optional argument is a comma-separated list of data type, array length, and array name describing the output variable to read from `filename`. At least one triplet is required. In each of the above cases, `n_vals` is output only and is the number of elements read.

NOTE: As many as 1024 variables can be read from an input data file. Optional arguments may be repeated to read the same type of data from multiple columns.

`IMSL_S_SHORT_USER`, *long *n_vals*, *short var[]* (Output)

`IMSL_S_INT_USER`, *long *n_val*, *int var[]* (Output)

`IMSL_S_LONG_USER`, *long *n_vals*, *long var[]* (Output)

`IMSL_S_FLOAT_USER`, *long *n_vals*, *float var[]* (Output)

`IMSL_S_DOUBLE_USER`, *long *n_vals*, *double var[]* (Output)

`IMSL_S_CHAR_USER`, *long *n_vals*, *char var[][]* (Output)

`IMSL_S_DATETIME_USER`, *long *n_vals*, *struct tm var[]* (Output)

`IMSL_S_TIME_VAL_USER`, *long *n_vals*, *lmsls_time_val var[]* (Output)

The above optional argument triplets allow users to provide storage for variables to be read from an ASCII file. In each of the above cases, `n_vals` is the length of `var` on input and is the number of elements read on output.

`IMSLS_COLUMNS`, (Input)

or

`IMSLS_ROWS`, (Input)

Specifies the file format organization. `IMSLS_COLUMNS` indicates that the data is column-oriented (processing of the file is done reading down columns in the file). `IMSLS_ROWS` indicates that the data is row-oriented (processing of the file is done reading left to right).

Default: It is assumed that the data is column-oriented.

`IMSLS_DELIM`, `int n_delimiters`, `char delimiters[]` (Input)

An array of length `n_delimiters` containing single-character field delimiters used to delineate columns or rows in the data file. If character fields contain delimiter characters, the string will be interpreted as more than one string, and the data in the file will not match the variable list. Use the decimal value 9 or `"\t"` to identify the TAB character.

Default: A comma – or space– delimited file is assumed.

`IMSLS_DATETIME_FORMAT`, `int n_formats`, `int datetime_format[]` (Input)

An array of length `n_formats` containing the date and time formats to be used for interpreting date or time data within the file. A single value is used for each `datetime_format` variable. Based on the data within the file, positive `datetime_format` numbers refer to date formats; negative `datetime_format` numbers refer to time formats. For a table of valid date and time formats see the discussion in the [Description](#) and [Example 5](#).

Default: It is assumed that the data is in neither date nor time format.

`IMSLS_FILTERS`, `int n_filters`, `char filters[]` (Input)

An array of length `n_filters` containing the one-character strings that `imsls_ascii_read` should check and filter out as it reads the data. Any character found on the keyboard can be used. For example, to filter characters `","` and `";"` specify

```
int n_filters = 2;
```

```
char filters[] = {',', ';'};
```

A special character not found on the keyboard is specified by ASCII code.

Default: Nothing is filtered out.

`IMSLS_COLUMN_INDEX`, `int n_columns`, `int columns[]` (Input)

An array of length `n_columns` containing integers indicating column numbers to read from the file. `IMSLS_COLUMN_INDEX` defines an array of column numbers (starting with the first column as column 1) that corresponds with data type variable name pairs supplied in the optional argument. For example, if an array, `columns=[5, 1, 3]`, is supplied with the list of variables `a1`, `a2`, `a3`, then the values in the `IMSLS_COLUMN_INDEX` array are automatically sorted before reading the columns, resulting in `a1 = column #1`, `a2 = column #3`, `a3 = column #5`.

Default: All columns are read.

`IMSL5_IGNORE`, `int n_ignore`, `char *ignore[]` (Input)

An array of length `n_ignore` containing strings to be ignored. If any of these strings are encountered, `ims5s_ascii_read` skips the entire line and starts reading data from the next line. Any character is allowed. For example, to skip lines containing either of the character strings "abc" or "def" specify

```
int n_ignore = 2;
```

```
char *ignore[] = {"abc", "def"};
```

Adding the special string `$BAD_DATE_TIME` in the `ignore` array has the effect of skipping lines where invalid date/time data is found.

Default: Nothing is ignored.

`IMSL5_IGNORE_BAD_DATETIME`, (Input)

Skip lines where invalid date/time data is found.

Default: Do not ignore invalid date/time data.

`IMSL5_REPLACEMENT_NUMBERS`, `int n_repl_nums`, `char *miss_vals[]`, `float repl_nums[]` (Input)

`n_repl_nums` is the length of arrays `miss_vals` and `repl_nums`. `miss_vals` contains strings representing missing data in the file. Each value in the `repl_nums` array corresponds to a string in the `miss_vals` array. As the input data file is read, occurrences of values that match those in `miss_vals` are replaced by the corresponding element of `repl_nums` for numerical output. If the output variable is of type `IMSL5_CHAR`, `repl_nums` is ignored. See [Example 3](#).

Default: Nothing is replaced.

`IMSL5_REPLACEMENT_STRINGS`, `int n_repl_strs`, `char *miss_vals[]`, `char *repl_strs[]` (Input)

`n_repl_strs` is the length of arrays `miss_vals` and `repl_strs`. `miss_vals` contains strings representing missing data in the file. Each of the strings in `repl_strs` corresponds to a string in the `miss_vals` array. As the input data file is read, occurrences of strings that match those in `miss_vals` are replaced by the corresponding string in `repl_strs` for `IMSL5_CHAR` output. If the output variable is not of type `IMSL5_CHAR`, `repl_strs` is ignored. See [Example 8](#).

Default: Nothing is replaced.

NOTE: `IMSL5_REPLACEMENT_NUMBERS` and `IMSL5_REPLACEMENT_STRINGS` may be used simultaneously when reading data written into output variables of mixed types such as `IMSL5_CHAR` and `IMSL5_INT`.

`IMSL_S_NRECS`, int `n_recs` (Input)

Number of records to read for row or column oriented data. If `n_recs` is not provided or is set to zero (0), the entire file is read. For more information about records, see the [Physical Records versus Logical Records](#) topic in Description.

Default: `n_recs = 0`.

`IMSL_S_NSKIP`, int `n_skip` (Input)

Number of physical records in the file to skip before data is read. If `n_skip` is not provided or is set to zero (0), no records are skipped.

Default: `n_skip = 0`.

`IMSL_S_VALS_PER_REC`, int `vals_per_rec` (Input)

An integer that specifies how many values comprise a single record in the input data file. If not provided, each line of data in the file is treated as a new record. See [Example 4](#).

Default: `vals_per_rec = 0`.

`IMSL_S_NO_BINARY_CHECK` (Input)

ASCII decimal or hex values 0 to 127 are allowed. If binary characters above the 127 decimal value are found, the file is considered binary and reading is terminated. If `IMSL_S_NO_BINARY_CHECK` is specified, the check for binary characters is omitted.

Default: A check for binary characters is performed.

`IMSL_S_ALLOW_CHARS`, int `n_allow`, char `allow_chars[]` (Input)

`allow_chars` is an array of length `n_allow`, containing the octal representation of ASCII values outside of the 0 to 127 decimal range preceded with a backwards slash. Characters specified in `allow_chars` are treated as ASCII characters instead of binary characters. For example, to allow `imsls_ascii_read` to accept characters 130 and 150 (octal values 202 and 226), set the `allow_chars` array as follows:

```
char allow_chars[2] = {"\202\226"};
```

Default: Binary characters are not allowed.

Description

Function `imsls_ascii_read` is adept at reading both row-oriented and column-oriented data files. The steps that `imsls_ascii_read` performs when reading a file include:

1. Open the file.
2. Assign the file a file descriptor.
3. Compose a format string that describes the organization of the data.

4. Close the file when reading of the data has been completed.

`imsls_ascii_read` needs to know which delimiters to expect in the file; comma and space are the default delimiters. `imsls_ascii_read` easily reads data values separated by any combination of commas and spaces or any other delimiters one explicitly defines using `IMSLD_DELIM`.

If neither `IMSLD_ROWS` nor `IMSLD_COLUMNS` is provided, the file is assumed to be organized by columns.

Physical Records versus Logical Records

In an ASCII text file, the end-of-line is signified by the presence of either a CTRL-J (linefeed) or a CTRL-M (return) character, and a record extends from one end-of-line character to the next. However, there are actually two kinds of records; physical records and logical records.

For column-oriented files, the amount of data in a physical record is often sufficient to provide exactly one value for each element in the data array, and then it is a logical record, as well. For row-oriented files, the concept of logical records is not relevant, since data is read as contiguous values separated by delimiters, and the end-of-line is interpreted as another delimiter.

NOTE: `IMSLD_NRECS` counts by logical records, if `IMSLD_VALS_PER_REC` has been defined. `IMSLD_NSkip`, on the other hand, counts by physical records, regardless of any logical record size that has been defined.

Changing the Logical Record Size

`IMSLD_VALS_PER_REC` can be used to explicitly define a different logical record size; In most cases, this optional argument does not need to be provided. For an example of how to use `IMSLD_VALS_PER_REC`, see [Example 4](#).

NOTE: By default, `imsls_ascii_read` considers the physical record to be one line in the file, and the concept of a logical record is not needed. When using logical records, the physical records in the file must all contain the same number of values. `IMSLD_VALS_PER_REC` can be specified only with column-oriented data files.

Filtering and Substitution While Reading Data

`IMSLD_FILTERS` can be used to filter certain characters from the data as it is read. Each character (or sequence of digits that represents the ASCII code for a character) must be enclosed with single quotes. For example, either of the following is a valid specification:

`';` or `'44'`

Furthermore, the two specifications shown above are equivalent to one another. For an example of using filters, see [Example 4](#).

NOTE: Do not filter characters that are used as delimiters. The delimiters enable `imsls_ascii_read` to discern where one data value ends and another one begins.

Characters that match one of the values in filters are treated as if they are not present; in other words, these characters are not treated as data and do not contribute to the size of the logical record, if one has been defined using `IMSL_S_VALS_PER_REC`.

NOTE: `IMSL_S_IGNORE` can be used to supply multi-character strings instead of individual characters. However, a character that matches filters is simply discarded, and filtering resumes from that point, while a string that matches `ignore` causes that entire line to be skipped.

When reading a data file that contains a value such as `#$*10.00**`, and it is preferred that the entire line be skipped, filter the characters individually with `filters = ['#','$','*']` instead of collectively with `ignore = ['#$*','**']`.

Delimiters in the Input File

Values in the file can be separated by commas, spaces, and any other delimiter characters specified with `IMSL_S_DELIM`. Characters not specified with `IMSL_S_DELIM` are treated as data and type conversion is attempted. If type conversion is not possible, `imsls_ascii_read` results in a terminal error.

NOTE: Date and time data, such as months, days, hours, and minutes, may only contain the separation characters slash (/), colon (:), hyphen (-), and comma (,). These four characters may not be used as delimiters for data containing dates and times.

Reading Row-Oriented Files

When reading row-oriented data (`IMSL_S_ROWS`) provide only a single data triplet. The file is read using the delimiter (or the default delimiters if `IMSL_S_DELIM` is omitted) to return an array of length `n`, where `n` is the number of values in the file. Providing `IMSL_S_NRECS` allows the control of how many rows (lines) in the file are transferred into the returned array.

Using `IMSL_S_ROWS` and a single data triplet is an easy way to read the entire content of the file and the result is a single array of data that contains all valid data found within the file.

Reading Column-Oriented Files

When `IMSLS_COLUMNS` is used, `imsls_ascii_read` views the data file as a series of columns with a one-to-one correspondence between columns in the file and variables. For example, for a file containing three columns of data values and with three output arrays defined as `var1`, `var2`, and `var3`; the values from the first record of the file will be transferred to `var1[0]`, `var2[0]`, and `var3[0]`. The three values from the second record of the file will be transferred to `var1[1]`, `var2[1]`, and `var3[1]`, and so forth, until all of the data in the file has been read. The exception is if `IMSLS_NRECS` is provided in which case transfer of data stops when `imsls_ascii_read` reaches the number of records to be read.

The length of the individual output arrays is specified by `n_vals`.

Date and Time Formats

`IMSLS_DATETIME_FORMAT` is used to describe the format of the input string data by specifying a format to use as the data is read. These formats are:

Table 54 – Date/Time Formats

<code>datetime_format[i]</code>	Date Format	Time Format
-1	NA	HH*MnMn*SS[.SSSS]
-2	NA	HHMnMn
1	MM*DD*YY[YY]	NA
2	DD*MM*YY[YY]	NA
3	ddd*YY[YY]	NA
4	DD*mmm[mmmmmm]*YY[YY]	NA
5	[YY]YY*MM*DD	NA
6	MM*DD*YY[YY]	HH*MnMn*SS[.SSSS]
7	MM*DD*YY[YY]	HHMnMn
8	DD*MM*YY[YY]	HH*MnMn*SS[.SSSS]
9	DD*MM*YY[YY]	HHMnMn
10	ddd*YY[YY]	HH*MnMn*SS[.SSSS]
11	ddd*YY[YY]	HHMnMn
12	DD*mmm[mmmmmm]*YY[YY]	HH*MnMn*SS[.SSSS]
13	DD*mmm[mmmmmm]*YY[YY]	HHMnMn
14	[YY]YY*MM*DD	HH*MnMn*SS[.SSSS]
15	[YY]YY*MM*DD	HHMnMn

The abbreviations used in the format descriptions in [Table 54](#) are:

Abbreviations	Description
MM	The numerical month. The month does not need to occupy two spaces. For example, you can enter a 1 for the month of January.
DD	The numerical day of the month. The day does not need to occupy two spaces. For example, for May 5, the numerical day can be 5.
[YY]YY	The numerical year. For example, 1992 can be entered as 92 or 1992.
Ddd	The numerical day of the year. The day does not need to occupy three spaces. For example, February 1 is 32.
mmm[mmmmmm]	The full name of the month or its three character abbreviation.
*	Represents a delimiter that separates the different fields of <code>datetime_format</code> data. The valid <code>datetime_format</code> delimiters can be a slash (/), a colon (:), a hyphen (-), period (.), or a comma (,).
HH	The numerical hour based on a 24-hour clock. For example, 14 is 2 o'clock in the afternoon. For the -1 format, both spaces do not need to be occupied. However, the -2 format requires that both spaces be occupied. For example, 1:00 in the morning must be entered as 01.
Mn	The number of minutes in the hour. For the -1 format, both spaces do not need to be occupied. However, the -2 format requires that both spaces be occupied. For example, 6 minutes must be entered as 06.
SS[.SSS]	The number of seconds in the minute. A decimal part of a second is optional.

`imsls_ascii_read` supports two time value structures. The standard `struct tm` C type is obtained with optional argument `IMSLD_DATETIME`. A time structure that includes a Julian value is accessible with optional argument `IMSLD_TIME_VAL`. A description of `IMSLD_TIME_VAL` follows.

`imsls_time_val`

`imsls_ascii_read` uses the structure `imsls_time_val`, provided in the standard header `<imsls.h>`, to represent an `imsls_time_val` variable and is declared as follows:

```
struct Imsls_time_val {
    short   year;
    unsigned char   month;
    unsigned char   day;
    unsigned char   hour;
    unsigned char   minute;
    float   second ;
    double  julian_dt;
    unsigned char   recalc;
};
```

Examples

Example 1

The data file shown below is a freely-formatted ASCII file named `monotonic.dat`:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

To read the entire contents of the file into a single array, the simplest approach is:

```
#include <imsls.h>

int main()
{
    int status;
    float *var;
    long n_vals;

    status = imsls_ascii_read("monotonic.dat",
        IMSLS_FLOAT, &n_vals, &var,
        IMSLS_ROWS,
        0);

    imsls_f_write_matrix("var", 1, (int) n_vals, var, 0);
}
```

Output

var					
1	2	3	4	5	6
1	2	3	4	5	6
7	8	9	10	11	12
7	8	9	10	11	12

13	14	15	16	17	18
13	14	15	16	17	18
19	20				
19	20				

Example 2

Using the same data file (`monotonic.dat`), our goal in this example is to read only the second and fourth column of data from the file. This time we use the default column-organized format. The code is:

```
#include <imsls.h>

int main()
{
    int i, status, getcols[2] = {2, 4};
    int *var1, *var2;
    long n_vals1, n_vals2;

    status = imsls_ascii_read("monotonic.dat",
        IMSLS_INT, &n_vals1, &var1,
        IMSLS_INT, &n_vals2, &var2,
        IMSLS_COLUMN_INDEX, 2, getcols,
        0);

    imsls_i_write_matrix("var1", 1, (int) n_vals1, var1, 0);
    imsls_i_write_matrix("var2", 1, (int) n_vals2, var2, 0);
}
```

Output

```
      var1
  1   2   3   4
  2   7  12  17

      var2
  1   2   3   4
  4   9  14  19
```

Example 3

The data file shown below is a freely-formatted ASCII file named `intake.dat`:

```
151-182-BADX-214-515
316-197-BADY-199-206
```

This example replaces “BADX” and “BADY” with float type numbers `-9999.0` and `9999.0`, respectively.

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

int main()
{
```

```

int status, i;
long n_vals, *var;
char *miss_vals[] = {"BADX", "BADY"}, delimiters[] = "-";
float repl_nums[] = {-9999.0, 9999.0};

status = imsls_ascii_read("intake.dat",
    IMSLS_LONG, &n_vals, &var,
    IMSLS_REPLACEMENT_NUMBERS, 2, miss_vals, repl_nums,
    IMSLS_ROWS,
    IMSLS_DELIM, 1, delimiters,
    0);

for (i = 0; i < n_vals; i++)
    printf("var[%d] = %ld\n", i, var[i]);
}

```

Output

```

var[0] = 151
var[1] = 182
var[2] = -9999
var[3] = 214
var[4] = 515
var[5] = 316
var[6] = 197
var[7] = 9999
var[8] = 199
var[9] = 206

```

Example 4

The data file shown below is a freely-formatted ASCII file named `level.dat`. This data file uses the semicolon (;) and the slash (/) as delimiters, and the comma (,) to separate the thousands digit from the hundreds digit. This file has three logical records on every line; at the end of each logical record is a slash:

```

5,992;17,121/8,348;17,562/5,672;19,451/
5,459;18,659/7,088;17,052/8,541;13,437/
6,362;15,894/8,992;17,509/7,785;14,796/

```

Optional argument `IMSLS_FILTERS` is provided to filter the commas out of the data.

```

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

int main()
{
    int status, i;
    long n_vals1, n_vals2, *gap, *bar;
    char delimiters[] = ";/", filter[] = {','};

    status = imsls_ascii_read("level.dat",
        IMSLS_LONG, &n_vals1, &gap,
        IMSLS_LONG, &n_vals2, &bar,
        IMSLS_COLUMNS,

```

```
        IMSLS_DELIM, 2, delimiters,
        IMSLS_FILTERS, 1, filter,
        IMSLS_VALS_PER_REC, 2,
        0);

    for (i = 0; i < n_vals1; i++)
        printf("gap[%d] = %ld\n", i, gap[i]);

    printf("\n");

    for (i = 0; i < n_vals2; i++)
        printf("bar[%d] = %ld\n", i, bar[i]);
}
```

Output

```
gap[0] = 5992
gap[1] = 8348
gap[2] = 5672
gap[3] = 5459
gap[4] = 7088
gap[5] = 8541
gap[6] = 6362
gap[7] = 8992
gap[8] = 7785

bar[0] = 17121
bar[1] = 17562
bar[2] = 19451
bar[3] = 18659
bar[4] = 17052
bar[5] = 13437
bar[6] = 15894
bar[7] = 17509
bar[8] = 14796
```

Example 5

Assume that you have a file, `events.dat`, that contains some data values and also some chronological information about when those data values were recorded:

```
01/01/92 5:45:12 10 01-01-92 3276
02/01/92 10:10:10 15.89 06-15-91 99
05/15/91 2:02:02 14.2 12-25-92 876
```

The date/time formats used to transfer this data have the following definitions:

- Format Number = 6— MM*DD*YY (* = any delimiter) HH*MM*SS (* = any delimiter)
- Format Number = 1— MM*DD*YY (* = any delimiter)

To read the date and time from the first and third column into `imsls_time_val` variable and read the fourth column of floating point data into another variable:

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

```

#include <stdio.h>

int main()
{
    int status, j, datetimeformat[1] = {6};
    long n_vals1, n_vals2;
    float *var;
    char delimiters[] = " ";
    Imsls_time_val *date;

    status = imsls_ascii_read("events.dat",
        IMSLS_TIME_VAL, &n_vals1, &date,
        IMSLS_FLOAT, &n_vals2, &var,
        IMSLS_DATETIME_FORMAT, 1, datetimeformat,
        IMSLS_DELIM, 1, delimiters,
        0);

    imsls_f_write_matrix("var", 1, (int) n_vals2, var, 0);
    printf("\n");

    for (j=0; j<n_vals1; j++){
        printf("var[%d] year: %d\n", j, date[j].year);
        printf("date[%d].month: %d\n", j, date[j].month);
        printf("date[%d].day: %d\n", j, date[j].day);
        printf("date[%d]a.hour: %d\n", j, date[j].hour);
        printf("date[%d].minute: %d\n", j, date[j].minute);
        printf("date[%d].second: %f\n\n", j, date[j].second);
    }
}

```

Output

```

var
      1          2          3
  10.00      15.89      14.20

var[0] year: 1992
date[0].month: 1
date[0].day: 1
date[0]a.hour: 5
date[0].minute: 45
date[0].second: 12.000000

var[1] year: 1992
date[1].month: 2
date[1].day: 1
date[1]a.hour: 10
date[1].minute: 10
date[1].second: 10.000000

var[2] year: 1991
date[2].month: 5
date[2].day: 15
date[2]a.hour: 2
date[2].minute: 2
date[2].second: 2.000000

```

Example 6

To read the first, third, and fourth columns of `events.dat`, define an integer array and a second `imsls_time_val` variable, and change the call to `imsls_ascii_read` as shown below.

Notice there are two different date/time formats for column 1 and column 3 in the `events.dat` file. For column 1, format 6 is used to read the date data into `date1`. For column 3, format 1 is used to read date data into `date2`. In the case where the date/time format is the same for all columns, provide only one date/time format to be reused by all columns.

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

int main()
{
    int status, j, *calib, datetimetypeformat[2] = {6, 1},
        getcolumns[4] = {1, 3, 4};
    long n_vals1, n_vals2, n_vals3;
    char delimiters[] = " ";
    char* ignore[] = {"$BAD_DATE_TIME"};
    Imsls_time_val *date1, *date2;

    status = imsls_ascii_read("events.dat",
        IMSLS_TIME_VAL, &n_vals1, &date1,
        IMSLS_TIME_VAL, &n_vals2, &date2,
        IMSLS_INT, &n_vals3, &calib,
        IMSLS_DELIM, 1, delimiters,
        IMSLS_COLUMN_INDEX, 3, getcolumns,
        IMSLS_DATETIME_FORMAT, 2, datetimetypeformat,
        0);

    for (j=0; j<n_vals1; j++){
        printf("date1[%d] year: %d\n", j, date1[j].year);
        printf("date1[%d].month: %d\n", j, date1[j].month);
        printf("date1[%d].day: %d\n", j, date1[j].day);
        printf("date1[%d].hour: %d\n", j, date1[j].hour);
        printf("date1[%d].minute: %d\n", j, date1[j].minute);
        printf("date1[%d].second: %f\n\n", j, date1[j].second);
    }

    for (j=0; j<n_vals2; j++){
        printf("date2[%d] year: %d\n", j, date2[j].year);
        printf("date2[%d].month: %d\n", j, date2[j].month);
        printf("date2[%d].day: %d\n", j, date2[j].day);
        printf("date2[%d].hour: %d\n", j, date2[j].hour);
        printf("date2[%d].minute: %d\n", j, date2[j].minute);
        printf("date2[%d].second: %f\n\n", j, date2[j].second);
    }

    imsls_i_write_matrix("calib", 1, (int) n_vals3, calib, 0);
}
```

Output

```
date1[0] year: 1992
```

```
date1[0].month: 1
date1[0].day: 1
date1[0].hour: 5
date1[0].minute: 45
date1[0].second: 12.000000

date1[1] year: 1992
date1[1].month: 2
date1[1].day: 1
date1[1].hour: 10
date1[1].minute: 10
date1[1].second: 10.000000

date1[2] year: 1991
date1[2].month: 5
date1[2].day: 15
date1[2].hour: 2
date1[2].minute: 2
date1[2].second: 2.000000

date2[0] year: 1992
date2[0].month: 1
date2[0].day: 1
date2[0].hour: 0
date2[0].minute: 0
date2[0].second: 0.000000

date2[1] year: 1991
date2[1].month: 6
date2[1].day: 15
date2[1].hour: 0
date2[1].minute: 0
date2[1].second: 0.000000

date2[2] year: 1992
date2[2].month: 12
date2[2].day: 25
date2[2].hour: 0
date2[2].minute: 0
date2[2].second: 0.000000

      calib
      1      2      3
3276      99      876
```

Example 7

The following data file is a freely-formatted ASCII file named `num.dat` in which a value is missing:

```
0,1,,3,4
5,6,7,8,9
```

The missing value is replaced with 99.0.

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

```

int main()
{
    int status, *var, getcol[1] = {3};
    long n_vals;
    char *miss_str[] = {" "};
    float repl_nums[] = {99.0};

    status = imsls_ascii_read("num.dat",
        IMSLS_INT, &n_vals, &var,
        IMSLS_COLUMN_INDEX, 1, getcol,
        IMSLS_REPLACEMENT_NUMBERS, 1, miss_str, repl_nums,
        0);

    imsls_i_write_matrix("var", 1, (int) n_vals, var, 0);
}

```

Output

```

var
 1  2
99  7

```

Example 8

The data file shown below is a freely-formatted ASCII file named `char.dat`:

```

a,b,c,d
e,BAD,g,h

```

The string GOOD is substituted for the missing value represented as BAD.

```

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

int main()
{
    int status, i, getcol[1] = {2};
    long n_vals;
    char *miss_val[] = {"BAD"}, *rep_str[] = {"GOOD"}, **var;

    status = imsls_ascii_read("char.dat",
        IMSLS_CHAR, &n_vals, &var,
        IMSLS_COLUMN_INDEX, 1, getcol,
        IMSLS_REPLACEMENT_STRINGS, 1, miss_val, rep_str,
        0);

    for(i=0; i<n_vals; i++)
        printf("var[%d] = %s\n", i, var[i]);
}

```

Output

```

var[0] = b
var[1] = GOOD

```

Example 9

The data file shown below is a freely-formatted ASCII file named `chemicals.dat`:

```
Elemental Carbon
Sulfate
Benzo[e]pyrene
Indeno[1,2,3-cd]pyrene
n-Heptadecanoic acid
```

Note that the first and fourth lines contain a space and commas respectively. Since the default delimiters are comma and space, in order to read the first and fourth lines of the data as single entities, a delimiter which does not occur as a character in the data file must be specified using `IMSLS_DELIM`. For this example, we know that there are no TAB characters, so TAB is specified as the delimiter (ASCII byte value 9 specified as a hex value). However, another single character could be specified, such as '\$' or 'Q'.

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

int main()
{
    int status,i;
    long n_vals;
    char **var, delimiters[] = {"\x9"};

    status = imsls_ascii_read("chemicals.dat",
        IMSLS_CHAR, &n_vals, &var,
        IMSLS_ROWS,
        IMSLS_DELIM, 1, delimiters,
        0);

    for (i=0; i<n_vals; i++)
        printf("var[%d] = %s\n", i, var[i]);
}
```

Output

```
var[0] = Elemental Carbon
var[1] = Sulfate
var[2] = Benzo[e]pyrene
var[3] = Indeno[1,2,3-cd]pyrene
var[4] = n-Heptadecanoic acid
```

Warning Errors

`IMSLS_M_INVALID_BINARY_CHAR`

A binary character may have been detected.

omp_options

Sets various OpenMP options.

Synopsis

```
#include <imsls.h>

void imsls_omp_options(..., 0)
```

Return Value

The return value for this function is void.

Synopsis with Optional Arguments

```
#include <imsls.h>

void imsls_omp_options
  (IMSL_SET_FUNCTIONS_THREAD_SAFE, int setting, (Input)
  IMSL_GET_FUNCTIONS_THREAD_SAFE, int *psetting, (Output)
  0)
```

Optional Arguments

IMSL_SET_FUNCTIONS_THREAD_SAFE, int setting (Input)

If nonzero, user supplied functions are assumed to be thread-safe. This allows user functions to be evaluated in parallel with different arguments.

Default: User supplied functions are not assumed to be thread-safe and will not be evaluated in parallel by IMSL C Stat Library functions.

IMSL_GET_FUNCTIONS_THREAD_SAFE, int *psetting (Output)

Sets the integer pointed to by **psetting** to zero if user functions are not assumed to be thread-safe and to one if they are assumed to be thread-safe.

Description

The performance of some IMSL C Stat Library functions can be improved if they evaluate user supplied functions in parallel. Unfortunately, incorrect results can occur if the user supplied functions are not thread-safe. By default, the IMSL C Stat Library assumes user supplied functions are not thread-safe and thus will not evaluate them in parallel. To change this assumption, use the optional argument `IMSL5_SET_FUNCTIONS_THREAD_SAFE` with its argument equal to one.

This function can be used multiple times in an application to change the thread-safe assumption.

Example

This example performs a chi-squared test on a randomly generated sample. A call to the function `imsls_omp_options` is used to indicate that function `cdf` is thread-safe and so can be safely evaluated by multiple, simultaneous threads.

```
#include <imsls.h>
static float cdf(float x);

#define SEED 123457
#define N_CATEGORIES 10
#define N_OBSERVATIONS 1000

int main()
{
    float *x, p_value;

    imsls_omp_options(IMSL5_SET_FUNCTIONS_THREAD_SAFE, 1, 0);

    imsls_random_seed_set(SEED);
    x = imsls_f_random_normal (N_OBSERVATIONS, 0);
    p_value = imsls_f_chi_squared_test (cdf, N_OBSERVATIONS,
        N_CATEGORIES, x, 0);
    printf ("p-value = %7.4f\n", p_value);
}

static float cdf(float x)
{
    return imsls_f_normal_cdf(x);
}
```

Output

```
p-value = 0.1546
```

machine (integer)

Returns integer information describing the computer's arithmetic.

Synopsis

```
#include <imsls.h>

long 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 $0 \leq x_k < A$ for $k = 0, \dots, M$. Then,

n	Definition
0	<i>C</i> , bits per character
1	<i>A</i> , the base
2	<i>M_s</i> , the number of base- <i>A</i> digits in a <i>short int</i>

n	Definition
3	$A^{M_s} - 1$, the largest <i>short int</i>
4	M_1 the number of base- A digits in a <i>long int</i>
5	$A^{M_l} - 1$, the largest <i>long int</i>

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 $0 \leq x_k < B$ for $k = 1, \dots, N$ and $E_{\min} \leq E \leq E_{\max}$. Then

n	Definition
6	B , the base
7	N_f , the number of base- B digits in <i>float</i>
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 <i>double</i>
11	E_{\max_f} , the largest long <i>int</i>
12	E_{\max_d} , the number of base- B digits in <i>double</i>

Example

In this example, all the values returned by `imsls_i_machine` on a 32-bit machine with IEEE (Institute for Electrical and Electronics Engineer) arithmetic are printed.

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

int main() {
    int n;
    long ans;

    for (n = 0; n <= 12; n++) {
        ans = imsls_i_machine(n);
        printf("imsls_i_machine(%d) = %ld\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; $0 \leq x_k < B$ for $k = 1, 2, \dots, N_f$; and

$$E_{\min_f} \leq E \leq E_{\max_f}$$

Note that $B = \text{imsls_i_machine}(6)$; $N_f = \text{imsls_i_machine}(7)$;

$$E_{\min_f} = \text{imsls_i_machine}(8)$$

and

$$E_{\max_f} = \text{imsls_i_machine}(9)$$

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* $B = \text{imsls_i_machine}(6)$, $N_d = \text{imsls_i_machine}(10)$,

$$E_{\min_d} = \text{imsls_i_machine}(11)$$

and

$$E_{\max_d} = \text{imsls_i_machine}(12)$$

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>
#include <stdio.h>

int 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);
    }
    return 0;
}
```

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 fourteen data sets. In this case, any optional arguments are ignored.

	<code>n_observations</code>	<code>n_variables</code>	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
10	958	10	Tic-Tac-Toe Endgame
11	4601	58	Spambase Data Set
12	690	16	Credit Approval
13	20000	17	Letter Recognition Data
14	366	35	Dermatology Database

Return Value

If `data_set_choice` \neq 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.

`IMSL_S_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 table gives the references for each data set:

<code>data_set_choice</code>	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)
10	Aha, D. W. (1991, pp. 117-121), and Asuncion, A. & Newman, D.J. (2007)
11	Asuncion, A. & Newman, D.J. (2007)
12	Quinlan (1987, pp. 221-234, 1997), and Asuncion, A. & Newman, D.J. (2007)
13	P. W. Frey and D. J. Slate, (Machine Learning Vol 6 #2 March 91), and Asuncion, A. & Newman, D.J. (2007)
14	G. Demiroz, H. A. Govenir, and N. Ilter, (Artificial Intelligence in Medicine), and Asuncion, A. & Newman, D.J. (2007)

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>

int 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

- IMSLA_MATRIX**, *int nrowa, int ncola, float a[]* (Input)
 The $nrowa \times ncola$ matrix *A*.
- IMSLA_COL_DIM**, *int a_col_dim* (Input)
 Column dimension of *A*.
 Default: `a_col_dim = ncola`
- IMSLB_MATRIX**, *int nrowb, int ncolb, float b[]* (Input)
 The $nrowb \times ncolb$ matrix *B*.
- IMSLB_COL_DIM**, *int b_col_dim* (Input)
 Column dimension of *B*.
 Default: `b_col_dim = ncolb`
- IMSLX_VECTOR**, *int nx, float *x* (Input)
 Vector *x* of size `nx`.
- IMSLY_VECTOR**, *int ny, float *y* (Input)
 Vector *y* of size `ny`.
- IMSL_RETURN_USER**, *float ans[]* (Output)
 User-allocated array containing the result.
- IMSL_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 , $x^T A^T$, AB , $B^T A^T$, $x^T y$, xy^T and $x^T A y$ are computed and printed.

```
#include <imsls.h>

int main()
{
    float    A[] = {1, 2, 9,
                   5, 4, 7};
    float    B[] = {3, 2,
                   7, 4,
                   9, 1};
    float    x[] = {7, 2, 1};
    float    y[] = {3, 4, 2};
    float    *ans;

    ans = 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,
    IMSLS_X_VECTOR, 2, x, /* use only the first 2 components of 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>

int 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` containing the permutation `permutation[0], ..., permutation[n-1]` of the integers `0, ..., n`, where `n = n_rows` if the rows of `a` are to be permuted and `n = n_columns` if the columns of `a` are to be permuted.

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[ ],
                               lmsls_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 $B = \text{result}$ and $p_i = \text{permutation}[i]$, then $B_{ij} = A_{p_{ij}}$ for all i, j .

Example

This example permutes the columns of a matrix `a`.

```
#include <imsls.h>

int 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

impute_missing

Locate and optionally replace dependent variable missing values with nearest neighbor estimates.

Synopsis

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

```
int imsls_f_impute_missing(int n_observations, int n_variables, int n_independent,  
    int indind[], float x[], ..., 0)
```

The type *double* function is `imsls_d_impute_missing`.

Required Arguments

int `n_observations` (Input)

Number of observations.

int `n_variables` (Input)

Number of variables.

int `n_independent` (Input)

Number of independent variables.

int `indind[]` (Input)

Array of size `n_independent` designating the indices of the columns of `x` containing the independent variables.

float `x[]` (Input)

Array of size `n_observations` × `n_variables` containing the observations. Missing values of the dependent variables may be imputed as functions of the independent variables, but if any of the independent variables have missing values, then imputation will not be performed and a warning will be issued. If one of the optional arguments, `IMSLS_REPLACEMENT_VALUE`, `IMSLS_IMPUTE_METHOD`, or `IMSLS_PURGE` is supplied, `x_imputed` (see optional argument `IMSLS_X_IMPUTED`) contains the imputed data on output.

Return Value

The number of missing values (`n_miss`) in the data array `x`.

Synopsis with Optional Arguments

```
#include <imsls.h>

int imsls_f_impute_missing(int n_observations, int n_variables, int n_independent,
    int indind[ ], float x[ ],

    IMSLS_MISSING_VALUE, float mval,

    IMSLS_METRIC_DIAG, float g[ ],

    IMSLS_REPLACEMENT_VALUE, float replacement_value, or

    IMSLS_IMPUTE_METHOD, int method, int k, or

    IMSLS_PURGE, int *n_missing_rows, int **missing_row_indices,

    IMSLS_MISSING_INDEX, int **indices,

    IMSLS_X_IMPUTED, float **x_imputed,

    IMSLS_X_IMPUTED_USER, float x_imputed[ ],

    0)
```

Optional Arguments

IMSLS_MISSING_VALUE, *float mval*, (Input)

Scalar value (other than NaN) representing a missing value. NaN always represents a missing value, so if *mval* is not NaN it will be treated as a second type of missing value.

IMSLS_METRIC_DIAG, *float g[]* (Input)

Array of length **n_independent** defining a diagonal metric for independent variable space. This scales the independent variables in the distance calculations used to determine nearest neighbors. The default measure of distance is Euclidean ($g[i] = 1$ for all i).

IMSLS_REPLACEMENT_VALUE, *float replacement_value* (Input)

Replace missing values in **x** with **replacement_value**. Output data array is returned in **x_imputed**. Requires optional argument **IMSLS_X_IMPUTED** or **IMSLS_X_IMPUTED_USER**.

or

IMSLS_IMPUTE_METHOD, *int method*, *int k* (Input)

The method to be used for imputing missing values using **k** nearest neighbors. Replace missing value of dependent variable **y** at point **x** in the space of independent variables with the mode, mean, median, geometric mean, or linear regression (**method**) of **y** on those **k** nearest neighbors of **x**

which have no missing values. To use all of the data and eliminate the need to compute neighborhoods, set $k \geq n_{\text{observations}}$. If there are no independent variables, set $k \geq n_{\text{observations}}$. Imputed data is returned in `x_imputed`. Requires optional argument `IMSLS_X_IMPUTED` or `IMSLS_X_IMPUTED_USER`.

Valid values for `method` are:

method	Description
<code>IMSLS_MODE_METH</code>	Mode
<code>IMSLS_MEAN_METH</code>	Mean
<code>IMSLS_MEDIAN_METH</code>	Median
<code>IMSLS_GEOMEAN_METH</code>	Geometric mean
<code>IMSLS_LINEAR_METH</code>	Linear regression

or

`IMSLS_PURGE`, *int* *`n_missing_rows`, *int* **`missing_row_indices` (Output)

All rows with missing values are removed from `x` and the resulting data array is returned in `x_imputed`. `n_missing_rows` is the number of rows that were removed.

`missing_row_indices` are the indices of the rows that were removed. Requires optional argument `IMSLS_X_IMPUTED` or `IMSLS_X_IMPUTED_USER`.

`IMSLS_MISSING_INDEX`, *int* **`indices` (Output)

Address of a pointer to the internally allocated array of size `n_miss`, containing the indices of `x` where missing values occur. `n_miss` is the function return value. If the data has no missing values, the pointer is returned as `NULL`.

`IMSLS_X_IMPUTED`, *float* ** `x_imputed` (Output)

Array containing imputed data. This argument is required when `IMSLS_REPLACEMENT_VALUE`, `IMSLS_IMPUTE_METHOD`, or `IMSLS_PURGE` is supplied. For options `IMSLS_REPLACEMENT_VALUE` and `IMSLS_IMPUTE_METHOD`, `x_imputed` contains all data from `x` with missing values replaced in the dependent variable columns. For option `IMSLS_PURGE`, `x_imputed` is an array of size $n_{\text{observations}} - n_{\text{missing_rows}} \times n_{\text{variables}}$, containing the data from `x` with the rows of missing data removed.

`IMSLS_X_IMPUTED_USER`, *float* `x_imputed[]` (Output)

Storage for array `x_imputed` is provided by the user. See `IMSLS_X_IMPUTED`. The size of this array must be the same as `x`, $n_{\text{observations}} \times n_{\text{variables}}$. For the `IMSLS_PURGE` option, use only the first $n_{\text{observations}} - n_{\text{missing_rows}} \times n_{\text{variables}}$ values on output.

Description

Function `imsls_f_impute_missing` locates missing values, and optionally, replaces them with estimated values. This replacement process, called imputation, applies only to dependent variables. If x denotes an arbitrary point in independent variable space and y denotes a dependent variable with a missing value at $x = x_i$, then y at x_i is estimated as $y(x_i) = f(x_i)$ where $f(x)$ is some function of x in some neighborhood of x_i .

`imsls_f_impute_missing` provides five options (see [IMSLS_IMPUTE_METHOD](#)) for the form of $f(x)$, and each option allows neighborhood size to be specified in terms of some given number of nearest neighbors. The neighbors exclude observations with missing values and are determined by distance, the norm relative to metric G ,

$$\|x\| = \sqrt{x^T G x}.$$

By default, $G = I$, but the `IMSLS_METRIC_DIAG` option can be used to specify any other diagonal metric G . A sixth option, `IMSLS_REPLACEMENT_VALUE`, allows the user to specify one value to be used as a replacement for all missing values.

Instead of being used for imputation, `imsls_f_impute_missing` can be used to simply remove all observations which contain missing values. This is accomplished with the option `IMSLS_PURGE`. With this option, all rows with missing values are removed from the input data matrix. Unlike imputation, this option is not limited to dependent variables and can be used to handle missing values in the independent variables.

Usually either imputation or deletion will be performed, but `imsls_f_impute_missing` can be used for the more basic task of returning the indices of missing values. The indices could then be used to implement other imputation methods.

Following the standard practice, missing data values are always represented by NaN. Option `IMSLS_MISSING_VALUE` allows the user to also specify a second value to represent missing values.

Examples

Example 1

Count the missing values in a data set, where the only valid missing value is NaN.

```
#include <imsls.h>
#include <stdio.h>
#define N_OBSERVATIONS 20
#define N_VARIABLES 4

int main()
{
    float x[N_OBSERVATIONS][N_VARIABLES];
    int count, i, j;
```

```

                                /* create the test data */

for(i=0;i<N_OBSERVATIONS;i++) {
    for(j=0;j<N_VARIABLES;j++) {
        x[i][j]= (float)((i*N_VARIABLES)+j);
    }
}

                                /* replace some of the data values */

x[3][1] = imsls_f_machine(6); /* NaN */
x[5][2] = imsls_f_machine(6); /* NaN */
x[7][2] = imsls_f_machine(7); /* positive infinity */
x[9][3] = imsls_f_machine(8); /* negative infinity */

                                /* declare no independent variables */

                                /* note +/-inf are not considered 'missing' */

count = imsls_f_impute_missing (N_OBSERVATIONS, N_VARIABLES, 0,
                                NULL, (float*)x,
                                0);
printf("number of missing values = %d\n", count);
}

```

Output

```
number of missing values = 2
```

Example 2

Set the value 20 to represent a missing value and find the indices in **x** which contain the missing value.

```

#include <imsls.h>
#include <stdio.h>
#define N_OBSERVATIONS 20
#define N_VARIABLES 4

int main()
{
    float x[N_OBSERVATIONS][N_VARIABLES];
    float mval;
    int n_independent, count, i, j;
    int indind[2];
    int *indices;

                                /* declare 2 independent variables */

n_independent = 2;
indind[0] = 2; /* declare that column 2 is independent */
indind[1] = 3; /* declare that column 3 is independent */

                                /* missing value is represented by 20 */
                                /* and will be located at x[5][0] */

mval = 20.0;

```

```

/* create the test data */

for(i=0;i<N_OBSERVATIONS;i++) {
    for(j=0;j<N_VARIABLES;j++) {
        x[i][j] = (float)((i*N_VARIABLES)+j);
    }
}

count = imsls_f_impute_missing (N_OBSERVATIONS, N_VARIABLES,
                                n_independent, indind, (float*)x,
                                IMSLS_MISSING_VALUE, mval,
                                IMSLS_MISSING_INDEX, &indices,
                                0);

printf("number of missing values = %d\n", count);
for (i=0; i<count;i++) {
    printf("indices[%d] = %d\n", i, indices[i]);
}
}

```

Output

```

number of missing values = 1
indices[0] = 20

```

Example 3

In this example both NaN and infinity represent missing values in the original data. In the first call to `imsls_f_impute_missing`, missing values are replaced by negative infinity. In the second call to `imsls_f_impute_missing`, negative infinity is set to represent missing values and the rows containing the missing values are purged for the final output.

```

#include <imsls.h>
#include <stdio.h>
#define N_ROWS 6
#define N_COLS 4

int main()
{

    float *x_imputed, *x_purged;
    float mval, replacement_value;
    float data[N_ROWS][N_COLS];
    int n_independent, count, npurge, i, j;
    int indind[1];
    int *bad_obs;
    char *fmt="%6.2f";

/* create the test data */

for(i=0;i<N_ROWS;i++) {
    for(j=0;j<N_COLS;j++) {
        data[i][j] = (float)((i*N_COLS)+j);
    }
}

```

```
}
                                /* insert bad values into data */
data[1][1] = imsls_f_machine(6); /* NaN */
data[2][2] = imsls_f_machine(7); /* positive infinity */
data[3][3] = imsls_f_machine(8); /* negative infinity */

imsls_f_write_matrix ("Original data with missing values",
                    N_ROWS, N_COLS, (float*)data,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);

                                /* set the missing value to be +inf */
mval = imsls_f_machine(7);
                                /* replace missing values with neg inf */
replacement_value = imsls_f_machine(8);
                                /* declare one independent variable */
n_independent = 1;
indind[0] = 0;

                                /* replace Nan and +inf values with -inf */
count = imsls_f_impute_missing (N_ROWS, N_COLS, n_independent,
                                indind, (float*)data,
                                IMSLS_MISSING_VALUE, mval,
                                IMSLS_REPLACEMENT_VALUE, replacement_value,
                                IMSLS_X_IMPUTED, &x_imputed,
                                0);

imsls_f_write_matrix ("Data with values replaced",
                    N_ROWS, N_COLS, x_imputed,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);

                                /* now purge all rows containing -inf */
mval = imsls_f_machine(8);
count = imsls_f_impute_missing (N_ROWS, N_COLS, n_independent,
                                indind, x_imputed,
                                IMSLS_MISSING_VALUE, mval,
                                IMSLS_PURGE, &npurge, &bad_obs,
                                IMSLS_X_IMPUTED, &x_purged,
                                0);

printf("\n number missing = %d, number of rows purged = %d\n",
       count, npurge);

printf("\n Purged row numbers:\n");
for(i=0;i<npurge;i++) {
    printf(" %d ",bad_obs[i]);
}
printf ("\n");

imsls_f_write_matrix ("New data with bad rows purged",
                    N_ROWS-npurge, N_COLS, x_purged,
                    IMSLS_WRITE_FORMAT, fmt,
                    0);
}
```

Output

```
Original data with missing values
```

	1	2	3	4
1	0.00	1.00	2.00	3.00
2	4.00	6.00	7.00
3	8.00	9.00	++++++	11.00
4	12.00	13.00	14.00	-----
5	16.00	17.00	18.00	19.00
6	20.00	21.00	22.00	23.00

```
Data with values replaced
```

	1	2	3	4
1	0.00	1.00	2.00	3.00
2	4.00	-----	6.00	7.00
3	8.00	9.00	-----	11.00
4	12.00	13.00	14.00	-----
5	16.00	17.00	18.00	19.00
6	20.00	21.00	22.00	23.00

```
number missing = 3, number of rows purged = 3
```

```
Purged row numbers:
```

```
1 2 3
```

```
New data with bad rows purged
```

	1	2	3	4
1	0.00	1.00	2.00	3.00
2	16.00	17.00	18.00	19.00
3	20.00	21.00	22.00	23.00

Example 4

Replace missing values computed using the mean of the 3 nearest neighbors.

```
#include <imsls.h>

#define N_ROWS 10
#define N_COLS 4

int main()
{
    float *x_imputed;
    float data[N_ROWS][N_COLS];
    int n_independent, count, i, j;
    int indind[2];
    char *fmt="%6.2f";

    /* create the test data */
    for(i=0;i<N_ROWS;i++) {
        for(j=0;j<N_COLS;j++) {
            data[i][j] = (float)((i*N_COLS)+j);
        }
    }

    data[1][3] = imsls_f_machine(6); /* insert NaN at row 1 col 3 */
    data[4][2] = imsls_f_machine(6); /* insert NaN at row 4 col 2 */
}
```

```

imsls_f_write_matrix ("Original data with missing values", N_ROWS,
    N_COLS, (float*) data,
    IMSLS_WRITE_FORMAT, fmt,
    0);

/* declare two independent variables */
n_independent = 2;
indind[0] = 0;
indind[1] = 1;

/* replace missing values using mean method */
count = imsls_f_impute_missing (N_ROWS, N_COLS, n_independent,
    indind, (float*)data,
    IMSLS_IMPUTE_METHOD,
    IMSLS_MEAN_METH, 3,
    IMSLS_X_IMPUTED, &x_imputed,
    0);

imsls_f_write_matrix ("Imputed data (using mean method)", N_ROWS,
    N_COLS, x_imputed,
    IMSLS_WRITE_FORMAT, fmt,
    0);
}

```

Output

```

Original data with missing values
   1   2   3   4
1  0.00  1.00  2.00  3.00
2  4.00  5.00  6.00  .....
3  8.00  9.00 10.00 11.00
4 12.00 13.00 14.00 15.00
5 16.00 17.00 ..... 19.00
6 20.00 21.00 22.00 23.00
7 24.00 25.00 26.00 27.00
8 28.00 29.00 30.00 31.00
9 32.00 33.00 34.00 35.00
10 36.00 37.00 38.00 39.00

```

```

Imputed data (using mean method)
   1   2   3   4
1  0.00  1.00  2.00  3.00
2  4.00  5.00  6.00  9.67
3  8.00  9.00 10.00 11.00
4 12.00 13.00 14.00 15.00
5 16.00 17.00 20.67 19.00
6 20.00 21.00 22.00 23.00
7 24.00 25.00 26.00 27.00
8 28.00 29.00 30.00 31.00
9 32.00 33.00 34.00 35.00
10 36.00 37.00 38.00 39.00

```

Warning Errors

IMSLS_NO_GOOD_ROW

Each row contains missing values. No imputation is performed.

IMSLS_INDEP_HAS_MISSING

At least one of the independent variables contains a missing value. No imputation is performed.

binomial_coefficient

Evaluates the binomial coefficient.

Synopsis

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

```
float 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>

int main()
{
    int    n = 9;
    int    m = 5;
    float  ans;

    ans = imsls_f_binomial_coefficient(n, m);
    printf("binomial coefficient = %.1f\n", ans);
}
```

Output

```
binomial coefficient = 126.0
```

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 $\beta(\mathbf{a}, \mathbf{b})$. If no result can be computed, then NaN is returned.

Description

The beta function, $\beta(\mathbf{a}, \mathbf{b})$, 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 $\beta(0.5, 0.2)$.

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

int 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);  
  return 0;  
}
```

Output

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

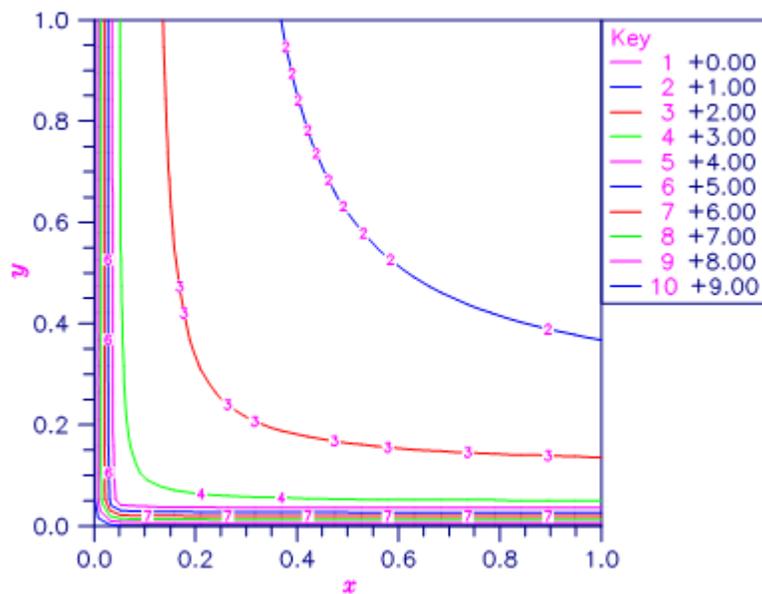


Figure 40, Plot of $\beta(x, b)$

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 regularized incomplete beta function.

Synopsis

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

```
float imsls_f_beta_incomplete (float x, float a, float b)
```

The type *double* function is `imsls_d_beta_incomplete`.

Required Arguments

float **x** (Input)

Argument at which the regularized incomplete beta function is to be evaluated.

float **a** (Input)

First shape parameter.

float **b** (Input)

Second shape parameter.

Return Value

The value of the regularized incomplete beta function.

Description

The regularized incomplete beta function $I_x(a, b)$ is defined

$$I_x(a, b) = B_x(a, b) / B(a, b)$$

where

$$B_x(a, b) = \int_0^x t^{a-1} (1-t)^{b-1} dt$$

is the incomplete beta function,

$$B(a, b) = B_1(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

is the (complete) beta function, and $\Gamma(a)$ is the gamma function.

The regularized incomplete beta function `imsls_f_beta_incomplete(x, a, b)` is identical to the beta probability distribution function `imsls_f_beta_cdf(x, a, b)` which represents the probability that a beta random variable X with shape parameters a and b takes on a value less than or equal to x . The regularized incomplete beta function requires that $0 \leq x \leq 1$, $a > 0$, and $b > 0$ and it underflows for sufficiently small x and large a . This underflow is not reported as an error. Instead, the value zero is returned.

Example

Suppose X is a beta random variable with shape parameters $a = b = 12$ (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>
#include <stdio.h>

int main()
{
    float a = 12, b = 12, x = 0.6, p;

    p = imsls_f_beta_incomplete(x, a, b);
    printf("The probability that X is less than %3.1f is "
           "%6.4f\n", x, p);
    x = 0.5;
    p -= imsls_f_beta_incomplete(x, a, b);
    printf("The probability that X is between %3.1f and "
           " 0.6 is %6.4f\n", x, 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
```

log_beta

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

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 $\beta(x, y)$.

Description

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

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

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

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

Example

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

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

int 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
```

Warning Errors

IMSLX_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 .

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.

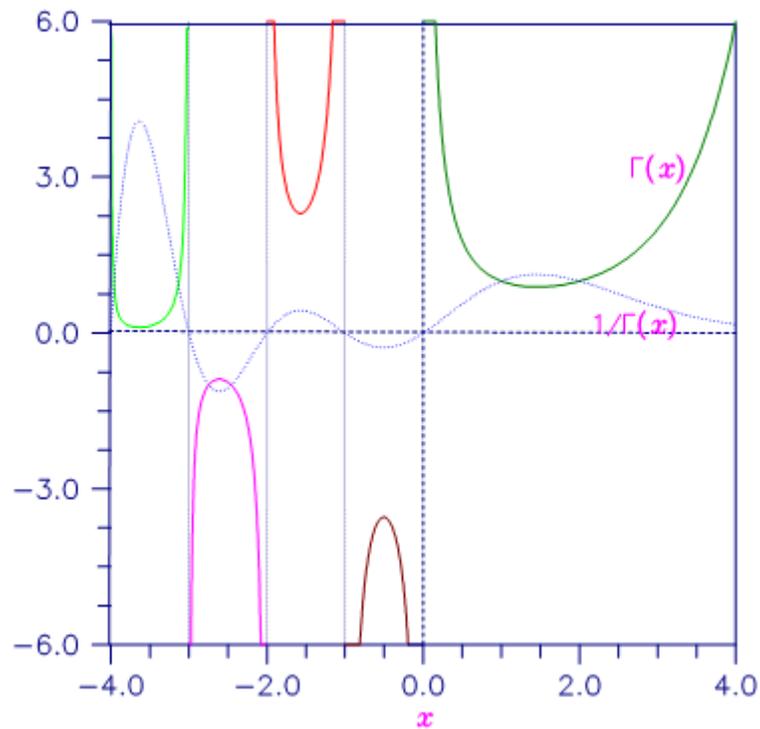


Figure 41, Plot of $\Gamma(x)$ and $1/\Gamma(x)$

Example

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

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

int 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
```

Alert Errors

IMSL5_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

IMSL5_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

IMSL5_ZERO_ARG_OVERFLOW

The argument for the gamma function is too close to zero.

IMSL5_NEAR_NEG_INT_FATAL

The argument for the function is too close to a negative integer.

IMSL5_LARGE_ARG_OVERFLOW

The function overflows because x is too large.

IMSL5_CANNOT_FIND_XMIN

The algorithm used to find x_{\min} failed. This error should never occur.

IMSL5_CANNOT_FIND_XMAX

The algorithm used to find x_{\max} failed. This error should never occur.

gamma_incomplete

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

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 $\Upsilon(a, x)$.

Description

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

$$\Upsilon(a, x) = \int_0^x t^{a-1} e^{-t} dt$$

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

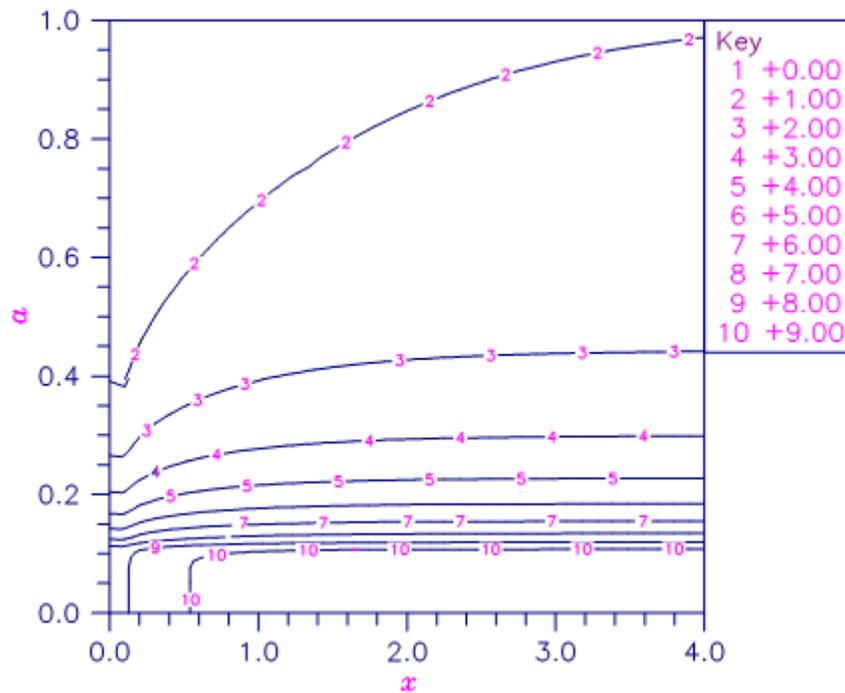


Figure 42, Contour Plot of $\gamma(a, x)$

Example

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

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

int 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)

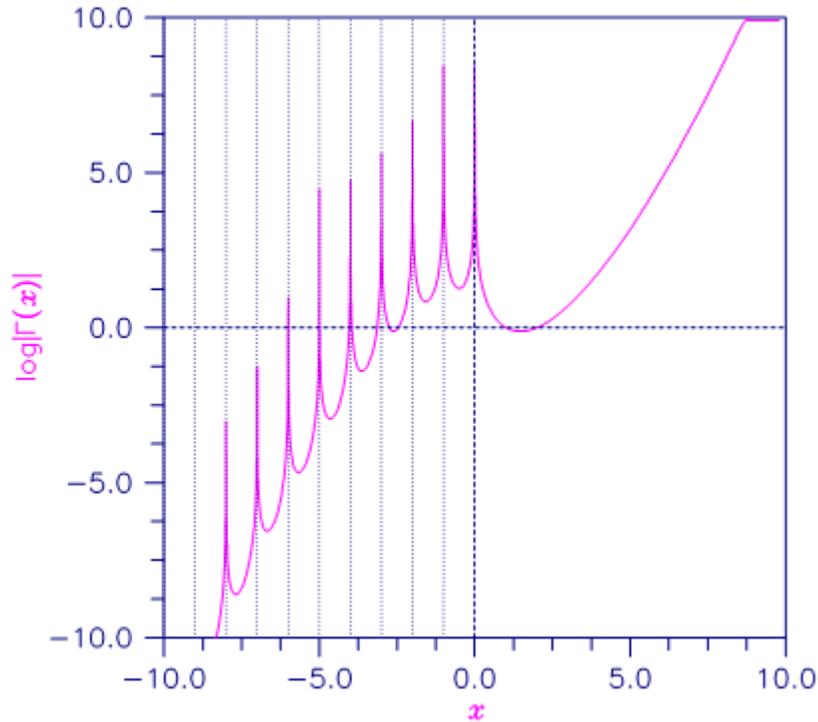
Point at which the logarithm of the absolute value of the gamma function is to be evaluated.

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 43, Plot of $\log|\Gamma(x)|$

Example

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

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

int 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

IMSL5_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

IMSL5_NEGATIVE_INTEGER

The argument for the function cannot be a negative integer.

IMSL5_NEAR_NEG_INT_FATAL

The argument for the function is too close to a negative integer.

IMSL5_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 varies slightly from run to run on the same machine.

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

int main()
{
    int    k;
    double sum, time;

    /* Sum 1 million values */
    for (sum=0, k=1; k<=1000000; k++)
        sum += k;

    /* Get amount of CPU time used */
    time = imsls_ctime();
    printf("sum = %f\n", sum);
    printf("time = %f\n", time);
    return 0;
}
```

Output

```
sum = 500000500000.000000  
time = 0.820000
```

Reference Material

Contents

User Errors

What Determines Error Severity	1915
Kinds of Errors and Default Actions	1916
Errors in Lower-level Functions	1917
Functions for Error Handling	1917
Threads and Error Handling	1918
Use of Informational Error to Determine Program Action	1918
Additional Examples	1918

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 `IMSL_TERMINAL` may be informational errors. The include file, *imsl.h*, defines each of `IMSL_NOTE`, `IMSL_ALERT`, `IMSL_WARNING`, `IMSL_FATAL`, `IMSL_TERMINAL`, `IMSL_WARNING_IMMEDIATE`, and `IMSL_FATAL_IMMEDIATE` as enumerated data type *imsl_error*.

IMSL_NOTE. A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations.

Default attributes: PRINT=NO, STOP=NO

IMSL_ALERT. An *alert* indicates that a function value has been set to 0 due to underflow.

Default attributes: PRINT=NO, STOP=NO

IMSL_WARNING. A *warning* indicates the existence of a condition that may require corrective action by the user or calling 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

IMSL_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

IMSL_TERMINAL. A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors 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

IMSL_WARNING_IMMEDIATE. An *immediate warning* error is identical to a warning error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=NO

IMSL_FATAL_IMMEDIATE. An *immediate fatal* error is identical to a fatal error, except it is printed immediately.

Default attributes: PRINT=YES, STOP=YES

The user can set PRINT and STOP attributes by calling function [imsls_error_options](#) as described in Chapter 15, *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 three ways with the IMSL error-handling system:

1. Change the default actions.
2. Determine the code of an informational error so as to take corrective action.
3. Initialize the error handling systems.

The functions that support these actions are:

- [imsls_error_options](#)
Sets the actions to be taken when errors occur.
- [imsls_error_type](#)
Retrieves the *imsl_error* enum error type value.
- [imsls_error_code](#)
Retrieves the integer code for an informational error.
- [imsls_error_message](#)
Retrieves the error message string.

- [imsls_initialize_error_handler](#)

Initializes the IMSL C Stat Library error handling system for the current thread. This function is not required but is always allowed. Use of this function is advised if the possibility of low heap memory exists when calling the IMSL C Stat Library for the first time in the current thread.

These functions are documented in Chapter 15, *Utilities*.

Threads and Error Handling

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options using [imsls_error_options](#) for each thread by calling [imsls_error_options](#) from within each thread.

See [Example 3](#) and [Example 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.

Appendix A References

Abdi

Abdi, Herve (2010), *Partial least squares regression and projection on latent structure regression (PLS regression)*, Wiley Interdisciplinary Reviews: Computational Statistics, 2, 97-106.

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.

Agrawal and Srikant

Agrawal, R. and Srikant, R. (1994), "Fast algorithms for mining association rules," *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29 - September 1, 1994.

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.

Aha

Aha, D. W. (1991). Incremental constructive induction: An instance-based approach. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 117--121). Evanston, ILL: Morgan Kaufmann.

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, H. (1980), *Seasonal Adjustment by Bayesian Modeling*, Journal of Time Series Analysis, Vol 1, 1-13.

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.

Asuncion and Newman

Asuncion, A.and Newman, D.J. (2007), UCI Machine Learning Repository, <http://archive.ics.uci.edu/ml/>. Irvine, CA: University of California, School of Information and Computer Science.

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.

Baker

Baker, J. E. (1987), Reducing Bias and Inefficiency in the Selection Algorithm. *Genetic Algorithms and their Applications: Proceeding of the Second international Conference on Genetic Algorithms*, 14-21.

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**, 156-157.

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: <https://www.salford-systems.com/cart.php>.

Breiman

Breiman, Leo (2001) *Random Forests*, Statistics Department, University of California, Berkeley. See <https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf>.

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.), *Neural computing: 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.

Brockwell and Davis

Brockwell, Peter J. and Davis, Richard A. (1977), *Time Series: Theory and Methods*, *Springer Series in Statistics*

Brown and Benedetti

Brown, Morton B. and Jacqueline 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*.

Chang and Lin

Chang, Chih-Chung; Lin, Chih-Jen (2011). "LIBSVM: A library for support vector machines". *ACM Transactions on Intelligent Systems and Technology* **2** (3).

Chatfield and Yar

Chatfield, C., Yar, M. (1988), Holt-Winters Forecasting; Some Practical Issues, *J. Royal Stat. Soc., Series D.* 7, (2), 129-140.

Chatfield, C., Yar, M. (1991), Prediction intervals for multiplicative Holt-Winters, *International Journal of Forecasting*. No. 7, 31-37.

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.

Coley

Coley, D. A. (1999), *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific Publishing Co.

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.

Cranley and Patterson

Cranley, R. and Patterson, T.N.L. (1976), Randomization of Number Theoretic Methods for Multiple Integration, *SIAM Journal of Numerical Analysis*, **13**, 904-914.

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 Lilliefors's test statistic for normality, *The American Statistician*, **40**, 294-296.

Davis and Rabinowitz

Davis, P.J. and Rabinowitz, P. (1984), *Methods of Numerical Integration*, Academic Press, 482-483.

De Jong

De Jong, K. A. (1975), An Analysis of the Behavior of a Class of Genetic Adaptive Systems. (Doctoral dissertation, Univ. of Michigan). *Dissertation Abstracts International* 36(10), 5140B. (University Microfilms No. 76-9381).

De Jong

de Jong, Sijmen (1993), *SIMPLS: An alternative approach to partial least squares regression*, *Chemometrics and Intelligent Laboratory Systems*, **18**, 251-263.

Demiroz et al.

Demiroz, G., H. A. Govenir, and N. Ilter (1988), "Learning Differential Diagnosis of Erythematous-Squamous Diseases using Voting Feature Intervals", *Artificial Intelligence in Medicine*.

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.

Doornik

Doornik, J.A. (2005), An Improved Ziggurat Method to Generate Normal Random Samples, <http://www.doornik.com/research/ziggurat.pdf>, University of Oxford.

Draper and Smith

Draper, N.R. and H. Smith (1981), *Applied Regression Analysis*, 2d ed., John Wiley & Sons, New York.

Dunnnett and Sobel

Dunnnett, C. W. and Sobel, M. (1955), Approximations to the Probability Integral and Certain Percentage Points of a Multivariate analogue of Student's t -distribution. *Biometrika*, **42**, 258-260.

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.

Eklblom

Eklblom, Hakan (1973), Calculation of linear best L_p -approximations, *BIT*, **13**, 292-300.

Eklblom, 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.

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.

Engle, R.F. and C.W.J. Granger

Engle, R.F. and C.W.J. Granger. *Long-run Economic Relationships: Readings in Cointegration*. Advanced Texts in Econometrics. Oxford University Press. New York, 1991.

Fan, Chen, and Joachims

Fan, Rong-en, Pai-hsuen Chen and Thorsten Joachims, *Working Set Selection Using Second Order Information for Training SVM*, Journal of Machine Learning Research, 2005.

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.

Frank and Friedman

Frank, Ildiko E., and Jerome J. Friedman (1993), *A Statistical View of Some Chemometrics Regression Tools*, Technometrics, Volume 35, Issue 2, pp. 109-135.

Friedman

Friedman, Jerome J. (2002), Stochastic Gradient Boosting, *Computational Statistics & Data Analysis*, 38 (4), 367-378.

Frey and Slate

Frey, P. W. and D. J. Slate. (1991), "Letter Recognition using Holland-style Adaptive Classifiers". (Machine Learning Vol 6 #2).

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.

Genz

Genz, A. (1992), Numerical Computation of Multivariate Normal Probabilities. *J. Comp. Graph Stat.*, **1**, 141-149.

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.

Gnanadesikan

Gnanadesikan, R. *Methods for Statistical Data Analysis of Multivariate Observations*. Wiley. New York. (1977).

Goldberg

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley Publishing Co.

Goldberg, D. E. and Deb, K. (1991), A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In G. Rawlins, Ed., *Foundations of Genetic Algorithms*. Morgan Kaufmann.

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, *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 χ^2 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 (New Jersey), 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.

Hastie et al

Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. Springer, New York.

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.

Hollmén

Hollmén, Jaakko, "Process Modeling Using the Self-Organizing Map," 15.2.1996, Helsinki University of Technology.

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.

Holland

Holland, J.H. (1975), *Adaptation in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press.

Hosmer and Lemeshow

Hosmer, D W. and Lemeshow, S (2000), *Applied Logistic Regression*, 2nd ed., John Wiley & Sons, New York.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hutchinson

Hutchinson, J. M. (1994) *A Radial Basis Function Approach to Financial Time 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.

Johansen

Johansen, S. (1988). *Statistical Analysis of Cointegration Vectors*. Journal of Economic Dynamics and Control. v 12 , pp 231-54.

Johansen, S. (1995). *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*. Oxford University Press, Oxford.

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 Kotz

Johnson, N.L. and Kotz, S. (1972), *Distributions in Statistics: Continuous Multivariate Distributions*, John Wiley & Sons, Inc., 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.

Kass

Kass, G.V. An Exploratory Technique for Investigating Large Quantities of Categorical Data, *Applied Statistics*, Vol. 29, No. 2 (1980), pp. 119-127.

Keast

Keast, P. (1973) Optimal Parameters for Multidimensional Integration, *SIAM Journal of Numerical Analysis*, **10**, 831-838.

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*, 2nd edition, Duxbury Press.

Kim and Jennrich

Kim, P.J., and R.I. Jennrich (1973), Tables of the exact sampling distribution of the two sample Kolmogorov-Smirnov criterion D_{mn} ($m < n$), in *Selected Tables in Mathematical Statistics*, Volume 1, (edited by H. L. Harter and D.B. Owen), American Mathematical Society, Providence, Rhode Island.

Kinderman and Ramage

Kinderman, A.J., and J.G. Ramage (1976), Computer generation of normal random variables, *Journal of the American Statistical Association*, **71**, 893-896.

Kinderman et al.

Kinderman, A.J., J.F. Monahan, and J.G. Ramage (1977), Computer methods for sampling from Student's t distribution, *Mathematics of Computation* **31**, 1009-1018.

Kinnucan and Kuki

Kinnucan, P. and H. Kuki (1968), *A Single Precision INVERSE Error Function Subroutine*, Computation Center, University of Chicago.

Kirk

Kirk, Roger E. (1982), *Experimental Design: Procedures for the Behavioral Sciences*, 2d ed., Brooks/Cole Publishing Company, Monterey, Calif.

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.

Konishi and Kitagawa

Konishi, S. and Kitagawa, G (2008), *Information Criteria and Statistical Modeling*, Springer, New York.

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 Theoretical 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 LLRAN-DOM, 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*, 1, 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.

Lütkepohl

Lutkepohl, *New Introduction to Multiple Time Series Analysis*. Springer. 2007, Chapter 12.

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.

Marsaglia and Tsang

Marsaglia, G. and Tsang, W. W. (2000), The Ziggurat Method for Generating Random Variables, *Journal of Statistical Software*, **5-8**, 1-7.

Martens and Martens

Martens, Harald, and Magni Martens (2000), *Modified Jack-knife estimation of parameter uncertainty in bilinear modelling by partial least squares regression (PLSR)*, Food Quality and Preference, 11(1-2), 5-16.

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, Lausanne.

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.

Mitchell

Mitchell, M. (1996), *An Introduction to Genetic Algorithms*, MIT Press.

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 Hillstom (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.

Otto et al

Otto, M.C., Bell, W.R. and Burman, J.P. (1987), "An Iterative GLS Approach to Maximum Likelihood Estimation of Regression Models With ARIMA Errors," American Statistical Association, *Proceedings of the Business and Economics Statistics Section*, 632-637.

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. and Tandy, D. (2000), Fast and Accurate Calculation of Owen's T-Function, *J. Statistical Software*, **5**, Issue 5, 1-25.

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/>.

Quinlan (1987). *Simplifying Decision Trees*. Int J Man-Machine Studies 27, pp. 221-234.

Rajaraman and Ullman

Rajaraman Anand and Ullman, Jeffrey David (2011), *Mining of Massive Datasets*, Cambridge University Press, Cambridge, UK.

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.

Royston, J. P. (1991), Approximating the Shapiro-Wilk W -test for non-normality, *Statistics and Computing*, **2**, 117-119.

Rumelhart

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, 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 Lioni

Sallas, William M. and Abby M. Lioni (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.

Savasere, Omiecinski, and Navathe

Savasere, Ashok; Omiecinski, Edward; and Navathe, Shamkant (1995), "An Efficient Algorithm for Mining Association Rules in Large Databases", *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995

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.

Shampine

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179-180.

Siegel

Siegel, 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.

Snedecor and Cochran

Snedecor, George W. and Cochran, William G. (1967) *Statistical Methods*, 6th edition, Iowa State University Press, 296-298.

Snedecor, George W. and Cochran, William G. (1967) *Statistical Methods*, 6th edition, Iowa State University Press, 432-436.

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.

Stephens, M.A. (1986): Tests based on EDF statistics. In: D'Agostino, R.B. and Stephens, M.A., eds.: Goodness-of-Fit Techniques. Marcel Dekker, New York.

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.

Story

Storey, John D. (2003). "The Positive False Discovery Rate: A Bayesian Interpretation and the q -value." *The Annals of Statistics*. Vol. 31, No. 6, pp 2013-2035.

Storey, John D. (2002). "A Direct Approach to False Discovery Rates." *Journal of the Royal Statistical Society, Series B*. 64, part 3, pp 479-498.

Storey, John D. and Robert Tibshirani (2003). "Statistical Significance for Genomewide Studies." *PNAS*. Vol. 100, No. 16. pp 9440-9445.

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.

Thisted

Thisted, Ronald. A. (1988). *Elements of Statistical Computing: Numerical Computation*. Chapman & Hall, New York.

Thompson

Thompson, James R, (1989), *Empirical Model Building*, John Wiley & Sons, New York.

Tong

Tong, Y. L. (1990), *The Multivariate Normal Distribution*, Springer-Verlag, 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.

Wetzel

Wetzel, A. (1983), *Evaluation of the Effectiveness of Genetic Algorithms in Combinatorial Optimization*, Unpublished manuscript, Univ. of Pittsburg, Pittsburg.

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.

Wold, Sjöström and Eriksson

Wold, Svante, Michael Sjöström, and Lennart Eriksson (2001), *PLS-regression: a basic tool of chemometrics*, Chemometrics and Intelligent Laboratory Systems, Volume 58, 109-130.

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.

Xi et al

Ruibin X., Lin N., and Chen Y., (2008), "Compression and Aggregation for Logistic Regression Analysis in Data Cubes," *IEEE Transactions on Knowledge and Data Engineering*. Vol. 1, No. 1.

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.

Yoav and Hochberg

Benjamini, Y., Hochberg, Y., (1995), "Controlling the False Discovery Rate: A Practical and Powerful approach to Multiple Testing." *Journal of the Royal Statistical Society, Series B*. Vol. 57, No. 1., pp . 289-300.

Appendix B Alphabetical Summary of Functions

[\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#)

A

Function

`ad_normality_test`

`ancovar`

`anova_balanced`

`anova_factorial`

`anova_nested`

`anova_oneway`

`apriori`

`aggr_apriori`

`arma`

`arma_forecast`

`ascii_read`

`autocorrelation`

`auto_arima`

Purpose Statement

Performs an Anderson-Darling test for normality.

Analyzes a one-way classification model with covariates.

Analyzes a balanced complete experimental design for a fixed, random, or mixed model.

Analyzes a balanced factorial design with fixed effects.

Analyzes a completely nested random model with possibly unequal numbers in the subgroups.

Analyzes a one-way classification model.

Computes the frequent itemsets in a transaction set.

Computes the frequent itemsets in a transaction set using aggregation.

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

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

Reads freely-formatted ASCII files.

Computes the sample autocorrelation function of a stationary time series.

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_parm`

Estimates structural breaks in non-stationary univariate time series.

`auto_uni_ar`

Automatic selection and fitting of a univariate autoregressive time series model.

B

Function

`bayesian_seasonal_adj`

`beta`

`beta_cdf`

`beta_incomplete`

`beta_inverse_cdf`

`binomial_cdf`

`binomial_coefficient`

`binomial_pdf`

`bivariate_normal_cdf`

`box_cox_transform`

Purpose Statement

Decomposes a time series into trend, seasonal, and an error component.

Evaluates the complete beta function.

Evaluates the beta probability distribution function.

Evaluates the real incomplete beta function.

Evaluates the inverse of the beta distribution function.

Evaluates the binomial distribution function.

Evaluates the binomial coefficient.

Evaluates the binomial probability function.

Evaluates the bivariate normal distribution function.

Performs a Box-Cox transformation.

C

Function	Purpose Statement
<code>canonical_correlation</code>	Given an input array of deviate values, generates a canonical correlation array.
<code>categorical_glm</code>	Analyzes categorical data using logistic, Probit, Poisson, and other generalized linear models.
<code>chi_squared_cdf</code>	Evaluates the chi-squared distribution function.
<code>chi_squared_inverse_cdf</code>	Evaluates the inverse of the chi-squared distribution function.
<code>chi_squared_normality_test</code>	Performs a chi-squared test for normality.
<code>chi_squared_test</code>	Performs a chi-squared goodness-of-fit test.
<code>cluster_hierarchical</code>	Performs a hierarchical cluster analysis given a distance matrix.
<code>cluster_k_means</code>	Performs a K -means (centroid) cluster analysis.
<code>cluster_number</code>	Computes cluster membership for a hierarchical cluster tree.
<code>cochran_q_test</code>	Performs a Cochran Q test for related observations.
<code>complementary_chi_squared_cdf</code>	Calculates the complement of the chi-squared distribution.
<code>complementary_F_cdf</code>	Calculates the complement of the F distribution function.
<code>complementary_non_central_F_cdf</code>	Evaluates the complementary noncentral F cumulative distribution function (CDF).
<code>complementary_t_cdf</code>	Calculates the complement of the Student's t distribution function.
<code>contingency_table</code>	Performs a chi-squared analysis of a two-way contingency table.
<code>continuous_table_setup</code>	Sets up a table to generate pseudorandom numbers from a general continuous distribution.

<code>covariances</code>	Computes the sample variance-covariance or correlation matrix.
<code>cox_stuart_trends_test</code>	Performs the Cox and Stuart' sign test for trends in location and dispersion.
<code>crd_factorial</code>	Analyzes data from balanced and unbalanced completely randomized experiments.
<code>crosscorrelation</code>	Computes the sample cross-correlation function of two stationary time series.
<code>cvm_normality_test</code>	Performs a Cramer-von-Mises test for normality.

D

Function

`data_sets`

`decision_tree`

`decision_tree_predict`

`decision_tree_print`

`decision_tree_free`

`difference`

`discrete_table_setup`

`discrete_uniform_cdf`

`discrete_uniform_inverse_cdf`

`discrete_uniform_pdf`

`discriminant_analysis`

`dissimilarities`

Purpose Statement

Retrieves a commonly analyzed data set.

Generates a decision tree for a single response variable and two or more predictor variables.

Computes predicted values using a decision tree.

Prints a decision tree.

Frees the memory associated with a decision tree.

Differences a seasonal or nonseasonal time series.

Sets up a table to generate pseudorandom numbers from a general discrete distribution.

Evaluates the discrete uniform cumulative distribution function (CDF).

Evaluates the inverse of the discrete uniform cumulative distribution function (CDF).

Evaluates the discrete uniform probability density function (PDF).

Performs discriminant function analysis.

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

E

Function

`empirical_quantiles`

`error_code`

`error_message`

`error_options`

`error_type`

`estimate_missing`

`exact_enumeration`

`exact_network`

`exponential_cdf`

`exponential_inverse_cdf`

`exponential_pdf`

Purpose Statement

Computes empirical quantiles.

Returns the code corresponding to the error message from the last function called.

Gets the text of the error message from the last function called.

Sets various error handling options.

Gets the type corresponding to the error message from the last function called.

Estimates missing values in a time series.

Computes exact probabilities in a two-way contingency table, using the total enumeration method.

Computes exact probabilities in a two-way contingency table using the network algorithm.

Evaluates the exponential cumulative distribution function (CDF).

Evaluates the inverse of the exponential cumulative distribution function (CDF).

Evaluates the exponential probability density function (PDF).

F

Function

`factor_analysis`

`false_discovery_rates`

`faure_next_point`

`fclose`

`fopen`

`free`

`free_apriori_itemsets`

`free_association_rules`

`friedmans_test`

Purpose Statement

Extracts initial factor-loading estimates in factor analysis.

Calculate the False Discovery Rate (FDR) q -values corresponding to a set of p -values from multiple simultaneous hypothesis tests.

Computes a shuffled Faure sequence.

Closes a file opened by `imsls_fopen`.

Opens a file using the C runtime library used by the IMSL C Stat Library.

Frees memory returned from an IMSL C Stat Library function.

Frees the memory allocated within a frequent itemsets structure.

Frees the memory allocated within an association rules structure.

Performs Friedman's test for a randomized complete block design.

G

Function

`ga_chromosome`

`ga_clone_chromosome`

`ga_clone_individual`

`ga_clone_population`

`ga_copy_chromosome`

`ga_copy_individual`

`ga_copy_population`

`ga_decode`

`ga_encode`

`ga_free_individual`

`ga_free_population`

`ga_grow_population`

`ga_individual`

`ga_merge_population`

`ga_mutate`

`ga_population`

Purpose Statement

Codes and decodes binary information from phenotypes to a chromosome.

Returns a new copy of an existing chromosome.

Returns a new copy of an existing individual.

Returns a new copy of an existing population.

Copies the contents of one chromosome into another chromosome.

Copies the contents of one individual into another individual.

Copies the contents of one population into another population.

Decodes an individual's chromosome into its binary, nominal, integer and real phenotypes.

Encodes an individual's binary, nominal, integer and real phenotypes into its chromosome.

Frees memory allocated to an existing individual.

Frees memory allocated to an existing population.

Adds individuals to an existing population.

Creates an *lmsls_f_individual* data structure from user supplied phenotypes.

Creates a new population by merging two populations with identical chromosome structures.

Performs the mutation operation on an individual's chromosome.

Creates an *lmsls_f_population* data structure from user supplied individuals.

<code>ga_random_population</code>	Creates an <i>lmsls_f_population</i> data structure from randomly generated individuals.
<code>gamma</code>	Evaluates the real gamma functions.
<code>gamma_cdf</code>	Evaluates the gamma distribution function.
<code>gamma_incomplete</code>	Evaluates the incomplete gamma function.
<code>gamma_inverse_cdf</code>	Evaluates the inverse of the gamma distribution function.
<code>garch</code>	Computes estimates of the parameters of a GARCH (p, q) model.
<code>genetic_algorithm</code>	Optimizes a user defined fitness function using a tailored genetic algorithm.
<code>geometric_cdf</code>	Evaluates the discrete geometric cumulative distribution function (CDF).
<code>geometric_inverse_cdf</code>	Evaluates the inverse of the discrete geometric cumulative distribution function (CDF).
<code>geometric_pdf</code>	Evaluates the discrete geometric probability density function (PDF).

H

Function

[homogeneity](#)

[hw_time_series](#)

[hypergeometric_cdf](#)

[hypergeometric_pdf](#)

[hypothesis_partial](#)

[hypothesis_scph](#)

[hypothesis_test](#)

Purpose Statement

Conducts Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.

Calculates parameters and forecasts using the Holt-Winters Multiplicative or Additive forecasting method for seasonal data.

Evaluates the hypergeometric distribution function.

Evaluates the hypergeometric probability function.

Constructs a completely testable hypothesis.

Sums of cross products for a multivariate hypothesis.

Tests for the multivariate linear hypothesis.

Function

`impute_missing`

`initialize_error_handler`

Purpose Statement

Locates and optionally replaces dependent variable missing values with nearest neighbor estimates.

Initializes the IMSL C Stat Library error handling system.

K

Function

`kalman`

`kaplan_meier_estimates`

`kohonenSOM_forecast`

`kohonenSOM_trainer`

`kolmogorov_one`

`kolmogorov_two`

`kruskal_wallis_test`

`k_trends_test`

Purpose Statement

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

Computes Kaplan-Meier estimates of survival probabilities in stratified samples.

Calculates forecasts using a trained Kohonen network.

Trains a Kohonen network.

Performs a Kolmogorov-Smirnov's one-sample test for continuous distributions.

Performs a Kolmogorov-Smirnov's two-sample test.

Performs a Kruskal-Wallis's test for identical population medians.

Performs k-sample trends test against ordered alternatives.

L

Function

`lack_of_fit`

`latin_square`

`lattice`

`life_tables`

`lilliefors_normality_test`

`lnorm_regression`

`log_beta`

`log_gamma`

`logistic_regression`

`logistic_reg_predict`

`lognormal_cdf`

`lognormal_inverse_cdf`

`lognormal_pdf`

Purpose Statement

Performs lack-of-fit test for an univariate time series or transfer function given the appropriate correlation function.

Analyzes data from latin-square experiments.

Analyzes balanced and partially-balanced lattice experiments.

Produces population and cohort life tables.

Performs a Lilliefors test for normality.

Fits a multiple linear regression model using criteria other than least squares.

Evaluates the log of the real beta function.

Evaluates the logarithm of the absolute value of the gamma function.

Fits a binomial or multinomial logistic regression model using iteratively reweighted least squares.

Predicts a binomial or multinomial outcome given an estimated model and new values of the independent variables.

Evaluates the lognormal cumulative distribution function (CDF).

Evaluates the inverse of the lognormal cumulative distribution function (CDF).

Evaluates the lognormal probability density function (PDF).

M

Function

`machine (float)`

`machine (integer)`

`mat_mul_rect`

`max_arma`

`max_likelihood_estimates`

`mlff_classification_trainer`

`mlff_initialize_weights`

`mlff_network`

`mlff_network_forecast`

`mlff_network_free`

`mlff_network_init`

`mlff_network_read`

`mlff_network_trainer`

`mlff_network_write`

Purpose Statement

Returns information describing the computer's floating-point arithmetic.

Returns integer information describing the computer's arithmetic.

Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any triple product.

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

Calculates maximum likelihood estimates for the parameters of one of several univariate probability distributions.

Trains a multilayered feedforward neural network for classification.

Initializes weights for multilayered feedforward neural networks prior to network training using one of four user selected methods.

Creates a multilayered feedforward neural network.

Calculates forecasts for trained multilayered feedforward neural networks.

Frees memory allocated for an *ImSls_f_NN_Network* data structure.

Initializes a data structure for training a neural network.

Retrieves a neural network from a file previously saved.

Trains a multilayered feedforward neural network.

Writes a trained neural network to an ASCII file.

<code>mlff_pattern_classification</code>	Calculates classifications for trained multilayered feedforward neural networks.
<code>multi_crosscorrelation</code>	Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.
<code>multiple_comparisons</code>	Performs Student-Newman-Keuls multiple comparisons test.
<code>multivar_normality_test</code>	Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.
<code>multivariate_normal_cdf</code>	Computes the cumulative distribution function for the multivariate normal distribution.

N

Function	Purpose Statement
<code>naive_bayes_classification</code>	Classifies unknown patterns using a previously trained Naïve Bayes classifier.
<code>naive_bayes_trainer</code>	Trains a Naïve Bayes classifier.
<code>nb_classifier_free</code>	Frees memory allocated to an <i>imsls_f_nb_classifier</i> data structure.
<code>nb_classifier_read</code>	Retrieves a Naive Bayes Classifier previously filed using <code>imsls_f_nb_classifier_write</code> .
<code>nb_classifier_write</code>	Writes a Naive Bayes Classifier to an ASCII file for later retrieval using <code>imsls_f_nb_classifier_read</code> .
<code>noether_cyclical_trend</code>	Performs a Noether's test for cyclical trend.
<code>non_central_beta_cdf</code>	Evaluates the noncentral beta cumulative distribution function (<i>CDF</i>).
<code>non_central_beta_inverse_cdf</code>	Evaluates the inverse of the noncentral beta cumulative distribution function (<i>CDF</i>).
<code>non_central_beta_pdf</code>	Evaluates the noncentral beta probability density function (<i>PDF</i>).
<code>non_central_chi_sq</code>	Evaluates the noncentral chi-squared distribution function.
<code>non_central_chi_sq_inv</code>	Evaluates the inverse of the noncentral chi-squared function.
<code>non_central_chi_sq_pdf</code>	Evaluates the noncentral chi-squared probability density function.
<code>non_central_F_cdf</code>	Evaluates the noncentral <i>F</i> cumulative distribution function.
<code>non_central_F_inverse_cdf</code>	Evaluates the inverse of the noncentral <i>F</i> cumulative distribution function.
<code>non_central_F_pdf</code>	Evaluates the noncentral <i>F</i> probability density function.
<code>non_central_t_cdf</code>	Evaluates the noncentral Student's <i>t</i> distribution function.

<code>non_central_t_inv_cdf</code>	Evaluates the inverse of the noncentral Student's t distribution function.
<code>non_central_t_pdf</code>	Evaluates the noncentral Student's t probability density function.
<code>nonlinear_optimization</code>	Fits a nonlinear regression model using Powell's algorithm.
<code>nonlinear_regression</code>	Fits a nonlinear regression model.
<code>nonparam_hazard_rate</code>	Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.
<code>normal_cdf</code>	Evaluates the standard normal (Gaussian) distribution function.
<code>normal_inverse_cdf</code>	Evaluates the inverse of the standard normal (Gaussian) distribution function.
<code>normal_one_sample</code>	Computes statistics for mean and variance inferences using a sample from a normal population.
<code>normal_two_sample</code>	Computes statistics for mean and variance inferences using samples from two normal populations.

O

Function

`omp_options`

`output_file`

Purpose Statement

Sets various OpenMP options.

Sets the output file or the error message output file.

P

Function	Purpose Statement
<code>page</code>	Sets or retrieves the page width or length.
<code>pareto_cdf</code>	Evaluates the Pareto cumulative probability distribution function.
<code>pareto_pdf</code>	Evaluates the Pareto probability density function.
<code>partial_autocorrelation</code>	Computes the sample partial autocorrelation function of a stationary time series.
<code>partial_covariances</code>	Computes partial covariances or partial correlations from the covariance or correlation matrix.
<code>permute_matrix</code>	Permutes the rows or columns of a matrix.
<code>permute_vector</code>	Rearranges the elements of a vector as specified by a permutation.
<code>pls_regression</code>	Performs partial least squares regression for one or more response variables and one or more predictor variables.
<code>poisson_cdf</code>	Evaluates the Poisson distribution function.
<code>poisson_pdf</code>	Evaluates the Poisson probability function.
<code>poly_prediction</code>	Computes predicted values, confidence intervals, and diagnostics after fitting a polynomial regression model.
<code>poly_regression</code>	Performs a polynomial least-squares regression.
<code>pooled_covariances</code>	Computes a pooled variance-covariance from the observations.
<code>principal_components</code>	Computes principal components.
<code>prop_hazards_gen_lin</code>	Analyzes time event data via the proportional hazards model.

R

Function

`random_arma`

`random_beta`

`random_binomial`

`random_cauchy`

`random_chi_squared`

`random_exponential`

`random_exponential_mix`

`random_gamma`

`random_general_continuous`

`random_general_discrete`

`random_geometric`

`random_GFSR_table_get`

`random_GFSR_table_set`

`random_hypergeometric`

`random_logarithmic`

`random_lognormal`

Purpose Statement

Generates pseudorandom ARMA process numbers.

Generates pseudorandom numbers from a beta distribution.

Generates pseudorandom binomial numbers.

Generates pseudorandom numbers from a Cauchy distribution.

Generates pseudorandom numbers from a chi-squared distribution.

Generates pseudorandom numbers from a standard exponential distribution.

Generates pseudorandom mixed numbers from a standard exponential distribution.

Generates pseudorandom numbers from a standard gamma distribution.

Generates pseudorandom numbers from a general continuous distribution.

Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.

Generates pseudorandom numbers from a geometric distribution.

Retrieves the current table used in the GFSR generator.

Sets the current table used in the GFSR generator.

Generates pseudorandom numbers from a hypergeometric distribution.

Generates pseudorandom numbers from a logarithmic distribution.

Generates pseudorandom numbers from a lognormal distribution.

<code>random_MT32_init</code>	Initializes the 32-bit Mersenne Twister generator using an array.
<code>random_MT32_table_get</code>	Retrieves the current table used in the 32-bit Mersenne Twister generator.
<code>random_MT32_table_set</code>	Sets the current table used in the 32-bit Mersenne Twister generator.
<code>random_MT64_init</code>	Initializes the 64-bit Mersenne Twister generator using an array.
<code>random_MT64_table_get</code>	Retrieves the current table used in the 64-bit Mersenne Twister generator.
<code>random_MT64_table_set</code>	Sets the current table used in the 64-bit Mersenne Twister generator.
<code>random_multinomial</code>	Generates pseudorandom numbers from a multinomial distribution.
<code>random_mvar_from_data</code>	Generates pseudorandom numbers from a multivariate distribution determined from a given sample.
<code>random_mvar_gaussian_copula</code>	Given a Cholesky factorization of a correlation matrix, generates pseudorandom numbers from a Gaussian Copula distribution.
<code>random_mvar_t_copula</code>	Given a Cholesky factorization of a correlation matrix, generates pseudorandom numbers from a Student's t Copula distribution.
<code>random_neg_binomial</code>	Generates pseudorandom numbers from a negative binomial distribution.
<code>random_normal</code>	Generates pseudorandom numbers from a normal, $N(\mu, \sigma^2)$, distribution.
<code>random_normal_multivariate</code>	Generates pseudorandom numbers from a multivariate normal distribution.
<code>random_npp</code>	Generates pseudorandom numbers from a nonhomogeneous Poisson process.
<code>random_option</code>	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.
<code>random_option_get</code>	Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.

<code>random_order_normal</code>	Generates pseudorandom order statistics from a standard normal distribution.
<code>random_order_uniform</code>	Generates pseudorandom order statistics from a uniform (0, 1) distribution.
<code>random_orthogonal_matrix</code>	Generates a pseudorandom orthogonal matrix or a correlation matrix.
<code>random_permutation</code>	Generates a pseudorandom permutation.
<code>random_poisson</code>	Generates pseudorandom numbers from a Poisson distribution.
<code>random_sample</code>	Generates a simple pseudorandom sample from a finite population.
<code>random_sample_indices</code>	Generates a simple pseudorandom sample of indices.
<code>random_seed_get</code>	Retrieves the current value of the seed used in the IMSL random number generators.
<code>random_seed_set</code>	Initializes a random seed for use in the IMSL random number generators.
<code>random_sphere</code>	Generates pseudorandom points on a unit circle or κ -dimensional sphere.
<code>random_stable</code>	Generates pseudorandom numbers from a stable distribution.
<code>random_student_t</code>	Generates pseudorandom Student's t .
<code>random_substream_seed_get</code>	Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.
<code>random_table_get</code>	Retrieves the current table used in the shuffled generator.
<code>random_table_set</code>	Sets the current table used in the shuffled generator.
<code>random_table_twoway</code>	Generates a pseudorandom two-way table.
<code>random_triangular</code>	Generates pseudorandom numbers from a triangular distribution.
<code>random_uniform</code>	Generates pseudorandom numbers from a uniform (0, 1) distribution.

<code>random_uniform_discrete</code>	Generates pseudorandom numbers from a discrete uniform distribution.
<code>random_von_mises</code>	Generates pseudorandom numbers from a von Mises distribution.
<code>random_weibull</code>	Generates pseudorandom numbers from a Weibull distribution.
<code>randomness_test</code>	Performs a test for randomness.
<code>ranks</code>	Computes the ranks, normal scores, or exponential scores for a vector of observations.
<code>rcbd_factorial</code>	Analyzes data from balanced and unbalanced randomized complete-block experiments.
<code>regression</code>	Fits a multiple linear regression model using least squares.
<code>regression_arima</code>	Fits a univariate, non-seasonal ARIMA time series model with the inclusion of one or more regression variables.
<code>regression_prediction</code>	Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.
<code>regression_selection</code>	Selects the best multiple linear regression models.
<code>regression_stepwise</code>	Builds multiple linear regression models using forward selection, backward selection or stepwise selection.
<code>regression_summary</code>	Produces summary statistics for a regression model given the information from the fit.
<code>regressors_for_glm</code>	Generates regressors for a general linear model.
<code>robust_covariances</code>	Computes a robust estimate of a covariance matrix and mean vector.

S

Function

`scale_filter`

`seasonal_fit`

`set_user_fcn_return_flag`

`shapiro_wilk_normality_test`

`sign_test`

`simple_statistics`

`sort_data`

`split_plot`

`split_split_plot`

`strip_plot`

`strip_split_plot`

`support_vector_trainer`

`support_vector_classification`

`survival_estimates`

Purpose Statement

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.

Estimates the optimum seasonality parameters for a time series using an autoregressive model, $AR(p)$, to represent the time series.

Indicates a condition has occurred in a user-supplied function necessitating a return to the calling function.

Performs the Shapiro-Wilk test for normality.

Performs a sign test.

Computes basic univariate statistics.

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

Analyzes a wide variety of split-plot experiments with fixed, mixed or random factors.

Analyzes data from split-split-plot experiments.

Analyzes data from strip-plot experiments.

Analyzes data from strip-split-plot experiments.

Trains a Support Vector Machines classifier

Classifies patterns using a previously trained Support Vector Machines classifier

Estimates using various parametric models.

`survival_glm`

Analyzes survival data using a generalized linear model.

`svm_classifier_free`

Frees memory allocated for a Support Vector Machines classifier

T

Function

`t_cdf`

`t_inverse_cdf`

`table_oneway`

`table_twoway`

`tie_statistics`

`time_series_class_filter`

`time_series_filter`

`ts_outlier_forecast`

`ts_outlier_identification`

Purpose Statement

Evaluates the Student's t distribution function.

Evaluates the inverse of the Student's t distribution function.

Tallies observations into one-way frequency table.

Tallies observations into a two-way frequency table.

Computes tie statistics for a sample of observations.

Converts time series data sorted with nominal classes in decreasing chronological order to useful format for processing by a neural network.

Converts time series data to the format required for processing by a neural network.

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.

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

Function

`unsupervised_nominal_filter`

`unsupervised_ordinal_filter`

Purpose Statement

Converts nominal data into a series of binary encoded columns for input to a neural network.

Converts ordinal data into percentages.

V

Function

`version`

`vector_autoregression`

Purpose Statement

Returns integer information describing the version of the library, license number, operating system, and compiler.

Estimates a vector auto-regressive time series model with optional moving average components.

W

Function

`wilcoxon_rank_sum`

`wilcoxon_sign_rank`

`write_apriori_itemsets`

`write_association_rules`

`write_matrix`

`write_options`

Purpose Statement

Performs a Wilcoxon rank sum test.

Performs a Wilcoxon sign rank test.

Prints frequent itemsets.

Prints association rules.

Prints a rectangular matrix (or vector) stored in contiguous memory locations.

Sets or retrieves an option for printing a matrix.

Product Support

Contacting IMSL Support

Users within support warranty may contact Rogue Wave Software regarding the use of the IMSL C Numerical Library. IMSL Support can consult on the following topics:

- Clarity of documentation
- Possible IMSL-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem

Not included in these topics are mathematical/statistical consulting and debugging of your program.

Refer to the following for Rogue Wave Product Support contact information:

- <http://www.roguewave.com/help-support/customer-support>

The following describes the procedure for consultation with IMSL Support:

1. Include your IMSL license number
2. Include the product name and version number: IMSL C Numerical Library, Version *<current_version>*.
3. Include compiler and operating system version numbers
4. Include the name of the function for which assistance is needed and a description of the problem

Index

A

additive (AO) 766
AIC 759
Airline Data 759
Akaike's information criterion 748
alpha factor analysis 963
Analysis of Covariance 331
ANCOVAR 325, 331
 one-way classification
 model 325
ARIMA models 783
 forecasts 716
 least squares estimates 754
 least-square estimates 696
 method of moments 698
 method of moments
 estimates 712
 method of moments
 estimation 701
 multiplicative seasonal 790
ARIMA models XE 754
ARIMA time series 724
ARMA model 709
asciiRead Function
 column-oriented files 1852
 delimiters 1851
 filtering and substitution 1850
 physical vs. logical
 records 1850
 row-oriented files 1851
Autoregressive Moving Average
Model 690

B

balanced 363
balanced experimental design 363
beta distribution function
 inverse 1102
beta functions 1897, 1900, 1902
binary encoded 1749
binomial coefficient 1895

binomial distribution 1074, 1201,
1203
binomial distributions 1223, 1233,
1245, 1290
binomial probability 1076
bivariate normal distribution
 function 1113
Bonferroni method 318
bounded scaling 1734
Box-Cox transformation 822

C

C Language
 imsld_ascii_read function 1844
Cartesian coordinates 1308
cauchy distributions 1254
chi-squared distribution
 function 1115, 1118, 1120
chi-squared distributions 1256
chi-squared goodness-of-fit
 test 635
chi-squared statistics 518, 523
chi-squared test 634
classification model
 one-way 313
cluster analysis 920
cluster membership 934
cluster_hierarchical 928
cluster_number 934
Cochran Q test 624
coefficient
 excess (kurtosis) 25
 skewness 25
column-oriented ASCII files 1852
confidence intervals 196
 mean 26
constants 1865, 1868
constrained non-linear
 optimization 1210
contingency coefficient 523
contingency tables 533, 536
 two-way 519
correlation matrix 266, 1298
counts 26, 67
covariates 325
Cox and Stuart sign test 600
CPU 1913
Cramer's V 523
Cramer-von Mises test 673
Crd factorial
 factorial experiments 381
 pooled location
 interaction 381
 unbalanced 375
 unbalanced completely ran-
 domized experiments 375
crd factorial 375
crosscorrelation 839
cross-correlation function 839, 846,
1000, 1007, 1053, 1063
cubic spline interpolation 868

D

data
 ASCII, column-oriented
 files 1852
 ASCII, delimiters 1851
 ASCII, filtering and substitution
 while reading 1850
 ASCII, row-oriented files 1851
data sets 1871
datetime format
 C Structure 1854
 in Python 1852
datetime structure 1854
detection 765
deviation, standard 25
diagnostic checking 690
discrete uniform
 distributions 1238
dissimilarities 923

- distribution functions
 beta 1100
 chi-squared 1115, 1120
 chi-squared, noncentral 1123
 inverse 1127
 F_cdf
 inverse 1138, 1143
 F_inverse_cdf 1141
 inverse 1182
 Dunn-Sidák method 318
- E**
 eigensystem analysis 920
 empirical quantiles 35
 median 37
 empirical tests 1221
 error handling 8, 1819, 1826, 1915
 estimate of scale
 simple robust 30
 excess 29
 exponential smoothing 814, 815,
 875
- F**
 F statistic 51
 factor analysis 920, 952
 factorial 339
 factorial design
 analysis 339
 false discovery rates 497
 Faure 1382
 Faure sequence 1380
 faure_next_point 1381
 finite population 1350
 Fisher's information 1211
 Fisher's LSD 319
 forecasting 690
 forecasts 774
 ARMA models 774
 GARCH 774, 883
 frequency tables 61
 Friedman's test 618
- G**
 gamma distribution function 1160
 gamma functions 1904, 1907, 1910
 gamma_inverse_cdf 1163
- Gaussian distribution
 functions 1171
 general continuous
 distribution 1287
 general discrete distribution 1240,
 1241, 1245, 1290
 general distributions 634
 Generalized Feedback Shift
 Register 1219
 generalized feedback shift register
 method 1218
 generalized linear models 518
 Genetic Algorithm
 Alleles 1493
 Chromosome
 Base-2 encoding 1493
 decoding 1493
 encoding 1493
 Gray encoding 1493
 Mutation 1496
 Crossover 1498
 Inversion 1498
 Multiple crossovers 1550
 Partially matched
 (PMX) 1498, 1550
 Examples
 N-Queens 1563
 Traveling De Jong 1554
 Traveling salesman 1563
 Individual 1494
 chromosome 1495
 Performance
 Off-line 1552
 On-line 1552
 Velocity 1552
 Phenotype 1493
 Population 1496
 Chromosome 1496
 Generation 1496
 Reproduction 1500
 Reproduction Model
 Selection 1497
 Scaling
 Linear 1554
 Sigma 1554
 Genetic Algorithm
 Chromosome 1493
 geometric distributions 1226
 GFSR 1354
- GFSR generator 1219, 1365, 1366
 goodness-of-fit tests 634
 gradient_boosting 1469
 Gray code 1383
- H**
 Haar measure 1299
 hierarchical cluster analysis 928
 hierarchical cluster tree 934
 Holt-Winters method 873
 homogeneity 484
 hypergeometric distribution
 function 1084
 hypergeometric distributions 1228
 hypergeometric_pdf 1087
 hyper-rectangle 1380
 hypothesis 152
- I**
 image analysis 962
 imslld_ascii_read function 1844,
 1854
 optional arguments 1846
 required arguments 1844
 return value 1844
 innovational (AO) 792
 innovational (IO) 766, 792
 integrated rate function 1344
 iterative generalized least
 squares 730, 742
- K**
 Kalman filtering 889
 Kaplan_meier_estimates 1001
 Kaplan-meier estimates 1000
 Kaplan-Meier estimates
 computes 1000
 Kappa analysis 518
 K-dimensional sphere 1308
 kernel functions 999, 1053
 K-means analysis 939
 Kolmogorov one-sample test 657
 Kolmogorov two-sample test 661
 Kruskal-Wallis test 615
 k-sample trends test 628

- L**
- lack-of-fit test 861
 - lack-of-fit tests 94
 - latent structures 254
 - latin square 396
 - Lattice
 - 3x3 balanced-lattice 410
 - balanced lattice
 - experiments 409
 - intra-Block Error 411
 - partially-balanced lattice
 - experiments 405, 410
 - lattice 405
 - Least Absolute Value 96, 230, 234, 243
 - Least Maximum Value 96, 230
 - Least Squares
 - Alternatives
 - Least Absolute Value 96
 - Lp Norm 96
 - Lebesgue measure 1382
 - level Shift (LS) 766
 - level shift (LS) 792
 - likelihood function 692
 - linear dependence 90
 - linear discriminant function
 - analysis 973
 - linear regression
 - multiple 85
 - logarithmic distributions 1231
 - logical records 1850
 - logistic regression 560
 - low-discrepancy 1383
 - Lp Norm 96, 235
- M**
- Mardia's multivariate
 - measures 667
 - Mardia's multivariate tests 665
 - matrices 923, 1875
 - matrix of dissimilarities 923
 - matrix storage modes 22
 - maximum 26, 30
 - maximum likelihood
 - estimates 895, 1210
 - maximum likelihood
 - estimation 1205
 - mean 25, 29, 39, 41
 - measures of association 518, 524
 - measures of prediction 525
 - median 30
 - absolute deviation 30
 - memory allocation 9
 - Mersenne Twister 1370, 1371, 1374, 1375, 1376, 1379
 - method of moments 1211
 - minimum 26, 29
 - missing values 97
 - MLE 1205
 - models 206
 - general linear 98, 737
 - nonlinear regression 91
 - module
 - datetime
 - datetime module 1852
 - Monte Carlo applications 1221
 - multinomial distribution 1305
 - Multiple comparisons 491
 - multiple comparisons 491
 - Multiple comparisons test
 - Bonferroni, Tukey's 491
 - Student-Newman-Keuls 491
 - multiple linear regression
 - models 108, 165, 176, 230, 349, 363, 591, 596, 600, 606, 618, 657, 661, 857
 - multiple_crosscorrelation 846
 - multiplicative congruential
 - generator 1218
 - multiplicative generator 1218
 - multivariate distribution 1301
 - multivariate functions
 - multivariate normal 1174
 - multivariate general linear
 - hypothesis 157
 - multivariate normal distribution
 - function 1174
 - multivariate normal distribution,
 - simulation 1294
- N**
- Naive Bayes
 - Classification 1569
 - Continuous
 - attributes 1569
 - Nominal attributes 1569
 - Count Table 1578
 - Error rate 1569, 1570, 1579
 - Fisher's Iris classification
 - problem 1581
 - Missing Values 1591
 - Probability Density Function (PDF) 1574
 - Gamma 1576
 - Gaussian 1574
 - Log-Normal 1575
 - Poisson 1576
 - User defined 1576, 1577, 1592
 - Smoothing Parameter 1580
 - Smoothing Zero
 - Correction 1581
 - Zero Correction 1574
 - nested 349
 - nested random model 349, 354
 - network 1634
 - Neural Network Classification
 - continuous attributes 1663
 - nominal attributes 1663
 - target attributes 1710
 - Weight initialization 1661
 - Noether test 596
 - noncentral chi-squared distribution
 - function 1123
 - inverse 1127
 - noncentral Student's t distribution
 - function 1193, 1196
 - nonhomogeneous Poisson
 - process 1342
 - nonlinear model 219
 - nonlinear regression models 91
 - nonparam_hazard_rate 1053
 - nonparametric hazard rate
 - estimation 1053
 - nonuniform generators 1220
 - normality test 648, 651, 654
- O**
- one-step-ahead forecasts 866
 - oneway 313

- one-way frequency table 56
- order statistics 1331, 1334
- orthogonal matrix 1298
- outlier
 - description 792
- outlier contaminated series 774
- output files 1813
- overflow 8
- P**
- parameter estimation 690
- partial covariances 275
- partial least squares regression 254
- partially tested hypothesis 145
- permutations 1880, 1882
- phi 523
- physical records 1850
- PLS
 - PRESS 256
- PLSR
 - cross-validation 256
 - PRESS 256
 - SIMPLS 256
- Poisson distribution function 1089
- Poisson distribution, simulation 1236
- poisson_pdf 1092
- polynomial models 86
- polynomial regression 187
- pooled variance-covariance 281
- population 1063
- principal components 945
- printing
 - matrices 1798
 - options 1807
 - retrieving page size 1805
- probability limits
 - ARMA models 774
 - outlier contaminated series 776
- prop_hazards_gen_lin 1007
- pseudorandom number generators 634
- pseudorandom numbers 1241, 1245, 1266, 1275, 1282, 1284, 1290
- pseudorandom permutation 1346
- pseudorandom sample 1348
- PsLSR
 - k-fold 256
- p-values 524
- PyIMSL Studio
 - allowable datetime formats 1852
- R**
- random number generator 1370, 1371, 1374, 1375, 1376, 1379
- random numbers
 - beta distribution 1251
 - exponential distribution 1258
 - gamma distribution 1263
 - seed
 - current value 1357
 - initializing 1361
 - selecting generator 1354, 1356
- random numbers generators 1269
- random_MT32_init 1370
- random_MT32_table_get 1371
- random_MT32_table_set 1374
- random_MT64_init 1375
- random_MT64_table_get 1376
- random_MT64_table_set 1379
- randomness test 676
- range 26, 30
- rcbd factorial 386
- records, physical vs. logical 1850
- regression arima 729, 737, 741
- regression models 85, 124, 133
- robust covariances 288
- row-oriented ASCII files 1851
- S**
- sample autocorrelation function 833
- sample correlation function 690
- sample partial autocorrelation function 857
- scale filter 1734
- Scheffé method 319
- scores
 - normal 76
- seasonal adjustment 756
- seasonality parameters 755
- Seed 1219
- seed 1359
- serial number 1817
- shuffled generator 1362, 1363
- sign test 587
- simulation of random variables 1218
- skewness 29
- Split plot
 - blocking factor 430
 - completely randomized 422
 - completely randomized design 430
 - experiments 422
 - fixed effects 430
 - IMSLS_RCBD default setting 431
 - random effects 432
 - randomized complete block design 422, 430
 - randomizing whole-plots 431
 - split plot factor 431
 - split plot factors 430
 - whole plot 430
 - whole plot factor 431
 - whole plot factors 430
- split plot 422
- Split Plots
 - whole-plots 422
- Split-split plot
 - split-plot factors 436
 - split-split-plot
 - experiments 436
 - sub-plot factors 436
 - whole plot factors 436
- split-split plot 436
- stable distribution 1272
- standard deviation 41
- state vector 889
- statespace model 889
- stationary/stationarity 713
- strip plot 452
- strip-split plot 463
- Student's t distribution

function 1184, 1189
inverse 1187
summary statistics 91
survival probabilities 1000, 1001

T

t statistic 50
temporary change (TC) 766, 793
tests for randomness 634
Thread Safe 15
 multithreaded application 15
 single-threaded application 15
 threads and error handling 1918
tie statistics 606
time domain methodology 690
time event data 1007
Time series 737
time series 690, 1337
 difference 827
time series class filter 1744
time series filter 1741
transformation 690
transformations 96
triangular distributions 1277
Tukey method 318
Tukey-Kramer method 318
two-way contingency table 1312
two-way table 1311

U

unable to identify (UI) 766, 793
underflow 8
uniform distribution,
 simulation 1279
unit sphere 1309
univariate statistics 25, 543, 1022,
 1046, 1260
unscales 1734
unsupervised nominal filter 1749
unsupervised ordinal filter 1753
update equations 890

V

variable selection 86
variables

datetime structure 1854
variance 25, 29
 for two normal populations 44
variation, coefficient of 30

W

weighted least squares 91
white noise
 Gaussian 769, 771, 778
 process 767
white noise process 712
Wilcoxon rank sum test 609, 1600
Wilcoxon signed rank test 591
Wilcoxon two-sample test 616
Wolfer Sunspot series 713, 751

Y

yates 505

Z

zeros of a function 1843
z-score scaling 1734

