Visual Numerics®

VERSION 5.5

IMSL

C Numerical Library™

User's Guide

VOLUME 4 of 4: C Stat Library™ [CHAPTERS 8-14]

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

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

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

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

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

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

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

Visual Numerics International GmbH
Zettachring 10
D-70567 Stuttgart
Germany

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

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

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

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

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

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

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

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

IMSL Fortran and C and Java Application Development Tools

CStat Library /V2 Table of Contents

Chapter 8: Time Series and Forecasting

Routines

Usage Notes

The functions in this chapter assume the time series does not contain any missing observations. If missing values are present, they should be set to NaN (see the routine `imsls_f_machine`, Chapter 14), and the routine will return an appropriate error message. To enable fitting of the model, the missing values must be replaced by appropriate estimates.

General Methodology

A major component of the model identification step concerns determining if a given time series is stationary. The sample correlation functions computed by routines `imsls_f_autocorrelation` (page541), `imsls_f_crosscorrelation` (page 546), `imsls_f_multi_crosscorrelation` (page 552), and `imsls_f_partial_autocorrelation` (page 560) may be used to diagnose the presence of nonstationarity in the data, as well as to indicate the type of transformation required to induce stationarity. The family of power transformations provided by routine `imsls_f_box_cox_transform` (page 537) coupled with the ability to difference the transformed data using routine `imsls_f_difference` (page 532) affords a convenient method of transforming a wide class of nonstationary time series to stationarity.

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

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

Time Domain Methodology

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

ARIMA Model (Autoregressive Integrated Moving Average)

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

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

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

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

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

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

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

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` (page 532) 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` (page 517) for preliminary parameter estimates. These estimates can be used as initial values into the least-squares procedure by including argument `IMSLS_LEAST_SQUARES` in a call to function `imsls_f_arma`. Other initial estimates provided by the user can be used. The least-squares procedure can be used to compute conditional or unconditional least-squares estimates of the parameters, depending on the choice of the backcasting length. The parameter estimates from either the method of moments or least-squares procedures can be input to function `imsls_f_arma_forecast` (page 527) through the `arma_info` structure. The functions for preliminary parameter estimation, least-squares parameter estimation, and forecasting follow the approach of Box and Jenkins (1976, Programs 2–4, pp. 498–509).

arma

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

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_arma`.

Required Arguments

int `n_observations` (Input)
Number of observations.

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

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

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

Return Value

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

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_arma (int n_observations, float z[], int p, int q,
        IMSLS_NO_CONSTANT, or
        IMSLS_CONSTANT,
        IMSLS_AR_LAGS, int ar_lags[],
        IMSLS_MA_LAGS, int ma_lags[],
        IMSLS_METHOD_OF_MOMENTS, or
        IMSLS_LEAST_SQUARES,
        IMSLS_BACKCASTING, int length, float tolerance,
        IMSLS_CONVERGENCE_TOLERANCE,
                float convergence_tolerance,
        IMSLS_RELATIVE_ERROR, float relative_error,
        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_RETURN_USER, float *constant, float ar[], float ma[],
        IMSLS_ARMA_INFO, Imsls_f_arma **arma_info,
        0)
```

Optional Arguments

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

IMSLS_AR_LAGS, *int* ar_lags[] (Input)

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

IMSLS_MA_LAGS, *int* ma_lags[] (Input)

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

IMSLS_METHOD_OF_MOMENTS, *or*
IMSLS_LEAST_SQUARES

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

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

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

IMSLS_CONVERGENCE_TOLERANCE, *float* convergence_tolerance (Input)

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

IMSLS_RELATIVE_ERROR, *float* relative_error (Input)

Stopping criterion for use in the nonlinear equation solver used in both the method of moments and least-squares algorithms.
Default: relative_error = 100 × imsls_f_machine(4)
See documentation for function imsls_f_machine (Chapter 14, "Utilities").

IMSLS_MAX_ITERATIONS, *int* max_iterations (Input)

Maximum number of iterations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms.
Default: max_iterations = 200

IMSLS_MEAN_ESTIMATE, *float* *z_mean (Input or Input/Output)

On input, z_mean is an initial estimate of the mean of the time series z.

On return, `z_mean` contains an update of the mean.
If `IMSLS_NO_CONSTANT` and `IMSLS_LEAST_SQUARES` are specified, `z_mean` is not used in parameter estimation.

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

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

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

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

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

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

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

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

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

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

Description

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

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

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

Method of Moments Estimation

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

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

for $t \in \{0, \pm1, \pm2, ...\}$

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

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

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

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

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

where

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

The overall constant θ_0 is estimated by the following:

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

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

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

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

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

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

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

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

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

and

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

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

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

The estimation procedure begins with the initial value

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

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

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

The random shock variance is estimated by the following:

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

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,

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

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

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

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

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

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

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

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

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

Consider the sum-of-squares function

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

where

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

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

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

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

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

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

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

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

dominates

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

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

In practice, a finite value of T will enable sufficient approximation of the unconditional sum-of-squares function. The values of $[A_T]$ needed to compute the unconditional sum of squares are computed iteratively with initial values of Z_t obtained by back forecasting. The residuals (including backcasts), estimate of random shock variance, and covariance matrix of the final parameter estimates also are computed. ARIMA parameters can be computed by using `imsls_f_difference` (page 532), with `imsls_f_arma`.

Examples

Example 1

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

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

for the ARMA(2, 1) model

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

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

$$\sigma_A^2$$

```
#include <imsls.h>

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

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

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

Output

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

Example 2

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

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

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

```
#include <imsls.h>

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

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

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

}
```

Output

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

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

Warning Errors

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

arma_forecast

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

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_arma_forecast`.

Required Arguments

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

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

Return Value

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

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_arma_forecast` (*Imsls_f_arma* `*arma_info`,
 int `n_predict`,
 `IMSLS_CONFIDENCE`, *float* `confidence`,
 `IMSLS_BACKWARD_ORIGIN`, *int* `backward_origin`,
 `IMSLS_RETURN_USER`, *float* `forecasts[]`,
 0)

Optional Arguments

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

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

`IMSLS_BACKWARD_ORIGIN`, *int* `backward_origin` (Input)

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

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

If specified, a user-specified array of length `n_predict` × (`backward_origin` + 3) as defined below.

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

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

Description

The Box-Jenkins forecasts and their associated probability limits for a nonseasonal ARMA model are computed given a sample of `n = n_observations` $\{Z_t\}$ for $t = 1, 2, ..., n$.

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

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

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

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

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

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

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

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

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

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

where the following is true:

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

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

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

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

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

$$\sigma_A^2$$

(returned from `imsls_f_arma`) and

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

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

The Box-Jenkins forecasts minimize the mean-square error

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

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

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

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

Example

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

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

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

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

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

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

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

Output

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

                     * * * Forecast Table * * *

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

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

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

`IMSLS_RETURN_USER`, *float* `w[]` (Output)
If specified, `w` contains the differenced series. If `IMSLS_EXCLUDE_FIRST` also is specified, `w` is of length `n_observations`. If `IMSLS_SET_FIRST_TO_NAN` is specified or neither `IMSLS_EXCLUDE_FIRST` nor `IMSLS_SET_FIRST_TO_NAN` is specified, `w` is of length `n_observations` − `n_lost`.

Description

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

Consider the backward shift operator B given by

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

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

$$\Delta_s Z_t = (1 - B^s)\, Z_t = Z_t - Z_{t-s} \qquad \text{for } s \geq 0$$

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

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

where d ≥ 0 is the order of differencing. Note that

$$\Delta_s^d Z_t$$

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

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

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

where n_L represents the number of observations “lost” because of differencing and NaN represents the missing value code. See the functions `imsls_f_machine` and `imsls_d_machine` (Chapter 14, “Utilities”) to retrieve missing values. Note that

$$n_L = \sum_j s_j d_j$$

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

Examples

Example 1

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

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

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

```
#include <imsls.h>

void main()

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

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

}
```

Output

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

```
21      133.000000  -8.000000
22      114.000000  -4.000000
23      140.000000  12.000000
```

Example 2

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

```
#include <imsls.h>

void main()
{

    int    i;
    int    n_observations = 24;
    int    n_differences = 2;
    int    periods[2] = {1, 12};
    int    n_lost;
    float  *z;
    float  *difference;
                  /* Get airline data */
    z = imsls_f_data_sets (4, 0);
                  /* Compute differenced time series when observations
                     lost are excluded from the differencing */
    difference = imsls_f_difference (n_observations, z,
                                     n_differences, periods,
                                     IMSLS_EXCLUDE_FIRST,
                                     IMSLS_LOST, &n_lost,
                                     0);
                  /* Print the number of lost observations */
    printf ("n_lost equals %d\n", n_lost);
    printf ("\n\ni\tz[i]\t        difference[i]\n");
                  /* Print the original time series and the differenced
                     time series */
    for (i = 0; i < n_observations - n_lost; i++)
        printf ("%d\t%f\t%f\n", i, z[i], difference[i]);
}
```

Output

```
n_lost equals 13


 i      z[i]          difference[i]
 0      112.000000    5.000000
 1      118.000000    1.000000
 2      132.000000   -3.000000
 3      129.000000   -2.000000
 4      121.000000   10.000000
 5      135.000000    8.000000
 6      148.000000    0.000000
 7      148.000000    0.000000
 8      136.000000   -8.000000
 9      119.000000   -4.000000
10      104.000000   12.000000
```

Fatal Errors

`IMSLS_PERIODS_LT_ZERO`	"period[#]" = #. All elements of "period" must be greater than 0.
`IMSLS_ORDER_NEGATIVE`	"order[#]" = #. All elements of "order" must be nonnegative.
`IMSLS_Z_CONTAINS_NAN`	"z[#]" = NaN; "z" can not contain missing values. There may be other elements of "z" that are equal to NaN.

box_cox_transform

Performs a forward or an inverse Box-Cox (power) transformation.

Synopsis

#include `<imsls.h>`

float `*imsls_f_box_cox_transform` (*int* `n_observations`, *float* `z[]`,
float `power`, ..., 0)

The type *double* function is `imsls_d_box_cox_transform`.

Required Arguments

int `n_observations` (Input)
Number of observations in `z`.

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

float `power` (Input)
Exponent parameter in the Box-Cox (power) transformation.

Return Value

Pointer to an internally allocated array of length `n_observations` containing the transformed data. To release this space, use `free`. If no value can be computed, then `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_box_cox_transform` (*int* `n_observations`, *float* `z[]`,
float `power`,
`IMSLS_SHIFT`, *float* `shift`,
`IMSLS_INVERSE_TRANSFORM`,
`IMSLS_RETURN_USER`, *float* `x[]`
0)

Optional Arguments

`IMSLS_SHIFT`, *float* `shift` (Input)
  Shift parameter in the Box-Cox (power) transformation. Parameter shift must satisfy the relation min ($z(i)$) + `shift` > 0.
  Default: `shift` = 0.0.

`IMSLS_INVERSE_TRANSFORM`
  If `IMSLS_INVERSE_TRANSFORM` is specified, the inverse transform is performed.

`IMSLS_RETURN_USER`, *float* `x[]` (Output)
  User-allocated array of length `n_observations` containing the transformed data.

Description

Function `imsls_f_box_cox_transform` performs a forward or an inverse Box-Cox (power) transformation of n = `n_observations` observations $\{Z_t\}$ for $t = 1, 2, ..., n$.

The forward transformation is useful in the analysis of linear models or models with nonnormal errors or nonconstant variance (Draper and Smith 1981, p. 222). In the time series setting, application of the appropriate transformation and subsequent differencing of a series can enable model identification and parameter estimation in the class of homogeneous stationary autoregressive-moving average models. The inverse transformation can later be applied to certain results of the analysis, such as forecasts and prediction limits of forecasts, in order to express the results in the scale of the original data. A brief note concerning the choice of transformations in the time series models is given in Box and Jenkins (1976, p. 328).

The class of power transformations discussed by Box and Cox (1964) is defined by

$$X_t = \begin{cases} \dfrac{(Z_t + \xi)^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \ln(Z_t + \xi) & \lambda = 0 \end{cases}$$

where $Z_t + \xi > 0$ for all t. Since

$$\lim_{\lambda \to 0} \frac{(Z_t + \xi)^\lambda - 1}{\lambda} = \ln(Z_t + \xi)$$

the family of power transformations is continuous.

Let λ = `power` and ξ = `shift`; then, the computational formula used by `imsls_f_box_cox_transform` is given by

$$X_t = \begin{cases} (Z_t + \xi)^\lambda & \lambda \neq 0 \\ \ln(Z_t + \xi) & \lambda = 0 \end{cases}$$

where $Z_t + \xi > 0$ for all t. The computational and Box-Cox formulas differ only in the scale and origin of the transformed data. Consequently, the general analysis of the data is unaffected (Draper and Smith 1981, p. 225).

The inverse transformation is computed by

$$X_t = \begin{cases} Z_t^{1/\lambda} - \xi & \lambda \neq 0 \\ exp(Z_t) - \xi & \lambda = 0 \end{cases}$$

where $\{Z_t\}$ now represents the result computed by `imsls_f_box_cox_transform` for a forward transformation of the original data using parameters λ and ξ.

Examples

Example 1

The following example performs a Box-Cox transformation with `power` = 2.0 on 10 data points.

```
#include <imsls.h>

void main() {
    int n_observations = 10;
    float power = 2.0;
    float *x;
    static float z[10] ={
        1.0, 2.0, 3.0, 4.0, 5.0, 5.5, 6.5, 7.5, 8.0, 10.0};

    /* Transform Data using Box Cox Transform */
    x = imsls_f_box_cox_transform(n_observations, z, power, 0);

    imsls_f_write_matrix("Transformed Data", 1, n_observations, x, 0);

    free(x);
}
```

Output

```
                           Transformed Data
         1            2            3            4            5            6
       1.0          4.0          9.0         16.0         25.0         30.2

         7            8            9           10
      42.2         56.2         64.0        100.0
```

Example 2

This example extends the first example—an inverse transformation is applied to the transformed data to return to the orignal data values.

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

```
void main() {
    int n_observations = 10;
    float power = 2.0;
    float *x, *y;
    static float z[10] ={
        1.0, 2.0, 3.0, 4.0, 5.0, 5.5, 6.5, 7.5, 8.0, 10.0};

    /* Transform Data using Box Cox Transform */
    x = imsls_f_box_cox_transform(n_observations, z, power, 0);

    imsls_f_write_matrix("Transformed Data", 1, n_observations, x, 0);

    /* Perform an Inverse Transform on the Transformed Data */
    y = imsls_f_box_cox_transform(n_observations, x, power,
            IMSLS_INVERSE_TRANSFORM, 0);

    imsls_f_write_matrix("Inverse Transformed Data", 1, n_observations, y,
0);

    free(x);
    free(y);
}
```

Output

```
                         Transformed Data
         1            2            3            4            5            6
       1.0          4.0          9.0         16.0         25.0         30.2

         7            8            9           10
      42.2         56.2         64.0        100.0

                     Inverse Transformed Data
         1            2            3            4            5            6
       1.0          2.0          3.0          4.0          5.0          5.5

         7            8            9           10
       6.5          7.5          8.0         10.0
```

Fatal Errors

`IMSLS_ILLEGAL_SHIFT`	"shift" = # and the smallest element of "z" is "z[#]" = #. "shift" plus "z[#]" = #. "shift" + "z[i]" must be greater than 0 for *i* = 1, ..., "n_observations". "n_observations" = #.
`IMSLS_BCTR_CONTAINS_NAN`	One or more elements of "z" is equal to NaN (Not a number). No missing values are allowed. The smallest index of an element of "z" that is equal to NaN is #.
`IMSLS_BCTR_F_UNDERFLOW`	Forward transform. "power" = #. "shift" = #. The minimum element of "z" is "z[#]" = #. ("z[#]"+ "shift") ^ "power" will underflow.

`IMSLS_BCTR_F_OVERFLOW`	Forward transformation. “power” = #. “shift” = #. The maximum element of “z” is “z[#]” = #. (“z[#]” + “shift”) ^ “power” will overflow.
`IMSLS_BCTR_I_UNDERFLOW`	Inverse transformation. “power” = #. The minimum element of “z” is “z[#]” = #. exp(“z[#]”) will underflow.
`IMSLS_BCTR_I_OVERFLOW`	Inverse transformation. “power” = #. The maximum element of “z[#]” = #. exp(“z[#]”) will overflow.
`IMSLS_BCTR_I_ABS_UNDERFLOW`	Inverse transformation. “power” = #. The element of “z” with the smallest absolute value is “z[#]” = #. “z[#]” ^ (1/ “power”) will underflow.
`IMSLS_BCTR_I_ABS_OVERFLOW`	Inverse transformation. “power” = #. The element of “z” with the largest absolute value is “z[#]” = #. “z[#]” ^ (1/ “power”) will overflow.

autocorrelation

Computes the sample autocorrelation function of a stationary time series.

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_autocorrelation`.

Required Arguments

int `n_observations` (Input)
    Number of observations in the time series `x`. `n_observations` must be greater than or equal to 2.

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

int `lagmax` (Input)
    Maximum lag of autocovariance, autocorrelations, and standard errors of autocorrelations to be computed. `lagmax` must be greater than or equal to 1 and less than `n_observations`.

Return Value

Pointer to an array of length `lagmax` + 1 containing the autocorrelations of the time series `x`. The *0*-th element of this array is 1. The *k*-th element of this array contains the autocorrelation of lag *k* where *k* = 1, ..., `lagmax`.

Synopsis with Optional Arguments

```
#include <imsls.h>

float imsls_f_autocorrelation (int n_observations, float x[],
        int lagmax,
        IMSLS_RETURN_USER, float autocorrelations[],
        IMSLS_PRINT_LEVEL, int iprint,
        IMSLS_ACV, float **autocovariances,
        IMSLS_ACV_USER, float autocovariances[],
        IMSLS_SEAC, float **standard_errors, int
                  se_option,
        IMSLS_SEAC_USER, float standard_errors[],
                  int se_option,
        IMSLS_X_MEAN_IN, float x_mean_in,
        IMSLS_X_MEAN_OUT, float *x_mean_out,
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `autocorrelations[]` (Output)
If specified, `autocorrelations` is an array of length `lagmax + 1` containing the `autocorrelations` of the time series `x`. The *o*th element of this array is 1. The *k*th element of this array contains the `autocorrelation` of lag *k* where *k* = 1, `. . .`, `lagmax`.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option. Default = 0.

`Iprint`	**Action**
`0`	No printing is performed.
`1`	Prints the mean and variance.
`2`	Prints the mean, variance, and autocovariances.
`3`	Prints the mean, variance, autocovariances, autocorrelations, and standard errors of autocorrelations.

`IMSLS_ACV`, *float* `**autocovariances` (Output)
Address of a pointer to an array of length `lagmax` + 1 containing the variance and autocovariances of the time series `x`. The *0*-th element of this array is the variance of the time series `x`. The *k*th element contains the autocovariance of lag *k* where *k* = 1, ..., `lagmax`.

`IMSLS_ACV_USER`, *float* `autocovariances[]` (Output)
If specified, `autocovariances` is an array of length `lagmax` + 1 containing the variance and autocovariances of the time series `x`. See `IMSLS_ACV`.

`IMSLS_SEAC`, *float* `**standard_errors`, *int* `se_option` (Output)
Address of a pointer to an array of length `lagmax` containing the standard errors of the autocorrelations of the time series `x`. Method of computation for standard errors of the autocorrelations is chosen by `se_option`.

Se_option	**Action**
1	Compute the standard errors of `autocorrelations` using Barlett's formula.
2	Compute the standard errors of `autocorrelations` using Moran's formula.

`IMSLS_SEAC_USER`, *float* `standard_errors[]`, *int* `se_option` (Output)
If specified, `autocovariances` is an array of length `lagmax` containing the standard errors of the autocorrelations of the time series `x`. See `IMSLS_SEAC`.

`IMSLS_X_MEAN_IN`, *float* `x_mean_in` (Input)
User input the estimate of the time series `x`.

`IMSLS_X_MEAN_OUT`, *float* `*x_mean_out` (Output)
If specified, `x_mean_out` is the estimate of the mean of the time series `x`.

Description

Function `imsls_f_autocorrelation` estimates the autocorrelation function of a stationary time series given a sample of n = `n_observations` observations $\{X_t\}$ for $t = 1, 2, \ldots, n$.

Let

$$\hat{\mu} = \texttt{x\_mean}$$

be the estimate of the mean μ of the time series $\{X_t\}$ where

$$\hat{\mu} = \begin{cases} \mu, & \mu \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_t & \mu \text{ unknown} \end{cases}$$

The autocovariance function σ(k) is estimated by

$$\hat{\sigma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu})(X_{t+k} - \hat{\mu}), \quad k = 0, 1, \ldots, K$$

where K = `lagmax`. Note that

$$\hat{\sigma}(0)$$

is an estimate of the sample variance. The autocorrelation function ρ(*k*) is estimated by

$$\hat{\rho}(k) = \frac{\hat{\sigma}(k)}{\hat{\sigma}(0)}, \qquad k = 0, 1, \ldots, K$$

Note that

$$\hat{\rho}(0) \equiv 1$$

by definition.

The standard errors of the sample autocorrelations may be optionally computed according to argument `se_option` for the optional argument `IMSLS_SEAC`. One method (Bartlett 1946) is based on a general asymptotic expression for the variance of the sample autocorrelation coefficient of a stationary time series with independent, identically distributed normal errors. The theoretical formula is

$$\operatorname{var}\{\hat{\rho}(\mathrm{k})\} = \frac{1}{n}\sum_{i=-\infty}^{\infty}\left[\rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k)\right]$$

where

$$\hat{\rho}(k)$$

assumes μ is unknown. For computational purposes, the autocorrelations r(k) are replaced by their estimates

$$\hat{\rho}(k)$$

for |k| ≤ K, and the limits of summation are bounded because of the assumption that r(k) = 0 for all k such that |k| > K.

A second method (Moran 1947) utilizes an exact formula for the variance of the sample autocorrelation coefficient of a random process with independent, identically distributed normal errors. The theoretical formula is

$$\operatorname{var}\{\hat{\rho}(k)\} = \frac{n-k}{n(n+2)}$$

where μ is assumed to be equal to zero. Note that this formula does not depend on the autocorrelation function.

Example

Consider the Wolfer Sunspot Data (Anderson 1971, page 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Function `imsls_f_autocorrelation` with optional arguments computes the estimated autocovariances, estimated autocorrelations, and estimated standard errors of the autocorrelations.

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

void main()
{
   float *result=NULL, data[176][2], x[100], xmean;
   int i, nobs = 100, lagmax = 20;
   float *acv=NULL, *seac=NULL;


   imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
   for (i=0;i<nobs;i++) x[i] = data[21+i][1];

   result = imsls_f_autocorrelation(nobs, x, lagmax,
                            IMSLS_X_MEAN_OUT, &xmean,
                            IMSLS_ACV, &acv,
                            IMSLS_SEAC, &seac, 1,
                            0);
   printf("Mean     = %8.3f\n", xmean);
   printf("Variance = %8.1f\n", acv[0]);
   printf("\nLag\t   ACV\t\t   AC\t\t   SEAC\n");
   printf("%2d\t%8.1f\t%8.5f\n", 0, acv[0], result[0]);
   for(i=1; i<21; i++)
      printf("%2d\t%8.1f\t%8.5f\t%8.5f\n", i, acv[i], result[i],
      seac[i-1]);

}
```

Output

```
Mean     =     46.976
Variance =     1382.9

Lag         ACV           AC          SEAC

 0         1382.9      1.00000
 1         1115.0      0.80629      0.03478
 2          592.0      0.42809      0.09624
 3           95.3      0.06891      0.15678
 4         -236.0     -0.17062      0.20577
 5         -370.0     -0.26756      0.23096
 6         -294.3     -0.21278      0.22899
 7          -60.4     -0.04371      0.20862
 8          227.6      0.16460      0.17848
 9          458.4      0.33146      0.14573
10          567.8      0.41061      0.13441
11          546.1      0.39491      0.15068
12          398.9      0.28848      0.17435
13          197.8      0.14300      0.19062
14           26.9      0.01945      0.19549
15          -77.3     -0.05588      0.19589
16         -143.7     -0.10394      0.19629
17         -202.0     -0.14610      0.19602
18         -245.4     -0.17743      0.19872
```

```
19          -230.8     -0.16691      0.20536
20          -142.9     -0.10332      0.20939
```

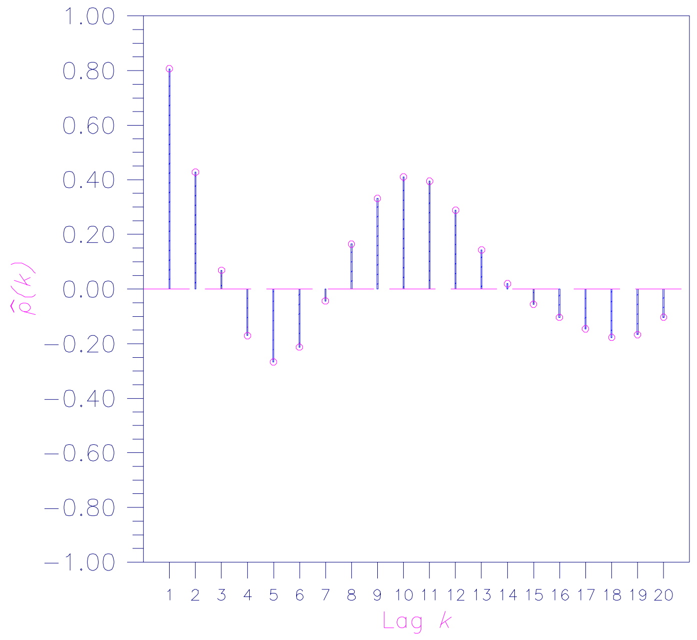


Figure 8-1 Sample Autocorrelation Function

crosscorrelation

Computes the sample cross-correlation function of two stationary time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_crosscorrelation` (*int* `n_observations`, *float* `x[]`,
    float `y[]`, *int* `lagmax`, ..., 0)

The type *double* function is `imsls_d_crosscorrelation`.

Required Arguments

int `n_observations` (Input)
    Number of observations in each time series. `n_observations` must be greater than or equal to 2.

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

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

int lagmax (Input)
Maximum lag of cross-covariances and cross-correlations to be computed. lagmax must be greater than or equal to 1 and less than n_observations.

Return Value

Pointer to an array of length 2*lagmax + 1 containing the cross-correlations between the time series x and y. The *k*th element of this array contains the cross-correlation between x and y at lag (*k*-lagmax) where *k* = 0, 1, …, 2*lagmax. To release this space, use free. If no solution can be computed, NULL is returned.

Synopsis with Optional Arguments

#include <imsls.h>

float *imsls_f_crosscorrelation (*int* n_observations, *float* x[],
float y[], *int* lagmax,
IMSLS_RETURN_USER, *float* crosscorrelations[],
IMSLS_PRINT_LEVEL, *int* iprint,
IMSLS_VARIANCES, *float* *x_variance, *float* *y_variance
IMSLS_SE_CCF, *float* **standard_errors, *int* se_option,
IMSLS_SE_CCF_USER, *float* standard_errors[], *int* se_option,
IMSLS_CROSS_COVARIANCES, *float* **cross_covariances,
IMSLS_CROSS_COVARIANCES_USER, *float* cross_covariances[],
IMSLS_INPUT_MEANS, *float* x_mean_in, *float* y_mean_in,
IMSLS_OUTPUT_MEANS, *float* *x_mean_out, *float* *y_mean_out,
0)

Optional Arguments

IMSLS_RETURN_USER, *float* crosscorrelations[] (Output)
If specified, crosscorrelations is an array of length 2*lagmax + 1 containing the cross-correlations between the time series x and y. The *k*th element of this array contains the cross-correlation between x and y at lag (*k*-lagmax) where *k* = 0, 1, …, 2*lagmax.

IMSLS_PRINT_LEVEL, *int* iprint (Input)
Printing option. Default = 0.

Iprint	Action
0	No printing is performed.
1	Prints the means and variances.
2	Prints the means, variances, and cross-covariances.
3	Prints the means, variances, cross-covariances, cross-correlations, and standard errors of cross-correlations.

IMSLS_VARIANCES, *float* *x_variance, *float* *y_variance (Output)
If specified, x_variance is variance of the time series x and y_variance is variance of the time series y.

`IMSLS_SE_CCF`, *float* `**standard_errors`, *int* `se_option` (Output)
Address of a pointer to an array of length 2*`lagmax` + 1containing the standard errors of the cross-correlations between the time series `x` and `y`. Method of computation for standard errors of the cross-correlations is chosen by `se_option`.

se_option	**Action**
1	Compute standard errors of cross-correlations using Bartlett's formula.
2	Compute standard errors of cross-correlations using Bartlett's formula with the assumption of no cross-correlation.

`IMSLS_SE_CCF_USER`, *float* `standard_errors[]`, *int* `se_option` (Output)
If specified, `standard_errors` is an array of length 2*`lagmax` + 1 containing the standard errors of the cross-correlations between the time series `x` and `y`. See `IMSLS_SE_CC`.

`IMSLS_CROSS_COVARIANCES`, *float* `**cross_covariances` (Output)
Address of a pointer to an array of length 2*`lagmax` + 1 containing the cross-covariances between the time series `x` and `y`. The *k*th element of this array contains the cross-covariances between `x` and `y` at lag (*k*-`lagmax`) where *k* = 0, 1, …, 2*`lagmax`.

`IMSLS_CROSS_COVARIANCES_USER`, *float* `cross_covariances[]` (Output)
If specified, `cross_covariances` is an array of length 2*`lagmax` + 1 the cross-covariances between the time series `x` and `y`. See `IMSLS_CROSS_COVARIANCES`.

`IMSLS_INPUT_MEANS`, *float* `x_mean_in`, *float* `y_mean_in` (Input)
If specified, `x_mean_in` is the user input of the estimate of the mean of the time series `x` and `y_mean_in` is the user input of the estimate of the mean of the time series `y`.

`IMSLS_OUTPUT_MEANS`, *float* `*x_mean_out`, *float* `*y_mean_out` (Output)
If specified, `x_mean_out` is the mean of the time series `x` and `y_mean_out` is the mean of the time series `y`.

Description

Function `imsls_f_crosscorrelation` estimates the cross-correlation function of two jointly stationary time series given a sample of n = `n_observations` observations $\{X_t\}$ and $\{Y_t\}$ for $t = 1, 2, \ldots, n$.

Let

$$\hat{\mu}_x = \texttt{x\_mean}$$

be the estimate of the mean μ_X of the time series $\{X_t\}$ where

$$\hat{\mu}_X = \begin{cases} \mu_X & \mu_X \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_t & \mu_X \text{ unknown} \end{cases}$$

The autocovariance function of $\{X_t\}$, $\sigma_X(k)$, is estimated by

$$\hat{\sigma}_X(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(X_{t+k} - \hat{\mu}_X), \quad k = 0, 1, \ldots, K$$

where K = `lagmax`. Note that

$$\hat{\sigma}_X(0)$$

is equivalent to the sample variance `x_variance`. The autocorrelation function $\rho_X(k)$ is estimated by

$$\hat{\rho}_X(k) = \frac{\hat{\sigma}_X(k)}{\hat{\sigma}_X(0)} \quad k = 0, 1, \ldots, K$$

Note that

$$\hat{\rho}_X(0) \equiv 1$$

by definition. Let

$$\hat{\mu}_Y = \text{y\_mean}, \hat{\sigma}_Y(k), \text{and } \hat{\rho}_Y(k)$$

be similarly defined.

The cross-covariance function $\sigma_{XY}(k)$ is estimated by

$$\hat{\sigma}_{XY}(k) = \begin{cases} \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \ldots, K \\ \frac{1}{n}\sum_{t=1-k}^{n}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \ldots, -K \end{cases}$$

The cross-correlation function $\rho_{XY}(k)$ is estimated by

$$\hat{\rho}_{XY}(k) = \frac{\hat{\sigma}_{XY}(k)}{\left[\hat{\sigma}_X(0)\hat{\sigma}_Y(0)\right]^{1/2}} \quad k = 0, \pm 1, \ldots, \pm K$$

The standard errors of the sample cross-correlations may be optionally computed according to argument `se_option` for the optional argument `IMSLS_SE_CCF`. One method is based on a general asymptotic expression for the variance of the sample cross-correlation coefficient of two jointly stationary time series with independent, identically distributed normal errors given by Bartlett (1978, page 352). The theoretical formula is

$$\begin{aligned}\operatorname{var}\{\hat{\rho}_{\mathrm{XY}}(k)\} &= \frac{1}{n-k}\sum_{i=-\infty}^{\infty}\Big[\rho_X(i)\rho_Y(i)+\rho_{XY}(i-k)\rho_{XY}(i+k) \\ &-2\rho_{XY}(k)\{\rho_X(i)\rho_{XY}(i+k)+\rho_{XY}(-i)\rho_Y(i+k)\} \\ &+\rho_{XY}^2(k)\left\{\rho_X(i)+\frac{1}{2}\rho_X^2(i)+\frac{1}{2}\rho_Y^2(i)\right\}\Big]\end{aligned}$$

For computational purposes, the autocorrelations $\rho_X(k)$ and $\rho_Y(k)$ and the cross-correlations $\rho_{XY}(k)$ are replaced by their corresponding estimates for $|k| \le K$, and the limits of summation are equal to zero for all k such that $|k| > K$.

A second method evaluates Bartlett's formula under the additional assumption that the two series have no cross-correlation. The theoretical formula is

$$\operatorname{var}\{\hat{\rho}_{\mathrm{XY}}(k)\} = \frac{1}{n-k}\sum_{i=-\infty}^{\infty}\rho_X(i)\rho_Y(i) \qquad k \ge 0$$

For additional special cases of Bartlett's formula, see Box and Jenkins (1976, page 377).

An important property of the cross-covariance coefficient is $\sigma_{XY}(k) = \sigma_{YX}(-k)$ for $k \ge 0$. This result is used in the computation of the standard error of the sample cross-correlation for lag $k < 0$. In general, the cross-covariance function is not symmetric about zero so both positive and negative lags are of interest.

Example

Consider the Gas Furnace Data (Box and Jenkins 1976, pages 532–533) where X is the input gas rate in cubic feet/minute and Y is the percent CO_2 in the outlet gas. Function `imsls_f_crosscorrelation` is used to compute the cross-covariances and cross-correlations between time series X and Y with lags from `−lagmax` = −10 through lag `lagmax` = 10. In addition, the estimated standard errors of the estimated cross-correlations are computed. The standard errors are based on the additional assumption that all cross-correlations for X and Y are zero.

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

#define nobs 296
#define lagmax 10

void main ()
{
  int i;
  float data[nobs][2], x[nobs], y[nobs];
  float *secc = NULL, *ccv = NULL, *cc = NULL;
  float xmean, ymean, xvar, yvar;

  imsls_f_data_sets (7, IMSLS_X_COL_DIM, 2, IMSLS_RETURN_USER, data, 0);
  for (i = 0; i < nobs; i++)
    {
      x[i] = data[i][0];
      y[i] = data[i][1];
    }

  cc = imsls_f_crosscorrelation (nobs, x, y, lagmax,
                           IMSLS_OUTPUT_MEANS, &xmean, &ymean,
                           IMSLS_VARIANCES, &xvar, &yvar,
                           IMSLS_SE_CCF, &secc, 2,
                           IMSLS_CROSS_COVARIANCES, &ccv, 0);

  printf ("Mean of series X     = %g\n", xmean);
  printf ("Variance of series X = %g\n\n", xvar);
  printf ("Mean of series Y     = %g\n", ymean);
  printf ("Variance of series Y = %g\n\n", yvar);

  printf ("Lag            CCV           CC         SECC\n\n");
  for (i = 0; i < 2 * lagmax + 1; i++)
    printf ("%-5d%13g%13g%13g\n", i - lagmax, ccv[i], cc[i], secc[i]);
}
```

Output

```
Mean of series X     = -0.0568344
Variance of series X = 1.14694

Mean of series Y     = 53.5091
Variance of series Y = 10.2189

Lag            CCV           CC         SECC

-10      -0.404502    -0.118154     0.162754
-9       -0.508491    -0.148529      0.16247
-8        -0.61437    -0.179456     0.162188
-7       -0.705476    -0.206067     0.161907
-6       -0.776167    -0.226716     0.161627
-5       -0.831474    -0.242871     0.161349
-4       -0.891316    -0.260351     0.161073
-3       -0.980605    -0.286432     0.160798
-2        -1.12477    -0.328542     0.160524
-1        -1.34704    -0.393467     0.160252
0         -1.65853    -0.484451     0.159981
1         -2.04865    -0.598405     0.160252
2         -2.48217    -0.725033     0.160524
3         -2.88541     -0.84282     0.160798
4         -3.16536    -0.924592     0.161073
5         -3.25344    -0.950319     0.161349
6         -3.13113    -0.914593     0.161627
7         -2.83919     -0.82932     0.161907
8         -2.45302    -0.716521     0.162188
9         -2.05269    -0.599584      0.16247
10        -1.69466    -0.495004     0.162754
```

multi_crosscorrelation

Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_multi_crosscorrelation` (*int* `n_observations_x`,
int `n_channel_x`, *float* `x[]`, *int* `n_observations_y`,
int `n_channel_y`, *float* `y[]`, *int* `lagmax`, ..., 0)

The type *double* function is `imsls_d_multi_crosscorrelation`.

Required Arguments

int `n_observations_x` (Input)
Number of observations in each channel of the first time series `x`. `n_observations_x` must be greater than or equal to two.

int `n_channel_x` (Input)
Number of channels in the first time series `x`. `n_channel_x` must be greater than or equal to one.

float `x[]` (Input)
Array of length `n_observations_x` by `n_channel_x` containing the first time series.

int `n_observations_y` (Input)
Number of observations in each channel of the second time series `y`. `n_observations_y` must be greater than or equal to two.

int `n_channel_y` (Input)
Number of channels in the second time series `y`. `n_channel_y` must be greater than or equal to one.

float `y[]` (Input)
Array of length `n_observations_y` by `n_channel_y` containing the second time series.

int `lagmax` (Input)
Maximum lag of cross-covariances and cross-correlations to be computed. `lagmax` must be greater than or equal to one and less than the minimum of `n_observations_x` and `n_observations_y`.

Return Value

Pointer to an array of length `n_channel_x` * `n_channel_y` * (`2` * `lagmax + 1`) containing the cross-correlations between the channels of `x` and `y`. The *m*th element of this array contains the cross-correlation between channel *i* of the `x` series and channel *j* of the `y` series at lag (*k*-`lagmax`) where

i = 1, …, `n_channel_x`

$j = 1, \ldots,$ `n_channel_y`
$k = 0, 1, \ldots, 2*$`lagmax`, and
$m = ($`n_channel_x`$*$`n_channel_y`$*k + (i*$`n_channel_x`$+ j))$

To release this space, use `free`. If no solution can be computed, `NULL` is return.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_multi_crosscorrelation (`*int* `n_observations_x,`
int `n_channel_x,`*float* `x[],`*int* `n_observations_y,`
int `n_channel_y,` *float* `y[],`*int* `lagmax,`
`IMSLS_RETURN_USER,` *float* `crosscorrelations[],`
`IMSLS_PRINT_LEVEL,` *int* `iprint,`
`IMSLS_VARIANCES,`*float* `**x_variance,`*float* `**y_variance,`
`IMSLS_VARIANCES_USER,`*float* `x_variance[],`
float `y_variance[],`
`IMSLS_CROSS_COVARIANCES,`*float* `**cross_covariances,`
`IMSLS_CROSS_COVARIANCES_USER,`
float `cross_covariances[],`
`IMSLS_INPUT_MEANS,`*float* `*x_mean_in,`*float* `*y_mean_in,`
`IMSLS_OUTPUT_MEANS,`*float* `**x_mean_out,`
float `**y_mean_out,`
`IMSLS_OUTPUT_MEANS_USER,`*float* `x_mean_out[],`
float `y_mean_out[],`
0)

Optional Arguments

`IMSLS_RETURN_USER,` *float* `crosscorrelations[]` (Output)
If specified, `crosscorrelations` is a user-specified array of `length n_channel_x * n_channel_y * (2*lagmax + 1)` containing the cross-correlations between the channels of `x` and `y`. See Return Value.

`IMSLS_PRINT_LEVEL,` *int* `iprint` (Input)
Printing option. Default = 0.

iprint	**Action**
0	No printing is performed.
1	Prints the means and variances.
2	Prints the means, variances, and cross-covariances.
3	Prints the means, variances, cross-covariances, and cross-correlations.

`IMSLS_VARIANCES,`*float* `**x_variance,`*float* `**y_variance` (Output)
If specified, `x_variance` is the address of a pointer to an array of length `n_channel_x` containing the variances of the channels of `x` and `y_variance` is the address of a pointer to an array of length `n_channel_y` containing the variances of the channels of `y`.

IMSLS_VARIANCES_USER, *float* x_variance[], *float* y_variance[] (Output)

If specified, x_variance is an array of length n_channel_x containing the variances of the channels of x and y_variance is an array of length n_channel_y containing the variances of the channels of y. See IMSLS_VARIANCES.

IMSLS_CROSS_COVARIANCES, *float* **cross_covariances (Output)

Address of a pointer to an array of length n_channel_x * n_channel_y * (2*lagmax + 1) containing the cross-covariances between the channels of x and y. The *m*th element of this array contains the cross-covariance between channel *i* of the x series and channel *j* of the y series at lag (*k*-lagmax) where

$i = 1, \ldots,$ n_channel_x
$j = 1, \ldots,$ n_channel_y
$k = 0, 1, \ldots, 2*$lagmax, and
$m =$ (n_channel_x*n_channel_y*k + (i*n_channel_x+j)).

IMSLS_CROSS_COVARIANCES_USER, *float* cross_covariances[] (Output)

If specified, cross_covariances is an array of length n_channel_x * n_channel_y * (2*lagmax + 1) containing the cross-covariances between the channels of x and y. See IMSLS_CROSS_COVARIANCES.

IMSLS_INPUT_MEANS, *float* *x_mean_in, *float* *y_mean_in (Input)

If specified, x_mean_in is an array of length n_channel_x containing the user input of the estimate of the means of the channels of x and y_mean_in is an array of length n_channel_y containing the user input of the estimate of the means of the channels of y.

IMSLS_OUTPUT_MEANS, *float* **x_mean_out, *float* **y_mean_out (Output)

If specified, x_mean_out is the address of a pointer to an array of length n_channel_x containing the means of the channels of x and y_mean_out is the address of a pointer to an array of length n_channel_y containing the means of the channels of y.

IMSLS_OUTPUT_MEANS_USER, *float* x_mean_out[], *float* y_mean_out[] (Output)

If specified, x_mean_out is an array of length n_channel_x containing the means of the channels of x and y_mean_out is an array of length n_channel_y containing the means of the channels of y. See IMSLS_OUTPUT_MEANS.

Description

Function imsls_f_multi_crosscorrelation estimates the multichannel cross-correlation function of two mutually stationary multichannel time series. Define the multichannel time series *X* by

$$X = (X_1, X_2, \ldots, X_p)$$

where

$$X_j = (X_{1j}, X_{2j}, \ldots, X_{nj})^T, \quad j = 1, 2, \ldots, p$$

with n = `n_observations_x` and p = `n_channel_x`. Similarly, define the multichannel time series Y by

$$Y = (Y_1, Y_2, \ldots, Y_q)$$

where

$$Y_j = (Y_{1j}, Y_{2j}, \ldots, Y_{mj})^T, \quad j = 1, 2, \ldots, q$$

with m = `n_observations_y` and q = `n_channel_y`. The columns of X and Y correspond to individual channels of multichannel time series and may be examined from a univariate perspective. The rows of X and Y correspond to observations of p-variate and q-variate time series, respectively, and may be examined from a multivariate perspective. Note that an alternative characterization of a multivariate time series X considers the columns to be observations of the multivariate time series while the rows contain univariate time series. For example, see Priestley (1981, page 692) and Fuller (1976, page 14).

Let

$$\hat{\mu}_X = \text{x\_mean}$$

be the row vector containing the means of the channels of X. In particular,

$$\hat{\mu}_X = \left(\hat{\mu}_{X_1}, \hat{\mu}_{X_2}, \ldots, \hat{\mu}_{X_p}\right)$$

where for $j = 1, 2, \ldots, p$

$$\hat{\mu}_{X_j} = \begin{cases} \mu_{X_j} & \mu_{X_j} \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_{tj} & \mu_{X_j} \text{ unknown} \end{cases}$$

Let

$$\hat{\mu}_Y = y\text{\_mean}$$

be similarly defined. The cross-covariance of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\sigma}_{X_iY_j}(k) = \begin{cases} \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = 0, 1, \ldots, K \\ \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = -1, -2, \ldots, -K \end{cases}$$

where $i = 1, \ldots, p, j = 1, \ldots, q$, and K = `lagmax`. The summation on t extends over all possible cross-products with N equal to the number of cross-products in the sum

Let

$$\hat{\sigma}_X(0) = \text{x\_variance}$$

be the row vector consisting of the estimated variances of the channels of X. In particular,

$$\hat{\sigma}_X(0) = (\hat{\sigma}_{X_1}(0), \hat{\sigma}_{X_2}(0), \ldots, \hat{\sigma}_{X_p}(0))$$

where

$$\hat{\sigma}_{X_j}(0) = \frac{1}{n}\sum_{t=1}^{n} X_{tj} - \hat{\mu}_{X_j})^2 \quad j = 1, 2, \ldots, p$$

Let

$$\hat{\sigma}_Y(0) = \text{y\_variance}$$

be similarly defined. The cross-correlation of lag k between channel i of X and channel j of Y is estimated by

$$\hat{\rho}_{X_iY_j}(k) = \frac{\hat{\sigma}_{X_iY_j(k)}}{\left[\hat{\sigma}_{X_i}(0)\hat{\sigma}_{Y_j}(0)\right]^{1/2}} \quad k = 0, \pm 1, \ldots, \pm K$$

Example

Consider the Wolfer Sunspot Data (Y) (Box and Jenkins 1976, page 530) along with data on northern light activity (X_1) and earthquake activity (X_2) (Robinson 1967, page 204) to be a three-channel time series. Function `imsls_f_multi_crosscorrelation` is used to compute the cross-covariances and cross-correlations between X_1 and Y and between X_2 and Y with lags from −`lagmax` = −10 through lag `lagmax` = 10.

```
#include "imsls.h"

void main () {
  int i, lagmax, nobsx, nchanx, nobsy, nchany;
  float x[100 * 2], y[100], *result = NULL, *xvar = NULL, *yvar = NULL,
    *xmean = NULL, *ymean = NULL, *ccv = NULL;
  float data[100][4];
  char line[20];

  nobsx = nobsy = 100;
  nchanx = 2;
  nchany = 1;
  lagmax = 10;

  imsls_f_data_sets (8, IMSLS_X_COL_DIM, 4, IMSLS_RETURN_USER, data, 0);
  for (i = 0; i < 100; i++)
    {
      y[i] = data[i][1];
      x[i * 2] = data[i][2];
      x[i * 2 + 1] = data[i][3];
    }

  result =
    imsls_f_multi_crosscorrelation (nobsx, nchanx, &x[0], nobsy, nchany,
                                &y[0], lagmax, IMSLS_VARIANCES, &xvar,
                                &yvar, IMSLS_OUTPUT_MEANS, &xmean, &ymean,
                                IMSLS_CROSS_COVARIANCES, &ccv, 0);

  imsls_f_write_matrix ("Channel means of x", 1, nchanx, xmean, 0);
  imsls_f_write_matrix ("Channel variances of x", 1, nchanx, xvar, 0);
  imsls_f_write_matrix ("Channel means of y", 1, nchany, ymean, 0);
  imsls_f_write_matrix ("Channel variances of y", 1, nchany, yvar, 0);

  printf ("\nMultichannel cross-covariance between x and y\n");
  for (i = 0; i < (2 * lagmax + 1); i++)
    {
      sprintf (line, "Lag K = %d", i - lagmax);
      imsls_f_write_matrix (line, nchanx, nchany,
                        &ccv[nchanx * nchany * i], 0);
    }

  printf ("\nMultichannel cross-correlation between x and y\n");
  for (i = 0; i < (2 * lagmax + 1); i++)
    {
      sprintf (line, "Lag K = %d", i - lagmax);
      imsls_f_write_matrix (line, nchanx, nchany,
                        &result[nchanx * nchany * i], 0);
    }
}
```

Output

```
   Channel means of x
         1           2
      63.43       97.97

 Channel variances of x
         1           2
       2644        1978

Channel means of y
          46.94

Channel variances of y
             1384

Multichannel cross-covariance between x and y

  Lag K = -10
1       -20.51
2        70.71

  Lag K = -9
1        65.02
2        38.14

  Lag K = -8
1        216.6
2        135.6

  Lag K = -7
```

```
1        246.8
2        100.4

  Lag K = -6
1        142.1
2         45.0

  Lag K = -5
1        50.70
2       -11.81

  Lag K = -4
1        72.68
2        32.69

  Lag K = -3
1        217.9
2        -40.1

  Lag K = -2
1        355.8
2       -152.6

  Lag K = -1
1        579.7
2       -213.0

   Lag K = 0
1        821.6
2       -104.8

   Lag K = 1
1        810.1
2         55.2

   Lag K = 2
1        628.4
2         84.8

   Lag K = 3
1        438.3
2         76.0

   Lag K = 4
1        238.8
2        200.4

   Lag K = 5
1        143.6
2        283.0

   Lag K = 6
1        253.0
2        234.4

   Lag K = 7
1        479.5
2        223.0
```

```
   Lag K = 8
1        724.9
2        124.5

   Lag K = 9
1        925.0
2        -79.5

  Lag K = 10
1        922.8
2       -279.3

Multichannel cross-correlation between x and y

  Lag K = -10
1     -0.01072
2      0.04274

  Lag K = -9
1      0.03400
2      0.02305

  Lag K = -8
1       0.1133
2       0.0819

  Lag K = -7
1       0.1290
2       0.0607

  Lag K = -6
1      0.07431
2      0.02718

  Lag K = -5
1      0.02651
2     -0.00714

  Lag K = -4
1      0.03800
2      0.01976

  Lag K = -3
1       0.1139
2      -0.0242

  Lag K = -2
1       0.1860
2      -0.0923

  Lag K = -1
1       0.3031
2      -0.1287

   Lag K = 0
1       0.4296
2      -0.0633

   Lag K = 1
```

```
1       0.4236
2       0.0333

   Lag K = 2
1       0.3285
2       0.0512

   Lag K = 3
1       0.2291
2       0.0459

   Lag K = 4
1       0.1248
2       0.1211

   Lag K = 5
1       0.0751
2       0.1710

   Lag K = 6
1       0.1323
2       0.1417

   Lag K = 7
1       0.2507
2       0.1348

   Lag K = 8
1       0.3790
2       0.0752

   Lag K = 9
1       0.4836
2      -0.0481

  Lag K = 10
1       0.4825
2      -0.1688
```

partial_autocorrelation

Computes the sample partial autocorrelation function of a stationary time series.

Synopsis

#include `<imsls.h>`

float `*imsls_f_partial_autocorrelation` (*int* `lagmax`, *int* `cf[]`, …, 0)

The type *double* function is `imsls_d_partial_autocorrelation`.

Required Arguments

int `lagmax` (Input)
Maximum lag of partial autocorrelations to be computed.

float `cf[]` (Input)
Array of length `lagmax + 1` containing the autocorrelations of the time series `x`.

Return Value

Pointer to an array of length `lagmax` containing the partial autocorrelations of the time series `x`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_partial_autocorrelation` (*int* `lagmax`, *float* `cf[]`,
`IMSLS_RETURN_USER`, *float* `partial_autocorrelations[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `partial_autocorrelations[]` (Output)
If specified, the partial autocorrelations are stored in an array of length `lagmax` provided by the user.

Description

Function `imsls_f_partial_autocorrelation` estimates the partial autocorrelations of a stationary time series given the K = `lagmax` sample autocorrelations

$$\hat{\rho}(k)$$

for $k = 0, 1, \ldots, K$. Consider the AR(k) process defined by

$$X_t = \phi_{k1}X_{t-1} + \phi_{k2}X_{t-2} + \ldots + \phi_{kk}X_{t-k} + A_t$$

where ϕ_{kj} denotes the j-th coefficient in the process. The set of estimates

$$\left\{\hat{\phi}_{kk}\right\}$$

for $k = 1, \ldots, K$ is the sample partial autocorrelation function. The autoregressive parameters

$$\left\{\hat{\phi}_{kj}\right\}$$

for $j = 1, \ldots, k$ are approximated by Yule-Walker estimates for successive AR(k) models where $k = 1, \ldots, K$. Based on the sample Yule-Walker equations

$$\hat{\rho}(j) = \hat{\phi}_{k1}\hat{\rho}(j-1) + \hat{\phi}_{k2}\hat{\rho}(j-2) + \ldots + \hat{\phi}_{kk}\hat{\rho}(j-k), \quad j = 1, 2, \ldots, k$$

a recursive relationship for $k = 1, \ldots, K$ was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & k = 1 \\ \dfrac{\hat{\rho}(k) - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(k-j)}{1 - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(j)} & k = 2, \ldots, K \end{cases}$$

and

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & k = 1 \\ \dfrac{\hat{\rho}(k) - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(k-j)}{1 - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(j)} & k = 2, \ldots, K \end{cases}$$

This procedure is sensitive to rounding error and should not be used if the parameters are near the nonstationarity boundary. A possible alternative would be to estimate $\{\phi_{kk}\}$ for successive AR(k) models using least or maximum likelihood. Based on the hypothesis that the true process is AR(p), Box and Jenkins (1976, page 65) note

$$\text{var}\{\hat{\phi}_{kk}\} \simeq \frac{1}{n} \quad k \geq p+1$$

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. Routine `imsls_f_partial_autocorrelation` is used to compute the estimated partial autocorrelations.

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

void main()
{
   float *partial=NULL, data[176][2], x[100];
   int i, nobs = 100, lagmax = 20;
   float *ac;

   imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
   for (i=0;i<nobs;i++) x[i] = data[21+i][1];

   ac = imsls_f_autocorrelation(100, x, lagmax, 0);
   partial = imsls_f_partial_autocorrelation(lagmax, ac, 0);
   imsls_f_write_matrix("Lag      PACF", 20, 1, partial, 0);
}
```

Output

```
 Lag      PACF
 1     0.806
 2    -0.635
 3     0.078
 4    -0.059
 5    -0.001
 6     0.172
 7     0.109
 8     0.110
 9     0.079
10     0.079
11     0.069
12    -0.038
13     0.081
14     0.033
15    -0.035
16    -0.131
17    -0.155
18    -0.119
19    -0.016
20    -0.004
```

lack_of_fit

Performs lack-of-fit test for a univariate time series or transfer function given the appropriate correlation function.

Synopsis

#include `<imsls.h>`

    float `imsls_lack_of_fit` (*int* `n_observations`, *float* `cf[]`,
    int `lagmax`, *int* `npfree`, `...`, 0)

Required Arguments

int `n_observations` (Input)
  Number of observations of the stationary time series.

float `cf[]` (Input)
  Array of length `lagmax`+1 containing the correlation function.

int `lagmax` (Input)
  Maximum lag of the correlation function.

int `npfree` (Input)
  Number of free parameters in the formulation of the time series model. `npfree` must be greater than or equal to zero and less than `lagmax`. Woodfield (1990) recommends `npfree = p + q`.

Return Value

Pointer to an array of length 2 with the test statistic, Q, and its *p*-value, *p*. Under the null hypothesis, Q has an approximate chi-squared distribution with `lagmax-lagmin+1-npfree` degrees of freedom.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_lack_of_fit` (*int* `n_observations`, *float* `cf[]`, *int* `lagmax`, *int* `npfree`,
`IMSLS_RETURN_USER`, *float* `stat[]`,
`IMSLS_LAGMIN`, *int* `lagmin`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `stat[]` (Input)
User defined array for storage of lack-of-fit statistics.

`IMSLS_LAGMIN`, *int* `lagmin` (Input)
Minimum lag of the correlation function. `lagmin` corresponds to the lower bound of summation in the lack of fit test statistic. Default value is 1.

Description

Routine `imsls_f_lack_of_fit` may be used to diagnose lack of fit in both ARMA and transfer function models. Typical arguments for these situations are

Model	LAGMIN	LAGMAX	NPFREE
ARMA (*p, q*)	1	$\sqrt{\text{NOBS}}$	$p + q$
Transfer function	0	$\sqrt{\text{NOBS}}$	$r + s$

Function `imsls_f_lack_of_fit` performs a portmanteau lack of fit test for a time series or transfer function containing `n` observations given the appropriate sample correlation function

$$\hat{\rho}(k)$$

for $k = L, L + 1, \ldots, K$ where L = `lagmin` and K = `lagmax`.

The basic form of the test statistic Q is

$$Q = n(n+2)\sum_{k=L}^{K}(n-k)^{-1}\hat{\rho}(k)$$

with $L = 1$ if

$$\hat{\rho}(k)$$

is an autocorrelation function. Given that the model is adequate, Q has a chi-squared distribution with $K - L + 1 - m$ degrees of freedom where $m =$ `npfree` is the number of parameters estimated in the model. If the mean of the time series is estimated, Woodfield (1990) recommends not including this in the count of the parameters estimated in the model. Thus, for an ARMA(p, q) model set `npfree`= $p + q$ regardless of whether the mean is estimated or not. The original derivation for time series models is due to Box and Pierce (1970) with the above modified version discussed by Ljung and Box (1978). The extension of the test to transfer function models is discussed by Box and Jenkins (1976, pages 394–395).

Example

Consider the Wölfer Sunspot Data (Anderson 1971, page 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. An ARMA(2,1) with nonzero mean is fitted using routine `imsls_f_arma` (page 517). The autocorrelations of the residuals are estimated using routine `imsls_f_autocorrelation` (page 541). A portmanteau lack of fit test is computed using 10 lags with `imsls_f_lack_of_fit`.

The warning message from `imsls_f_arma` in the output can be ignored. (See the example for routine `imsls_f_arma` for a full explanation of the warning message.)

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

void main()
{
  int   p = 2;
  int   q = 1;
  int   i;
  int   n_observations = 100;
  int   max_itereations = 0;
  int   lagmin = 1;
  int   lagmax = 10;
  int   npfree = 4;
  float data[176][2], x[100];
  float *parameters;
  float *correlations;
  float *residuals;
  float tolerance = 0.125;
  float *result;

  /* Get sunspot data for 1770 through 1869, store it in x[].      */
  imsls_f_data_sets(2, IMSLS_RETURN_USER, data, 0);
  for (i=0;i<n_observations;i++) x[i] = data[21+i][1];

  /* Get residuals from ARMA(2,1) for autocorrelation/lack of fit  */
  parameters = imsls_f_arma(n_observations, x, p, q,
                            IMSLS_LEAST_SQUARES,
                            IMSLS_CONVERGENCE_TOLERANCE, tolerance,
                              IMSLS_RESIDUAL, &residuals,
                              0);
  /* Get autocorrelations from residuals for lack of fit test      */
  /*     NOTE:  number of OBS is equal to number of residuals      */

correlations = imsls_f_autocorrelation(n_observations-p+lagmax,
   residuals, lagmax,
                                    0);

  /*  Get lack of fit test statistic and p-value                   */
  /*     NOTE:  number of OBS is equal to original number of data  */

   result = imsls_f_lack_of_fit(n_observations,  correlations, lagmax,
  npfree, 0);

  /*  Print parameter estimates, test statistic, and p-value       */
  /*     NOTE: Test Statistic Q follows a Chi-squared dist.        */

 printf("Lack of Fit Statistic,  Q = \t%3.5f\n             P-value of Q
         = \t %1.5f\n\n",result[0], result[1]);

}
```

Output

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

Lack of Fit statistic (Q) =       14.572

         P-value (PVALUE) =       0.9761
```

garch

Computes estimates of the parameters of a GARCH(p,q) model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_garch` (*int* `p`, *int* `q`, *int* `m`, *float* `y[]`, *float* `xguess[]`, …, 0)

The type *double* function is `imsls_d_garch`.

Required Arguments

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

int `q` (Input)
Number of ARCH parameters.

int `m` (Input)
Length of the observed time series.

float `y[]` (Input)
Array of length `m` containing the observed time series data.

float `xguess[]` (Input)
Array of length `p + q + 1` containing the initial values for the parameter array `x[]`.

Return Value

Pointer to the parameter array `x[]` of length `p + q + 1` containing the estimated values of sigma squared, followed by the `q` ARCH parameters, and the `p` GARCH parameters.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_garch` (*int* `p`, *int* `q`, *int* `m`, *float* `y[]`, *float* `xguess[]`,
`IMSLS_MAX_SIGMA`, *float* `max_sigma`,
`IMSLS_A`, *float* `*a`,
`IMSLS_AIC`, *float* `*aic`,
`IMSLS_VAR`, *float* `*var`,
`IMSLS_VAR_USER`, *float* `var[]`,
`IMSLS_VAR_COL_DIM`, *int* `var_col_dim`,
`IMSLS_RETURN_USER`, *float* `x[]`,
0)

Optional Arguments

`IMSLS_MAX_SIGMA`, *float* `max_sigma`, (Input)
Value of the upperbound on the first element (sigma) of the array of returned estimated coefficients. Default = 10.

`IMSLS_A`, *float* `*a`, (Output)
Value of Log-likelihood function evaluated at the estimated parameter array `x`.

`IMSLS_AIC`, *float* `*aic`, (Output)
Value of Akaike Information Criterion evaluated at the estimated parameter array `x`.

`IMSLS_VAR`, *float* `*var`, (Output)
Array of size `(p+q+1)x(p+q+1)` containing the variance-covariance matrix.

`IMSLS_VAR_USER`, *float* `var[]`, (Output)
Storage for array `var` is provided by the user.
See `IMSLS_VAR`.

`IMSLS_VAR_COL_DIM`, *int* `var_col_dim`, (Input)

Column dimension `(p+q+1)` of the variance-covariance matrix.

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

If specified, `x` returns an array of length `p + q + 1` containing the estimated values of sigma squared, followed by the `q` ARCH parameters, and the `p` GARCH parameters. Storage for estimated parameter array `x` is provided by the user.

Description

The Generalized Autoregressive Conditional Heteroskedastic (GARCH) model for a time series $\{w_t\}$ is defined as

$$w_t = z_t \sigma_t$$
$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i w_{t-i}^2,$$

where z_t's are independent and identically distributed standard normal random variables,

$$0 < \sigma^2 < \texttt{max\_sigma},\ \beta_i \geq 0,\ \alpha_i \geq 0 \text{ and}$$
$$\sum_{i=2}^{p+q+1} x(i) = \sum_{i=1}^{p} \beta_i + \sum_{i=1}^{q} \alpha_i < 1.$$

The above model is denoted as GARCH(p,q). The β_i and α_i coeffecients will be referred to as GARCH and ARCH coefficents, respectively. When $\beta_i = 0$, $i = 1,2,\ldots,p$, the above model reduces to ARCH(q) which was proposed by Engle (1982). The nonnegativity conditions on the parameters imply a nonnegative variance and the condition on the sum of the β_i's and α_i's is required for wide sense stationarity.

In the empirical analysis of observed data, GARCH(1,1) or GARCH(1,2) models have often found to appropriately account for conditional heteroskedasticity (Palm 1996). This finding is similar to linear time series analysis based on ARMA models.

It is important to notice that for the above models positive and negative past values have a symmetric impact on the conditional variance. In practice, many series may have strong asymmetric influence on the conditional variance. To take into account this phenomena, Nelson (1991) put forward Exponential GARCH (EGARCH). Lai (1998) proposed and studied some properties of a general class of models that extended linear relationship of the conditional variance in ARCH and GARCH into nonlinear fashion.

The maximum likelihood method is used in estimating the parameters in GARCH(p,q). The log-likelihood of the model for the observed series $\{w_t\}$ with length m = `nobs` is

$$\log(L) = -\frac{m}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{m} y_t^2 / \sigma_t^2 - \frac{1}{2}\sum_{t=1}^{m}\log\sigma_t^2,$$

$$where\ \sigma_t^2 = \sigma^2 + \sum_{i=1}^{p}\beta_i\sigma_{t-i}^2 + \sum_{i=1}^{q}\alpha_i w_{t-i}^2.$$

Thus log(L) is maximized subject to the constraints on the α_i, β_i, and σ.

In this model, if $q = 0$, the GARCH model is singular since the estimated Hessian matrix is singular.

The initial values of the parameter vector `x` entered in vector `xguess` must satisfy certain constraints. The first element of `xguess` refers to σ^2 and must be greater than zero and less than `max_sigma`. The remaining `p+q` initial values must each be greater than or equal to zero and sum to a value less than one.

To guarantee stationarity in model fitting,

$$\sum_{i=2}^{p+q+1} x(i) = \sum_{i=1}^{p}\beta_i + \sum_{i=1}^{q}\alpha_i < 1$$

is checked internally. The initial values should selected from values between zero and one.

`AIC` is computed by

$$- 2 \log (L) + 2(p+q+1),$$

where log(L) is the value of the log-likelihood function.

Statistical inferences can be performed outside the routine `GARCH` based on the output of the log-likelihood function `(A)`, the Akaike Information Criterion (`AIC`), and the variance-covariance matrix `(VAR)`.

Example

The data for this example are generated to follow a GARCH(p,q) process by using a random number generation function `sgarch` . The data set is analyzed and estimates of sigma, the ARCH parameters, and the GARCH parameters are returned. The values of the Log-likelihood function and the Akaike Information Criterion are returned from the optional arguments `IMSLS_A and IMSLS_AIC`.

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

static void  sgarch (int p, int q, int m, float x[],
                float y[], float z[], float y0[], float sigma[]);
#define       M      1000
#define       N      (P + Q + 1)
#define       P      2
#define       Q      1

void main ()
{
    int        n, p, q, m;
    float      a, aic, wk1[M + 1000], wk2[M + 1000],
                wk3[M + 1000], x[N], xguess[N],  y[M];
    float      *result;

    imsls_random_seed_set (182198625);
    m = M;
    p = P;
    q = Q;
    n = p+q+1;
    x[0] = 1.3;
    x[1] = .2;
    x[2] = .3;
    x[3] = .4;
    xguess[0] = 1.0;
    xguess[1] = .1;
    xguess[2] = .2;
    xguess[3] = .3;
    sgarch (p, q, m, x, y, wk1, wk2, wk3);
    result = imsls_f_garch(p, q, m, y, xguess,
                    IMSLS_A, &a,
                    IMSLS_AIC, &aic,
                    0);
    printf("Sigma estimate is\t%11.4f\n", result[0]);
    printf("ARCH(1) estimate is\t%11.4f\n", result[1]);
    printf("GARCH(1) estimate is\t%11.4f\n", result[2]);
    printf("GARCH(2) estimate is\t%11.4f\n", result[3]);
    printf("\nLog-likelihood function value is\t%11.4f\n", a);
    printf("Akaike Information Criterion value is\t%11.4f\n", aic);
    return;
}

static void sgarch (int p, int q, int m, float x[],
                float y[], float z[], float y0[], float sigma[])
{
    int        i, j, l;
    float      s1, s2, s3;

   imsls_f_random_normal ( m + 1000, IMSLS_RETURN_USER, z, 0);

    l = imsls_i_max (p, q);
    l = imsls_i_max (l, 1);
    for (i = 0; i < l; i++) y0[i] = z[i] * x[0];

    /* COMPUTE THE INITIAL VALUE OF SIGMA */
    s3 = 0.0;
    if (imsls_i_max (p, q) >= 1) {
       for (i = 1; i < (p + q + 1); i++) s3 += x[i];
    }
    for (i = 0; i < l; i++) sigma[i] = x[0] / (1.0 - s3);

    for (i = l; i < (m + 1000); i++) {
       s1 = 0.0;
       s2 = 0.0;
       if (q >= 1) {
           for (j = 0; j < q; j++)
             s1 += x[j + 1] * y0[i - j - 1] * y0[i - j - 1];
       }
       if (p >= 1) {
           for (j = 0; j < p; j++)
```

```
            s2 += x[q + 1 + j] * sigma[i - j - 1];
        }
        sigma[i] = x[0] + s1 + s2;
        y0[i] = z[i] * sqrt (sigma[i]);
    }
    /*
     * DISCARD THE FIRST 1000 SIMULATED OBSERVATIONS
     */
    for (i = 0; i < m; i++) y[i] = y0[1000 + i];
    return;
}                         /* end of function */
```

```
Output
Sigma estimate is   1.6480
ARCH(1) estimate is 0.2427
GARCH(1) estimate is      0.3175
GARCH(2) estimate is      0.3335

Log-likelihood function value is -2707.0903
Akaike Information Criterion value is 5422.1807
```

kalman

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

Synopsis

#include `<imsls.h>`

void `imsls_f_kalman` (*int* `nb`, *float* `nb[]`, *float* `covb[]`, *int* `*n`, *float* `*ss`, *float* `*alndet`, ..., 0)

The type *double* function is `imsls_d_kalman`.

Required Arguments

int `nb` (Input)
Number of elements in the state vector.

float `b[]` (Input/Output)
Array of length `nb` containing the estimated state vector. The input is the estimated state vector at time *k* given the observations through time *k* – 1. The output is the estimated state vector at time *k* + 1 given the observations through time *k*. On the first call to `imsls_f_kalman`, the input `b` must be the prior mean of the state vector at time 1.

float `covb[]` (Input/Output)
Array of size `nb` by `nb` such that `covb*` σ^2 is the mean squared error matrix for `b`.
Before the first call to `imsls_f_kalman`, `covb *` σ^2 must equal the variance-covariance matrix of the state vector.

int `*n` (Input/Output)
Pointer to the rank of the variance-covariance matrix for all the observations. `n` must be initialized to zero before the first call to `imsls_f_kalman`. In the usual case when the variance-covariance matrix is nonsingular, `n` equals the sum of the `ny`'s from the invocations to `imsls_f_kalman`. See optional argument `IMSLS_UPDATE` below for the definition of `ny`.

float `*ss` (Input/Output)
Pointer to the generalized sum of squares.
`ss` must be initialized to zero before the first call to `imsls_f_kalman`.
The estimate of σ^2 is given by $\frac{\mathtt{ss}}{\mathtt{n}}$.

float `*alndet` (Input/Output)
Pointer to the natural log of the product of the nonzero eigenvalues of P where $P * \sigma^2$ 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>`

voidt `*imsls_f_random_sample` (*int* `nb`, *float* `nb[]`, *float* `covb[]`,
int `*n`, *float* `*ss`, *float* `*alndet`,
`IMSLS_UPDATE`, *int* `ny`, *float* `*y`, *float* `*z`, *float* `*r`,
`IMSLS_Z_COL_DIM`, *int* `z_col_dim`,
`IMSLS_R_COL_DIM`, *int* `r_col_dim`,
`IMSLS_T`, *float* `*t`,
`IMSLS_T_COL_DIM`, *int* `t_col_dim`,
`IMSLS_Q`, *float* `*q`,
`IMSLS_Q_COL_DIM`, *int* `t_col_dim`,
`IMSLS_TOLERANCE`, *float* `tolerance`,
`IMSLS_V`, *float* `**v`,
`IMSLS_V_USER`, *float* `v[]`,
`IMSLS_COVV`, *float* `**v`,
`IMSLS_COVV_USER`, *float* `v[]`,
0)

Optional Arguments

`IMSLS_UPDATE`, *int* `ny`, *float* `*y`, *float* `*z`, *float* `*r` (Input)
Perform computation of the *update equations.*
`ny`: Number of observations for current update.
`y`: Array of length `ny` containing the observations.

z: ny by nb array containing the matrix relating the observations to the state vector in the observation equation.

r: ny by ny array containing the matrix such that r * σ^2 is the variance-covariance matrix of errors in the observation equation.
σ^2 is a positive unknown scalar. Only elements in the upper triangle of r are referenced.

IMSLS_Z_COL_DIM, *int* z_col_dim (Input)
Column dimension of the matrix z.
Default: z_col_dim = nb

IMSLS_R_COL_DIM, *int* r_col_dim (Input)
Column dimension of the matrix r.
Default: r_col_dim = ny

IMSLS_T, *float* *t (Input)
nb by nb transition matrix in the state equation
Default: t = identity matrix

IMSLS_T_COL_DIM, *int* r_col_dim (Input)
Column dimension of the matrix t.
Default: t_col_dim = nb

IMSLS_Q, *float* *q (Input)
nb by nb matrix such that q * σ^2 is the variance-covariance matrix of the error vector in the state equation.
Default: There is no error term in the state equation.

IMSLS_Q_COL_DIM, *int* q_col_dim (Input)
Column dimension of the matrix q.
Default: q_col_dim = nb

IMSLS_TOLERANCE, *float* tolerance (Input)
Tolerance used in determining linear dependence.
Default: tolerance = 100.0*imsls_f_machine(4)

IMSLS_V, *float* **v (Output)
Address to a pointer v to an array of length ny containing the one-step-ahead prediction error.

IMSLS_V_USER, *float* v[] (Output)
Storage for v is provided by the user. See IMSLS_V.

IMSLS_COVV, *float* **covv (Output)
The address to a pointer of size ny by ny containing a matrix such that covv * σ^2 is the variance-covariance matrix of v.

IMSLS_COVV_USER, *float* covv[] (Output)
Storage for covv is provided by the user. See IMSLS_COVV.

Description

Routine `imsls_f_kalman` is based on a recursive algorithm given by Kalman (1960), which has come to be known as the Kalman filter. The underlying model is known as the state-space model. The model is specified stage by stage where the stages generally correspond to time points at which the observations become available. The routine `imsls_f_kalman` avoids many of the computations and storage requirements that would be necessary if one were to process all the data at the end of each stage in order to estimate the state vector. This is accomplished by using previous computations and retaining in storage only those items essential for processing of future observations.

The notation used here follows that of Sallas and Harville (1981). Let y_k (input in `y` using optional argument `IMSLS_UPDATE`) be the $n_k \times 1$ vector of observations that become available at time *k*. The subscript *k* is used here rather than *t*, which is more customary in time series, to emphasize that the model is expressed in stages $k = 1, 2, \ldots$ and that these stages need not correspond to equally spaced time points. In fact, they need not correspond to time points of any kind. The *observation equation* for the state-space model is

$$y_k = Z_k b_k + e_k \qquad k = 1, 2, \ldots$$

Here, Z_k (input in `z` using optional argument `IMSLS_UPDATE`) is an $n_k \times q$ known matrix and b_k is the $q \times 1$ state vector. The state vector b_k is allowed to change with time in accordance with the *state equation*

$$b_{k+1} = T_{k+1} b_k + w_{k+1} \qquad k = 1, 2, \ldots$$

starting with $b_1 = \mu_1 + w_1$.

The change in the state vector from time *k* to $k + 1$ is explained in part by the *transition matrix* T_{k+1} (the identity matrix by default, or optionally input using `IMSLS_T`), which is assumed known. It is assumed that the *q*-dimensional w_ks ($k = 1, 2, \ldots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 Q_k$, that the n_k-dimensional e_ks ($k = 1, 2, \ldots$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 R_k$, and that the w_ks and e_ks are independent of each other. Here, μ_1 is the mean of b_1 and is assumed known, σ^2 is an unknown positive scalar. Q_{k+1}(input in `Q`) and R_k (input in `R`) are assumed known.

Denote the estimator of the realization of the state vector b_k given the observations $y_1, y_2, \ldots, y_j$ by

$$\hat{\beta}_{k|j}$$

By definition, the mean squared error matrix for

$$\hat{\beta}_{k|j}$$

is

$$\sigma^2 C_{k|j} = E(\hat{\beta}_{k|j} - b_k)(\hat{\beta}_{k|j} - b_k)^T$$

At the time of the *k*-th invocation, we have

$$\hat{\beta}_{k|k-1}$$

and

$C_{k|k-1}$, which were computed from the (*k*–1)-st invocation, input in `b` and `covb`, respectively. During the *k*-th invocation, routine `imsls_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 `v`) is the one-step-ahead prediction error, and $\sigma^2 H_k$ is the variance-covariance matrix for v_k. H_k is stored in `covv`. The “start-up values” needed on the first invocation of `imsls_f_kalman` are

$$\hat{\beta}_{1|0} = \mu_1$$

and $C_{1|0} = Q_1$ input via `b` and `covb`, respectively. Computations for the *k*-th invocation are completed by `imsls_f_kalman` computing the one-step-ahead estimate

$$\hat{\beta}_{k+1|k}$$

along with $C_{k+1|k}$ given by the *prediction equations*:

$$\begin{aligned}\hat{\beta}_{k+1|k} &= T_{k+1} \hat{\beta}_{k|k} \\ C_{k+1|k} &= T_{k+1} C_{k|k} T_{k+1}^T + Q_{k+1}\end{aligned}$$

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 `IMSLS_T` and `IMSLS_Q` to produce

$$\hat{\beta}_{k|k}$$

and $C_{k|k}$, and second without `IMSLS_UPDATE` to produce

$$\hat{\beta}_{k+1|k}$$

and $C_{k+1\,|k}$ (Without `IMSLS_T` and `IMSLS_Q`, the prediction equations are skipped. Without `IMSLS_UPDATE`, the update equations are skipped.).

Often, one desires the estimate of the state vector more than one-step-ahead, i.e., an estimate of

$$\hat{\beta}_{k|j}$$

is needed where $k > j + 1$. At time j, `imsls_f_kalman` is invoked with `IMSLS_UPDATE` to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of `imsls_f_kalman` without `IMSLS_UPDATE` can compute

$$\hat{\beta}_{j+2|j}, \hat{\beta}_{j+3|j}, \ldots, \hat{\beta}_{k|j}$$

Computations for

$$\hat{\beta}_{k|j}$$

and $C_{k|j}$ assume the variance-covariance matrices of the errors in the observation equation and state equation are known up to an unknown positive scalar multiplier, σ^2. The maximum likelihood estimate of σ^2 based on the observations $y_1, y_2, \ldots, y_m$, is given by

$$\hat{\sigma}^2 = SS / N$$

where

$$N = \sum_{k=1}^{m} n_k \text{ and } SS = \sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

N and SS are the input/output arguments `n` and `ss`.

If σ^2 is known, the R_ks and Q_ks can be input as the variance-covariance matrices exactly. The earlier discussion is then simplified by letting $\sigma^2 = 1$.

In practice, the matrices T_k, Q_k, and R_k are generally not completely known. They may be known functions of an unknown parameter vector θ. In this case, `imsls_f_kalman` can be used in conjunction with an optimization program (see routine `imsl_f_min_uncon_multivar`, IMSL C/Math/Library, Chapter 8, “Optimization”) to obtain a maximum likelihood estimate of θ. The natural logarithm of the likelihood function for $y_1, y_2, \ldots, y_m$ differs by no more than an additive constant from

$$L(\theta, \sigma^2; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln \sigma^2$$
$$-\frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)] - \frac{1}{2}\sigma^{-2}\sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

(Harvey 1981, page 14, equation 2.21).

Here,

$$\sum_{k=1}^{m} \ln[\det(H_k)]$$

(stored in `alndet`) is the natural logarithm of the determinant of V where $\sigma^2 V$ is the variance-covariance matrix of the observations.

Minimization of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ over all θ and σ^2 produces maximum likelihood estimates. Equivalently, minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ where

$$L_c(\theta; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \ln\left(\frac{SS}{N}\right) - \frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)]$$

produces maximum likelihood estimates

$$\hat{\theta} \text{ and } \hat{\sigma}^2 = SS / N$$

The minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ instead of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$, reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

$$\hat{\sigma}^2(\theta) = SS(\theta) / N$$

minimizes $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ for all θ, consequently,

$$\hat{\sigma}^2(\theta)$$

can be substituted for σ^2 in $L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ to give a function that differs by no more than an additive constant from $L_c(\theta; y_1, y_2, \ldots, y_m)$.

The earlier discussion assumed H_k to be nonsingular. If H_k is singular, a modification for singular distributions described by Rao (1973, pages 527–528) is used. The necessary changes in the preceding discussion are as follows:

1. Replace

   $$H_k^{-1}$$

   by a generalized inverse.
2. Replace $\det(H_k)$ by the product of the nonzero eigenvalues of H_k.
3. Replace N by

   $$\sum_{k=1}^{m} \operatorname{rank}(H_k)$$

Maximum likelihood estimation of parameters in the Kalman filter is discussed by Sallas and Harville (1988) and Harvey (1981, pages 111–113).

Example 1

Routine `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

$$y_k = b_k + e_k$$
$$b_{k+1} = b_k + w_{k+1} \qquad k = 1,2,3,4$$

where the e_ks are identically and independently distributed normal with mean 0 and variance σ^2, the w_ks are identically and independently distributed normal with mean 0 and variance $4\sigma^2$, and b_1 is distributed normal with mean 4 and variance $16\sigma^2$. Two invocations of `imsls_f_kalman` are needed for each time point in order to compute the filtered estimate and the one-step-ahead estimate. The first invocation does not use the optional arguments `IMSLS_T` and `IMSLS_Q` so that the prediction equations are skipped in the computations. The update equations are skipped in the computations in the second invocation.

This example also computes the one-step-ahead prediction errors. Harvey (1981, page 117) contains a misprint for the value v_4 that he gives as 1.197. The correct value of v_4 = 1.003 is computed by `imsls_f_kalman`.

.

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

#define NB 1
#define NOBS 4
#define NY 1

void main()
{
    int         nb = NB, nobs = NOBS, ny = NY;
    int         ldcovb, ldcovv, ldq, ldr, ldt, ldz;
    int         i, iq, it, n, nout;
    float       alndet, b[NB], covb[NB][NB], covv[NY][NY],
                q[NB][NB], r[NY][NY], ss,
                t[NB][NB], tol, v[NY], y[NY], z[NY][NB];
    float       ydata[] = {4.4, 4.0, 3.5, 4.6};

    z[0][0] = 1.0;
    r[0][0] = 1.0;
    q[0][0] = 4.0;
    t[0][0] = 1.0;
    b[0] = 4.0;
    covb[0][0] = 16.0;

    /* Initialize arguments for initial call to imsls_f_kalman. */
    n = 0;
    ss = 0.0;
    alndet = 0.0;
    printf("k/j      b       covb n     ss      alndet     v       covv\n");

    for (i = 0; i < nobs; i++) {
      /* Update */
      y[0] = ydata[i];
      imsls_f_kalman(nb, b, (float*)covb, &n, &ss, &alndet,
                    IMSLS_UPDATE, ny, y, z, r,
                    IMSLS_V_USER, v,
                    IMSLS_COVV_USER, covv,
                    0);
```

```
        printf("%d/%d %8.3f %8.3f %d %8.3f %8.3f %8.3f %8.3f\n",
              i, i, b[0], covb[0][0], n, ss, alndet, v[0], covv[0][0]);

        /* Prediction */
        imsls_f_kalman(nb, b, (float*)covb, &n, &ss, &alndet,
                    IMSLS_T, t,
                    IMSLS_Q, q,
                    0);

        printf("%d/%d %8.3f %8.3f %d %8.3f %8.3f %8.3f %8.3f\n",
              i+1, i, b[0], covb[0][0], n, ss, alndet, v[0], covv[0][0]);
    }

}
```

Output

```
k/j      b      covb n     ss      alndet     v       covv
0/0    4.376    0.941 1    0.009    2.833    0.400   17.000
1/0    4.376    4.941 1    0.009    2.833    0.400   17.000
1/1    4.063    0.832 2    0.033    4.615   -0.376    5.941
2/1    4.063    4.832 2    0.033    4.615   -0.376    5.941
2/2    3.597    0.829 3    0.088    6.378   -0.563    5.832
3/2    3.597    4.829 3    0.088    6.378   -0.563    5.832
3/3    4.428    0.828 4    0.260    8.141    1.003    5.829
4/3    4.428    4.828 4    0.260    8.141    1.003    5.829
```

Example 2

Routine `imsls_f_kalman` is used with routine `imsl_f_min_uncon_multivar`, (see IMSL C/Math/Library, Chapter 8, "Optimization") to find a maximum likelihood estimate of the parameter θ in a MA(1) time series represented by $y_k = \varepsilon_k - \theta\varepsilon_{k-1}$. Routine `imsls_f_random_arma` (see IMSL C/Stat/Library, Chapter 12, "Random Number Generation") is used to generate 200 random observations from an MA(1) time series with θ = 0.5 and $\sigma^2 = 1$.

The MA(1) time series is cast as a state-space model of the following form (see Harvey 1981, pages 103–104, 112):

$$y_k = \begin{pmatrix} 1 & 0 \end{pmatrix} b_k$$

$$b_k = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} b_{k-1} + w_k$$

where the two-dimensional w_ks are independently distributed bivariate normal with mean 0 and variance $\sigma^2 Q_k$ and

$$Q_1 = \begin{pmatrix} 1+\theta^2 & -\theta \\ -\theta & \theta^2 \end{pmatrix}$$

$$Q_k = \begin{pmatrix} 1 & -\theta \\ -\theta & \theta^2 \end{pmatrix} \qquad k = 2, 3, ..., 200$$

The warning error that is printed as part of the output is not serious and indicates that `imsl_f_min_uncon_multivar` is generally used for multi-parameter minimization.

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

#define NOBS 200
#define NTHETA 1
#define NB 2
#define NY 1

float fcn(int ntheta, float theta[]);
float *ydata;
void main ()
{
    int  lagma[1];
    float pma[1];
    float *theta;

    imsls_random_seed_set(123457);
    pma[0] = 0.5;
    lagma[0] = 1;
    ydata = imsls_f_random_arma(200, 0, NULL, 1, pma,
                              IMSLS_ACCEPT_REJECT_METHOD,
                              IMSLS_NONZERO_MALAGS, lagma,
                              0);

    theta = imsl_f_min_uncon_multivar(fcn, NTHETA, 0);

    printf("* * * Final Estimate for THETA * * *\n");
    printf("Maximum likelihood estimate, THETA = %f\n", theta[0]);

}

float fcn(int ntheta, float theta[])
{
  int i, n;
  float res, ss, alndet;
  float t[] = {0.0, 1.0, 0.0, 0.0};
  float z[] = {1.0, 0.0};
  float q[NB][NB], r[NY][NY], b[NB], covb[NB][NB], y[NY];
  if (fabs(theta[0]) > 1.0) {
    res = 1.0e10;
  } else {
    q[0][0] = 1.0;
    q[0][1] = -theta[0];
    q[1][0] = -theta[0];
    q[1][1] = theta[0]*theta[0];

    r[0][0] = 0.0;

    b[0] = 0.0;
    b[1] = 0.0;

    covb[0][0] = 1.0 + theta[0]*theta[0];
    covb[0][1] = -theta[0];
    covb[1][0] = -theta[0];
    covb[1][1] = theta[0]*theta[0];

    n = 0;
    ss = 0.0;
    alndet = 0.0;

    for (i = 0; i<NOBS; i++) {
      y[0] = ydata[i];
      imsls_f_kalman(NB, b, (float*)covb, &n, &ss, &alndet,
                  IMSLS_UPDATE, NY, y, z, r,
                  IMSLS_Q, q,
                  IMSLS_T, t,
                  0);
    }
    res = n*log(ss/n) + alndet;
  }
  return(res);
}
```

Output

```
*** WARNING_IMMEDIATE Error from imsl_f_min_uncon_multivar.  This routine
***          may be inefficient for a problem of size "n" = 1.


*** WARNING_IMMEDIATE Error from imsl_f_min_uncon_multivar.  The last global
***          step failed to locate a lower point than the current X value.
***          The current X may be an approximate local minimizer and no more
***          accuracy is possible or the step tolerance may be too large
***          where "step_tol" = 2.422181e-05 is given.

* * * Final Estimate for THETA * * *
Maximum likelihood estimate, THETA = 0.453256
```

Chapter 9: Multivariate Analysis

Routines

Usage Notes

Cluster Analysis

Function `imsls_f_cluster_k_means` performs a *K*-means cluster analysis. Basic *K*-means clustering attempts to find a clustering that minimizes the within-cluster sums-of-squares. In this method of clustering the data, matrix *X* is grouped so that each observation (row in *X*) is assigned to one of a fixed number, *K*, of clusters. The sum of the squared difference of each observation about its assigned cluster's mean is used as the criterion for assignment. In the basic algorithm, observations are transferred from one cluster or another when doing so decreases the within-cluster sums-of-squared differences. When no transfer occurs in a pass through the entire data set, the algorithm stops. Function `imsls_f_cluster_k_means` is one implementation of the basic algorithm.

The usual course of events in *K*-means cluster analysis is to use `imsls_f_cluster_k_means` to obtain the optimal clustering. The clustering is then evaluated by functions described in Chapter 1, "Basic Statistics," and/or other chapters in this manual. Often, K-means clustering with more than one value of K is performed, and the value of K that best fits the data is used.

Clustering can be performed either on observations or variables. The discussion of the function `imsls_f_cluster_k_means` assumes the clustering is to be performed on the observations, which correspond to the rows of the input data matrix. If variables, rather than observations, are to be clustered, the data matrix should first be transposed. In the documentation for `imsls_f_cluster_k_means`, the words “observation” and “variable” are interchangeable.

Principal Components

The idea in principal components is to find a small number of linear combinations of the original variables that maximize the variance accounted for in the original data. This amounts to an eigensystem analysis of the covariance (or correlation) matrix. In addition to the eigensystem analysis, `imsls_f_principal_components` computes standard errors for the eigenvalues. Correlations of the original variables with the principal component scores also are computed.

Factor Analysis

Factor analysis and principal component analysis, while quite different in assumptions, often serve the same ends. Unlike principal components in which linear combinations yielding the highest possible variances are obtained, factor analysis generally obtains linear combinations of the observed variables according to a model relating the observed variable to hypothesized underlying factors, plus a random error term called the unique error or uniqueness. In factor analysis, the unique errors associated with each variable are usually assumed to be independent of the factors. Additionally, in the common factor model, the unique errors are assumed to be mutually independent. The factor analysis model is expressed in the following equation:

$$x - \mu = \Lambda f + e$$

where x is the p vector of observed values, μ is the p vector of variable means, Λ is the $p \times k$ matrix of factor loadings, f is the k vector of hypothesized underlying random factors, e is the p vector of hypothesized unique random errors, p is the number of variables in the observed variables, and k is the number of factors.

Because much of the computation in factor analysis was originally done by hand or was expensive on early computers, quick (but dirty) algorithms that made the calculations possible were developed. One result is the many factor extraction methods available today. Generally speaking, in the exploratory or model building phase of a factor analysis, a method of factor extraction that is not computationally intensive (such as principal components, principal factor, or image analysis) is used. If desired, a computationally intensive method is then used to obtain the final factors.

In exploratory factor analysis, the unrotated factor loadings obtained from the factor extraction are generally transformed (rotated) to simplify the interpretation of the factors. Rotation is possible because of the overparameterization in the factor analysis model. The method used for rotation may result in factors that are independent (orthogonal rotations) or correlated (oblique rotations). Prior information may be available (or hypothesized) in which case a Procrustes rotation could be used. When no prior information is available, an analytic rotation can be performed.

The steps generally used in a factor analysis are summarized as follows:

Steps in a Factor Analysis

Step 1

Calculate Covariance (Correlation) Matrix
IMSL routine `imsls_f_covariances`
(see Chapter 3, “Correlation and Covariance”)

Step 2

Initial Factor Extraction

`imsls_f_factor_analysis`, page 609

Step 3

Factor Rotation
using `imsls_f_factor_analysis`’ optional arguments

Orthogonal	**Oblique**
No Prior Info. `IMSLS_ORTHOMAX_ROTATION`, page 610	No Prior Info. `IMSLS_OBLIQUE_PROMAX_ROTATION`, page 610 `IMSLS_DIRECT_OBLIMIN_ROTATION`, page 610 `IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`, page 610
Prior Info. `IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION`, page 610	Prior Info. `IMSLS_OBLIQUE_PROCRUSTES_ROTATION`, page 610

Step 4

Factor Structure and Variance
`imsls_f_factor_analysis` optional argument `IMSLS_FACTOR_STRUCTURE`, page 610

dissimilarities

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_dissimilarities` (*int* `nrow`, *int* `ncol`, *float* `*x`, …, 0)

The type *double* function is `imsls_d_dissimilarities`.

Required Arguments

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

int `ncol` (Input)
Number of columns in the matrix.

float `*x` (Input)
Array of size `nrow` by `ncol` containing the matrix.

Return Value

An array of size *m* by *m* containing the computed dissimilarities or similarities, where *m* = `nrow` if optional argument `IMSLS_ROWS` is used, and *m* = `ncol` otherwise.

Synopsis with Optional Arugments

#include `<imsls.h>`

float `*imsls_f_dissimilarities` (*int* `nrow`, *int* `ncol`, *float* `*x`,
`IMSLS_ROWS`, or `IMSLS_COLUMNS`,
`IMSLS_INDEX`, *int* `ndstm`, *int* `ind[]`,
`IMSLS_METHOD`, *int* `imeth`,
`IMSLS_SCALE`, *int* `iscale`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,
`IMSLS_RETURN_USER`, *float* `dist[]`,
0)

Optional Arguments

`IMSLS_ROWS`,
or

`IMSLS_COLUMNS`, (Input)
Exactly one of these options can be present to indicate whether distances are computed between rows or columns of `x`.
Default: Distances are computed between rows.

`IMSLS_INDEX`, *int* `ndstm`, *int* `ind[]`, (Input)
Argument `ind` is an array of length `ndstm` containing the indices of the rows (columns if `IMSLS_ROWS` is used) to be used in computing the distance measure.
Default: All rows(columns) are used.

`IMSLS_METHOD`, *int* `imeth` (Input)
Method to be used in computing the dissimilarities or similarities.
Default: `imeth` = 0.

imeth	**Method**
0	Euclidean distance (L_2 norm)
1	Sum of the absolute differences (L_1 norm)
2	Maximum difference (L_∞ norm)
3	Mahalanobis distance
4	Absolute value of the cosine of the angle between the vectors
5	Angle in radians (0, π) between the lines through the origin defined by the vectors
6	Correlation coefficient
7	Absolute value of the correlation coefficient
8	Number of exact matches

See the Description section for a more detailed description of each measure.

`IMSLS_SCALE`, *int* iscale (Input)
Scaling option. (Input)
`iscale` is not used for methods 3 through 8.
Default: `iscale` = 0.

iscale	**Scaling Performed**
0	No scaling is performed.
1	Scale each column (row, if `IMSLS_ROWS` is used) by the standard deviation of the column (row).

iscale	**Scaling Performed**
2	Scale each column (row, if `IMSLS_ROWS` is used) by the range of the column (row).

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

`IMSLS_RETURN_USER`, *float* `dist[]` (Output)
User allocated array of size m by m containing the computed dissimilarities or similarities, where m = `nrow` if `IMSLS_ROWS` is used, and m = `ncol` otherwise.

Description

Function `imsls_f_dissimilarities` computes an upper triangular matrix (excluding the diagonal) of dissimilarities (or similarities) between the columns or rows of a matrix. Nine different distance measures can be computed. For the first three measures, three different scaling options can be employed. Output from `imsls_f_dissimilarities` is generally used as input to clustering or multidimensional scaling functions.

The following discussion assumes that the distance measure is being computed between the columns of the matrix, i.e., that `IMSLS_COLUMNS` is used. If distances between the rows of the matrix are desired, use optional argument `IMSLS_ROWS`.

For `imeth` = 0 to 2, each row of `x` is first scaled according to the value of `iscale`. The scaling parameters are obtained from the values in the row scaled as either the standard deviation of the row or the row range; the standard deviation is computed from the unbiased estimate of the variance. If `iscale` is 0, no scaling is performed, and the parameters in the following discussion are all 1.0. Once the scaling value (if any) has been computed, the distance between column i and column j is computed via the difference vector $z_k = (x_k - y_k)/s_k$, $i = 1, \ldots,$ `ndstm`, where x_k denotes the k-th element in the i-th column, and y_k denotes the corresponding element in the j-th column. For given z_i, the metrics 0 to 2 are defined as:

imeth		**Metric**
0	$\sqrt{\left(\sum_{i=1}^{\text{ndstm}} z_i^2\right)}$	Euclidean distance
1	$\sum_{i=1}^{\text{ndstm}} \lvert z_i \rvert$	L_1 norm
2	$\max_i \lvert z_i \rvert$	L_∞ norm

Distance measures corresponding to `imeth` = 3 to 8 do not allow for scaling. These measures are defined via the column vectors $X = (x_i)$, $Y = (y_i)$, and $Z = (x_i - y_i)$ as follows:

iscale	**Scaling Performed**
3	$Z'\hat{\Sigma}^{-1}Z =$ Mahalanobis distance, where $\hat{\Sigma}$ is the usual unbiased sample estimate of the covariance matrix of the rows.
4	$\cos(\theta) = X^TY / \left(\sqrt{X^TX}\sqrt{Y^TY}\right) =$ the dot product of X and Y divided by the length of X times the length of Y.
5	θ, where θ is defined in 4.
6	ρ = the usual (centered) estimate of the correlation between X and Y.
7	The absolute value of ρ (where ρ is defined in 6).
8	The number of times $x_i = y_i$, where x_i and y_i are elements of X and Y.

For the Mahalanobis distance, any variable used in computing the distance measure that is (numerically) linearly dependent upon the previous variables in the `ind` vector is omitted from the distance measure.

Example

The following example illustrates the use of `imsls_f_dissimilarities` for computing the Euclidean distance between the rows of a matrix.

```
#include "imsls.h"

void main()
{
  int ncol=2, nrow = 4;
  float x [4][2] = {1., 1.,
                    1., 0.,
                    1.,-1.,
                    1., 2.};
  float *dist;

  dist = imsls_f_dissimilarities(nrow, ncol, (float*)x, 0);
  imsls_f_write_matrix("dist", 4, 4, dist, 0);
}
```

Output

```
                   dist
          1          2          3          4
1         0          1          2          1
2         0          0          1          2
3         0          0          0          3
4         0          0          0          0
```

cluster_hierarchical

Performs a hierarchical cluster analysis given a distance matrix.

Synopsis

#include `<imsls.h>`

void `imsls_f_cluster_hierarchical` (*int* `npt`, *float* `*dist`, …, 0)

The type *double* function is `imsls_d_cluster_hierarchical`.

Required Arguments

int `npt` (Input)
Number of data points to be clustered.

float `*dist` (Input/Ouput)
An `npt` by `npt` symmetric matrix containing the distance (or similarity) matrix.
`dist` is a symmetric matrix. On input, only the upper triangular part needs to be present. The function `imsls_f_cluster_hierarchical` saves the upper triangular part of `dist` in the lower triangle. On return from `imsls_f_cluster_hierarchical`, the upper triangular part of `dist` is restored, and the matrix is made symmetric.

Synopsis with Optional Arugments

#include `<imsls.h>`

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.
Default: `imeth` = 0.

`Imeth`	**Method**
0	Single linkage (minimum distance)
1	Complete linkage (maximum distance)
2	Average distance within (average distance between objects within the merged cluster)
3	Average distance between (average distance between objects in the two clusters)
4	Ward's method (minimize the within-cluster sums of squares). For Ward's method, the elements of `dist` are assumed to be Euclidean distances.

`IMSLS_TRANSFORMATION`, *int* itrans (Input)
Option giving the method to be used for clustering.
Default: `itrans` = 0.

`Imeth`	**Method**
0	No transformation is required. The elements of `dist` are distances.
1	Convert similarities to distances by multiplication by −1.0.
2	Convert similarities (usually correlations) to distances by taking the reciprocal of the absolute value.

`IMSLS_CLUSTERS`, *float* **`clevel`, *int* **`iclson`, *int* **`icrson` (Output)
Argument `clevel` is the address of an array of length `npt` − 1 containing the level at which the clusters are joined. `clevel`[*k*-1] contains the distance (or similarity) level at which cluster `npt` + *k* was formed. If the original data in `dist` was transformed via the optional argument `IMSLS_TRANSFORMATION`, the inverse transformation is applied to the values in `clevel` prior to exit from `imsls_f_cluster_hierarchical`. Argument `iclson` is the address of an array of length `npt` − 1 containing the left sons of each merged cluster. Argument `icrson` is the address of an array of length `npt` − 1 containing the right sons of each merged cluster. Cluster `npt` + *k* is formed by merging clusters `iclson`[*k*-1] and `icrson`[*k*-1].

`IMSLS_CLUSTERS_USER`, *float* `clevel[]`, *int* `iclson[]`, *int* `icrson[]` (Output)
Storage for arrays `clevel`, `iclson`, and `icrson` is provided by the user. See `IMSLS_CLUSTERS`.

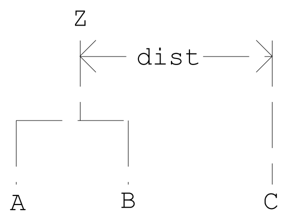
Description

Function `imsls_f_cluster_hierarchical` conducts a hierarchical cluster analysis based upon the distance matrix, or by appropriate use of the `IMSLS_TRANSFORMATION` optional argument, based upon a similarity matrix. Only the upper triangular part of the matrix `dist` is required as input to `imsls_f_cluster_hierarchical`.

Hierarchical clustering in `imsls_f_cluster_hierarchical` proceeds as follows. Initially, each data point is considered to be a cluster, numbered 1 to n = `npt`.

1. If the data matrix contains similarities, they are converted to distances by the method specified by `IMSLS_TRANSFORMATION`. Set $k = 1$.
2. A search is made of the distance matrix to find the two closest clusters. These clusters are merged to form a new cluster, numbered $n + k$. The cluster numbers of the two clusters joined at this stage are saved in `icrson` and `iclson`, and the distance measure between the two clusters is stored in `clevel`.
3. Based upon the method of clustering, updating of the distance measure in the row and column of `dist` corresponding to the new cluster is performed.
4. Set $k = k + 1$. If $k < n$, go to Step 2.

The five methods differ primarily in how the distance matrix is updated after two clusters have been joined. The `IMSLS_METHOD` optional argument specifies how the distance of the cluster just merged with each of the remaining clusters will be updated. Function `imsls_f_cluster_hierarchical` allows five methods for computing the distances. To understand these measures, suppose in the following discussion that clusters "*A*" and "*B*" have just been joined to form cluster "*Z*", and interest is in computing the distance of *Z* with another cluster called "*C*".



Imeth	**Method**
0	Single linkage method. The distance from *Z* to *C* is the minimum of the distances (*A* to *C*, *B* to *C*).
1	Complete linkage method. The distance from *Z* to *C* is the maximum of the distances (*A* to *C*, *B* to *C*).
2	Average-distance-within-clusters method. The distance from *Z* to *C* is the average distance of all

`Imeth`	**Method**
	objects that would be within the cluster formed by merging clusters *Z* and *C*. This average may be computed according to formulas given by Anderberg (1973, page 139).
3	Average-distance-between-clusters method. The distance from *Z* to *C* is the average distance of objects within cluster *Z* to objects within cluster *C*. This average may be computed according to methods given by Anderberg (1973, page 140).
4	Ward's method. Clusters are formed so as to minimize the increase in the within-cluster sums of squares. The distance between two clusters is the increase in these sums of squares if the two clusters were merged. A method for computing this distance from a squared Euclidean distance matrix is given by Anderberg (1973, pages 142–145).

In general, single linkage will yield long thin clusters while complete linkage will yield clusters that are more spherical. Average linkage and Ward's linkage tend to yield clusters that are similar to those obtained with complete linkage.

Function `imsls_f_cluster_hierarchical` produces a unique representation of the binary cluster tree via the following three conventions; the fact that the tree is unique should aid in interpreting the clusters. First, when two clusters are joined and each cluster contains two or more data points, the cluster that was initially formed with the smallest level (in `clevel`) becomes the left son. Second, when a cluster containing more than one data point is joined with a cluster containing a single data point, the cluster with the single data point becomes the right son. Finally, when two clusters containing only one object are joined, the cluster with the smallest cluster number becomes the right son.

Comments

1. The clusters corresponding to the original data points are numbered from 1 to `npt`. The `npt` – 1 clusters formed by merging clusters are numbered `npt` + 1 to `npt` + (`npt` – 1).
2. Raw correlations, if used as similarities, should be made positive and transformed to a distance measure. One such transformation can be performed by specifying optional argument `IMSLS_TRANSFORMATION`, with `itrans` = 2 in `imsls_f_cluster_hierarchical`.
3. The user may cluster either variables or observations in `imsls_f_cluster_hierarchical` since a dissimilarity matrix, not the original data, is used. Function `imsls_f_dissimilarities`

(page 586) 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 14, 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` (page 586) computes the distance matrix (`dist`) prior to calling `imsls_f_cluster_hierarchical`.

```
#include "imsls.h"

void main()
{
  int  iscale=1, ncol=5, nrow=150, nvar=4, npt = 150;
  int i, iclson[149], icrson[149], ind[4] = {1, 2, 3, 4};
  float clevel[149], *dist, *x;

  x = imsls_f_data_sets(3, 0);

  dist = imsls_f_dissimilarities(nrow, ncol, x,
                             IMSLS_INDEX, nvar, ind,
                             IMSLS_SCALE, iscale,
                             0);
  imsls_f_cluster_hierarchical(npt, dist,
              IMSLS_CLUSTERS_USER, clevel, iclson, icrson,
              IMSLS_METHOD, 2,
              0);

  for (i=0;i<149;i+=15) printf("%6.2f\t", clevel[i]);
  printf("\n");
  for (i=0;i<149;i+=15) printf("%6d\t", iclson[i]);
  printf("\n");
  for (i=0;i<149;i+=15) printf("%6d\t", icrson[i]);
  printf("\n");
}
```

Output

```
  0.00     0.17     0.23     0.27     0.31     0.37     0.41     0.48     0.60     0.78
   143      153       17      140       53      198      186      218      261      249
   102       29        6      113       51       91      212      243      266      262
```

cluster_number

Computes cluster membership for a hierarchical cluster tree.

Synopsis

#include `<imsls.h>`

int *`imsls_cluster_number` (*int* `npt`, *int* `*iclson`, *int* `*icrson`, *int* `k`, …, 0)

Required Arguments

int `npt` (Input)
Number of data points to be clustered.

int `*iclson` (Input)
Vector of length `npt` – 1 containing the left son cluster numbers. Cluster `npt` + `i` is formed by merging clusters `iclson[i-1]` and `icrson[i-1]`.

int `*icrson` (Input)
Vector of length `npt` – 1 containing the left son cluster numbers. Cluster `npt` + `i` is formed by merging clusters `iclson[i-1]` and `icrson[i-1]`.

int `k` (Input)
Desired number of clusters.

Return Value

Vector of length `npt` containing the cluster membership of each observation.

Synopsis with Optional Arugments

#include `<imsls.h>`

int `*imsls_cluster_number` (*int* `npt`, *int* `*iclson`, *int* `*icrson`, *int* `k`,
`IMSLS_OBS_PER_CLUSTERS`, *int* `**nclus`,
`IMSLS_OBS_PER_CLUSTERS_USER`, *int* `nclus[]`,
`IMSLS_RETURN_USER`, *int* `iclus[]`,
0)

Optional Arguments

`IMSLS_OBS_PER_CLUSTERS`, *int* `**nclus` (Output)
Address of a pointer to an internally allocated array of length `k` containing the number of observations in each cluster.

`IMSLS_OBS_PER_CLUSTERS_USER`, *int* `nclus[]` (Output)
Storage for array `nclus` is provided by the user. See `IMSLS_OBS_PER_CLUSTERS`.

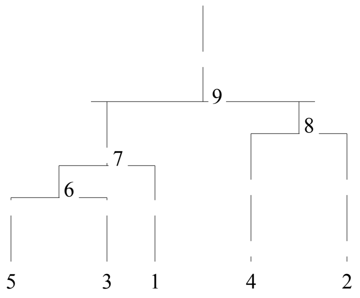
`IMSLS_RETURN_USER`, *float* `iclus[]` (Output)
User allocated array of length `npt` containing the cluster membership of each observation.

Description

Given a fixed number of clusters (K) and the cluster tree (vectors `icrson` and `iclson`) produced by the hierarchical clustering algorithm (see function imsls_f_cluster_hierarchical, page 590), function imsls_cluster_number determines the cluster membership of each observation. The function imsls_cluster_number first determines the root nodes for the *K* distinct subtrees forming the *K* clusters and then traverses each subtree to determine the cluster membership of each observation. The function imsls_cluster_number also returns the number of observations found in each cluster.

Example 1

In the following example, cluster membership for *K* = 2 clusters is found for the displayed cluster tree. The output vector iclus contains the cluster numbers for each observation.



```
#include "imsls.h"

void main()
{
  int  k = 2, npt = 5, *iclus;
  int iclson[] = {5, 6, 4, 7};
  int icrson[] = {3, 1, 2, 8};

  iclus = imsls_cluster_number(npt, iclson, icrson, k, 0);
  imsls_i_write_matrix("iclus", 1, 5, iclus, 0);
}
```

Output

```
        iclus
 1    2    3    4    5
 1    2    1    2    1
```

Example 2

This example illustrates the typical usage of imsls_cluster_number. The Fisher iris data (see function imsls_f_data_sets, see Chapter 14, Utilities) is clustered. First the distance between the irises are computed using function imsls_f_dissimilarities (page 586). The resulting distance matrix is then clustered using function imsls_f_cluster_hierarchical (page 590). 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"

void main()
{
  int ncol = 5, nrow = 150, nvar = 4, npt = 150, k = 5;
  int i, j, *iclson, *icrson, *iclus, *nclus;
  int ind[4] = {1, 2, 3, 4};
  float *clevel, dist[150][150], *x, f_rand;
  int *p_iclus = NULL, *p_nclus = NULL;

  x = imsls_f_data_sets (3, 0);
  imsls_f_dissimilarities(nrow, ncol, x,
                      IMSLS_INDEX, nvar, ind,
                      IMSLS_RETURN_USER, dist,
                      0);

  imsls_random_seed_set (4);
  for (i = 0; i < npt; i++)
    {
      for (j = i + 1; j < npt; j++)
       {
         imsls_f_random_uniform (1, IMSLS_RETURN_USER, &f_rand, 0);
         dist[i][j] = MAX (0.0, dist[i][j] + .001 * f_rand);
         dist[j][i] = dist[i][j];
       }
      dist[i][i] = 0.;
    }
  imsls_f_cluster_hierarchical (npt, (float*)dist,
              IMSLS_CLUSTERS, &clevel, &iclson, &icrson,
              0);

  iclus = imsls_cluster_number (npt, iclson, icrson, k,
                           IMSLS_OBS_PER_CLUSTER, &nclus,
                           0);

  imsls_i_write_matrix ("iclus", 25, 5, iclus, 0);
  imsls_i_write_matrix ("nclus", 1, 5, nclus, 0); }
```

Output

```
         iclus
     1    2    3    4    5
 1   5    5    5    5    5
 2   5    5    5    5    5
 3   5    5    5    5    5
 4   5    5    5    5    5
 5   5    5    5    5    5
 6   5    5    5    5    5
 7   5    5    5    5    5
 8   5    5    5    5    5
 9   5    5    5    5    5
10   5    5    5    5    5
11   2    2    2    2    2
```

```
12   2   2   1   2   2
13   1   2   2   2   2
14   2   2   2   2   2
15   2   2   2   2   2
16   2   2   2   2   2
17   2   2   2   2   2
18   2   2   2   2   2
19   2   2   2   1   2
20   2   2   2   1   2
21   2   2   2   2   2
22   2   3   2   2   2
23   2   2   2   2   2
24   2   2   4   2   2
25   2   2   2   2   2

          nclus
  1    2    3    4    5
  4   93    1    2   50
```

cluster_k_means

Performs a *K*-means (centroid) cluster analysis.

Synopsis

#include `<imsls.h>`

int `*imsls_f_cluster_k_means` (*int* `n_observations`,
int `n_variables`, *float* `x[]`, *int* `n_clusters`,
float `cluster_seeds`, ..., 0)

The type *double* function is `imsls_d_cluster_k_means`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `n_variables` (Input)
Number of variables to be used in computing the metric.

float `x[]` (Input)
Array of length `n_observations` × `n_variables` containing the observations to be clustered.

int `n_clusters` (Input)
Number of clusters.

float `cluster_seeds[]` (Input)
Array of length `n_clusters` × `n_variables` containing the cluster seeds, i.e., estimates for the cluster centers.

Return Value

The cluster membership for each observation is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

int *imsls_f_cluster_k_means (int n_observations,
        int n_variables, float x[], int n_clusters,
        float cluster_seeds,
        IMSLS_WEIGHTS, float weights[],
        IMSLS_FREQUENCIES, float frequencies[],
        IMSLS_MAX_ITERATIONS, int max_iterations,
        IMSLS_CLUSTER_MEANS, float **cluster_means,
        IMSLS_CLUSTER_MEANS_USER, float cluster_means[],
        IMSLS_CLUSTER_SSQ, float **cluster_ssq,
        IMSLS_CLUSTER_SSQ_USER, float cluster_ssq[],
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_CLUSTER_MEANS_COL_DIM,
                int cluster_means_col_dim,
        IMSLS_CLUSTER_SEEDS_COL_DIM,
                int cluster_seeds_col_dim,
        IMSLS_CLUSTER_COUNTS, int **cluster_counts,
        IMSLS_CLUSTER_COUNTS_USER, int cluster_counts[],
        IMSLS_CLUSTER_VARIABLE_COLUMNS,
                int cluster_variables[],
        IMSLS_RETURN_USER, int cluster_group[],
        0)
```

Optional Arguments

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

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

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations.
Default: `max_iterations` = 30

`IMSLS_CLUSTER_MEANS`, *float* `**cluster_means` (Output)
The address of a pointer to an internally allocated array of length `n_clusters` × `n_variables` containing the cluster means.

`IMSLS_CLUSTER_MEANS_USER`, *float* `cluster_means[]` (Output)
Storage for array `cluster_means` is provided by the user. See `IMSLS_CLUSTER_MEANS`.

`IMSLS_CLUSTER_SSQ`, *float* `**cluster_ssq` (Output)
The address of a pointer to internally allocated array of length `n_clusters` containing the within sum-of-squares for each cluster.

`IMSLS_CLUSTER_SSQ_USER`, *float* `cluster_ssq[]` (Output)
Storage for array `cluster_ssq` is provided by the user. See `IMSLS_CLUSTER_SSQ`.

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

`IMSLS_CLUSTER_MEANS_COL_DIM`, *int* `cluster_means_col_dim` (Input)
Column dimension for the vector `cluster_means`.
Default: `cluster_means_col_dim = n_variables`

`IMSLS_CLUSTER_SEEDS_COL_DIM`, *int* `cluster_seeds_col_dim` (Input)
Column dimension for the vector `cluster_seeds`.
Default: `cluster_seeds_col_dim = n_variables`

`IMSLS_CLUSTER_COUNTS`, *int* `**cluster_counts` (Output)
The address of a pointer to an internally allocated array of length `n_clusters` containing the number of observations in each cluster.

`IMSLS_CLUSTER_COUNTS_USER`, *int* `cluster_counts[]` (Output)
Storage for array `cluster_counts` is provided by the user. See `IMSLS_CLUSTER_COUNTS`.

`IMSLS_CLUSTER_VARIABLE_COLUMNS`, *int* `cluster_variables[]` (Input)
Vector of length `n_variables` containing the columns of `x` to be used in computing the metric. Columns are numbered 0, 1, 2, ..., `n_variables`
Default: `cluster_variables` [ ] = 0, 1, 2, …, `n_variables`

`IMSLS_RETURN_USER`, *int* `cluster_group[]` (Output)
User-allocated array of length `n_observations` containing the cluster membership for each observation.

Description

Function `imsls_f_cluster_k_means` is an implementation of Algorithm AS 136 by Hartigan and Wong (1979). It computes *K*-means (centroid) Euclidean metric clusters for an input matrix starting with initial estimates of the *K*-cluster means. The function allows for missing values coded as NaN (Not a Number) and for weights and frequencies.

Let p = `n_variables` be the number of variables to be used in computing the Euclidean distance between observations. The idea in *K*-means cluster analysis is to find a clustering (or grouping) of the observations so as to minimize the total within-cluster sums-of-squares. In this case, the total sums-of-squares within each cluster is computed as the sum of the centered sum-of-squares over all nonmissing values of each variable. That is,

$$\phi = \sum_{i=1}^{K} \sum_{j=1}^{p} \sum_{m=1}^{n_i} f_{\nu_{im}} w_{\nu_{im}} \delta_{\nu_{im},j} \left(x_{\nu_{im},j} - \overline{x}_{ij} \right)^2$$

where ν_{im} denotes the row index of the *m*-th observation in the *i*-th cluster in the matrix *X*; n_i is the number of rows of *X* assigned to group *i*; *f* denotes the frequency of the observation; *w* denotes its weight; δ is 0 if the *j*-th variable on observation ν_{im} is missing, otherwise δ is 1; and

$$\overline{x}_{ij}$$

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` (Chapter 14, Utilities). The initial cluster seed for each iris type is an observation known to be in the iris type.

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

main()
{
#define N_OBSERVATIONS 150
#define N_VARIABLES    4
#define N_CLUSTERS     3
    float        x[N_OBSERVATIONS][5];
    float        cluster_seeds[N_CLUSTERS][N_VARIABLES];
    float        cluster_means[N_CLUSTERS][N_VARIABLES];
    float        cluster_ssq[N_CLUSTERS];
    int          cluster_variables[N_VARIABLES] = {1, 2, 3, 4};
    int          cluster_counts[N_CLUSTERS];
    int          cluster_group[N_OBSERVATIONS];
    int          i;

                 /* Retrieve the data set */
    imsls_f_data_sets(3, IMSLS_RETURN_USER, x, 0);
                 /* Assign initial cluster seeds */
    for (i=0; i<N_VARIABLES; i++) {
        cluster_seeds[0][i] = x[0][i+1];
        cluster_seeds[1][i] = x[50][i+1];
        cluster_seeds[2][i] = x[100][i+1];
    }

                 /* Perform the analysis */
    imsls_f_cluster_k_means(N_OBSERVATIONS, N_VARIABLES, (float*)x,
        N_CLUSTERS, (float*)cluster_seeds,
        IMSLS_X_COL_DIM,          5,
        IMSLS_CLUSTER_VARIABLE_COLUMNS,  cluster_variables,
        IMSLS_CLUSTER_COUNTS_USER,     cluster_counts,
        IMSLS_CLUSTER_MEANS_USER, cluster_means,
        IMSLS_CLUSTER_SSQ_USER,   cluster_ssq,
        IMSLS_RETURN_USER,        cluster_group,
        0);
                /* Print results */
    imsls_i_write_matrix("Cluster Membership", 1, N_OBSERVATIONS,
        cluster_group, 0);
    imsls_f_write_matrix("Cluster Means", N_CLUSTERS, N_VARIABLES,
        (float*)cluster_means, 0);
    imsls_f_write_matrix("Cluster Sum of Squares", 1, N_CLUSTERS,
        cluster_ssq, 0);
    imsls_i_write_matrix("# Observations in Each Cluster", 1,
        N_CLUSTERS, cluster_counts, 0);
}
```

```
                              Cluster Membership
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
 1  1  1  1  1  1  1  1  1   1   1   1   1   1   1   1   1   1   1   1

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 1  1  1  1  1  1  1  1  1  1  2  2  3  2  2  2  2  2  2  2

61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  3  2  2

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   2   3   3   3   3   2   3   3   3   3   3   3   2   2

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   2   3   2   3   2   3   3   2   2   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   2   3   3   3   2   3   3   3   2

148 149 150
  3   3   2

                  Cluster Means
            1           2           3           4
1       5.006       3.428       1.462       0.246
2       5.902       2.748       4.394       1.434
3       6.850       3.074       5.742       2.071

      Cluster Sum of Squares
         1           2           3
     15.15       39.82       23.88

# Observations in Each Cluster
           1    2    3
          50   62   38
```

Warning Errors

`IMSLS_NO_CONVERGENCE`	Convergence did not occur.

principal_components

Computes principal components.

Synopsis

#include `<imsls.h>`

float `*imsls_f_principal_components` (*int* `n_variables`,
float `covariances[]`, ..., 0)

The type *double* function is `imsls_d_principal_components`.

Required Arguments

int `n_variables` (Input)
Order of the covariance matrix.

float `covariances[]` (Input)
Array of length `n_variables` × `n_variables` containing the covariance or correlation matrix.

Return Value

An array of length `n_variables` containing the eigenvalues of the matrix `covariances` ordered from largest to smallest.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_principal_components` (*int* `n_variables`,
float `covariances[]`,
`IMSLS_COVARIANCE_MATRIX`, *or*
`IMSLS_CORRELATION_MATRIX`,
`IMSLS_CUM_PERCENT`, *float* `**cum_percent`,
`IMSLS_CUM_PERCENT_USER`, *float* `cum_percent[]`,
`IMSLS_EIGENVECTORS`, *float* `**eigenvectors`,
`IMSLS_EIGENVECTORS_USER`, *float* `eigenvectors[]`,
`IMSLS_CORRELATIONS`, *float* `**correlations`,
`IMSLS_CORRELATIONS_USER`, *float* `correlations[]`,
`IMSLS_STD_DEV`, *int* `n_degrees_freedom`, *float* `**std_dev`,
`IMSLS_STD_DEV_USER`, *int* `n_degrees_freedom`,
float `std_dev[]`,
`IMSLS_COV_COL_DIM`, *int* `cov_col_dim`,
`IMSLS_RETURN_USER`, *float* `eigenvalues[]`,
0)

Optional Arguments

`IMSLS_COVARIANCE_MATRIX`
: Treat the input vector `covariances` as a covariance matrix. This option is the default.
or

`IMSLS_CORRELATION_MATRIX`
: Treat the input vector `covariances` as a correlation matrix.

`IMSLS_CUM_PERCENT`, *float* `**cum_percent` (Output)
: The address of a pointer to an internally allocated array of length `n_variables` containing the cumulative percent of the total variances explained by each principal component.

`IMSLS_CUM_PERCENT_USER`, *float* `cum_percent[]` (Output)
: Storage for array `cum_percent` is provided by the user. See `IMSLS_CUM_PERCENT`.

`IMSLS_EIGENVECTORS`, *float* `**eigenvectors` (Output)
: The address of a pointer to an internally allocated array of length `n_variables` × `n_variables` containing the eigenvectors of `covariances`, stored columnwise. Each vector is normalized to have Euclidean length equal to the value one. Also, the sign of each vector is set so that the largest component in magnitude (the first of the largest if there are ties) is made positive.

`IMSLS_EIGENVECTORS_USER`, *float* `eigenvectors[]` (Output)
: Storage for array `eigenvectors` is provided by the user. See `IMSLS_EIGENVECTORS`.

`IMSLS_CORRELATIONS`, *float* `**correlations` (Output)
: The address of a pointer to an internally allocated array of length `n_variables * n_variables` containing the correlations of the principal components (the columns) with the observed/standardized variables (the rows). If `IMSLS_COVARIANCE_MATRIX` is specified, then the correlations are with the observed variables. Otherwise, the correlations are with the standardized (to a variance of 1.0) variables. In the principal component model for factor analysis, matrix `correlations` is the matrix of unrotated factor loadings.

`IMSLS_CORRELATIONS_USER`, *float* `correlations[]` (Output)
: Storage for array `correlations` is provided by the user. See `IMSLS_CORRELATIONS`.

`IMSLS_STD_DEV`, *int* `n_degrees_freedom`, *float* `**std_dev`
: (Input/Output)
Argument `n_degrees_freedom` contains the number of degrees of freedom in `covariances`. Argument `std_dev` is the address of a pointer to an internally allocated array of length `n_variables` containing the estimated asymptotic standard errors of the eigenvalues.

`IMSLS_STD_DEV_USER`, *int* `n_degrees_freedom`, *float* `std_dev[]` (Input/Output)
Storage for array `std_dev` is provided by the user. See `IMSLS_STD_DEV`.

`IMSLS_COV_COL_DIM` *int* `cov_col_dim` (Input)
Column dimension of covariances.
Default: `cov_col_dim = n_variables`

`IMSLS_RETURN_USER`, *float* `eigenvalues[]` (Output)
User-supplied array of length `n_variables` containing the eigenvalues of covariances ordered from largest to smallest.

Description

Function `imsls_f_principal_components` finds the principal components of a set of variables from a sample covariance or correlation matrix. The characteristic roots, characteristic vectors, standard errors for the characteristic roots, and the correlations of the principal component scores with the original variables are computed. Principal components obtained from correlation matrices are the same as principal components obtained from standardized (to unit variance) variables.

The principal component scores are the elements of the vector $y = \Gamma^T x$, where Γ is the matrix whose columns are the characteristic vectors (eigenvectors) of the sample covariance (or correlation) matrix and x is the vector of observed (or standardized) random variables. The variances of the principal component scores are the characteristic roots (eigenvalues) of the covariance (correlation) matrix.

Asymptotic variances for the characteristic roots were first obtained by Girschick (1939) and are given more recently by Kendall et al. (1983, p. 331). These variances are computed either for covariance matrices or for correlation matrices.

The correlations of the principal components with the observed (or standardized) variables are given in the matrix `correlations`. When the principal components are obtained from a correlation matrix, `correlations` is the same as the matrix of unrotated factor loadings obtained for the principal components model for factor analysis.

Examples

Example 1

In this example, eigenvalues of the covariance matrix are computed.

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

main()
{
#define N_VARIABLES 9

    float  *values;
    static float covariances[N_VARIABLES][N_VARIABLES] = {
        1.0,   0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0,   0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0,   0.355, 0.27,  0.254, 0.452, 0.219, 0.504,
        0.471, 0.506, 0.355, 1.0,   0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27,  0.691, 1.0,   0.679, 0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0,   0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0,   0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0,   0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68,  0.47,  1.0};

                 /* Perform analysis */
    values = imsls_f_principal_components(N_VARIABLES, covariances, 0);

                 /* Print results. */
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, values, 0);

                 /* Free allocated memory. */
    free(values);
}
```

Output

```
                          Eigenvalues
         1            2            3            4            5            6
     4.677        1.264        0.844        0.555        0.447        0.429

         7            8            9
     0.310        0.277        0.196
```

Example 2

In this example, principal components are computed for a nine-variable correlation matrix.

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

main()
{
#define N_VARIABLES 9

    float  *values, *eigenvectors, *std_dev, *cum_percent, *a;
    static float covariances[N_VARIABLES][N_VARIABLES] = {
        1.0,   0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0,   0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0,   0.355, 0.27,  0.254, 0.452, 0.219, 0.504,
        0.471, 0.506, 0.355, 1.0,   0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27,  0.691, 1.0,   0.679, 0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0,   0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0,   0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0,   0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68,  0.47,  1.0};

                    /* Perform analysis */
    values = imsls_f_principal_components(N_VARIABLES, covariances,
        IMSLS_CORRELATION_MATRIX,
        IMSLS_EIGENVECTORS,                    &eigenvectors,
        IMSLS_STD_DEV,                          100, &std_dev,
        IMSLS_CUM_PERCENT,                      &cum_percent,
        IMSLS_CORRELATIONS, &a,
        0);

                       /* Print results */
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, values, 0);
    imsls_f_write_matrix("Eigenvectors", N_VARIABLES, N_VARIABLES,
        eigenvectors, 0);
    imsls_f_write_matrix("STD", 1, N_VARIABLES, std_dev, 0);
    imsls_f_write_matrix("PCT", 1, N_VARIABLES, cum_percent, 0);
    imsls_f_write_matrix("A", N_VARIABLES, N_VARIABLES, a, 0);

                      /* Free allocated memory */
    free(values);
    free(eigenvectors);
    free (cum_percent);
    free (std_dev);
    free(a);
}
```

Output

```
                             Eigenvalues
         1            2            3            4            5            6
     4.677        1.264        0.844        0.555        0.447        0.429

         7            8            9
     0.310        0.277        0.196


                             Eigenvectors
            1            2            3            4            5            6
1      0.3462      -0.2354       0.1386      -0.3317      -0.1088       0.7974
2      0.3526      -0.1108      -0.2795      -0.2161       0.7664      -0.2002
3      0.2754      -0.2697      -0.5585       0.6939      -0.1531       0.1511
4      0.3664       0.4031       0.0406       0.1196       0.0017       0.1152
5      0.3144       0.5022      -0.0733      -0.0207      -0.2804      -0.1796
6      0.3455       0.4553       0.1825       0.1114       0.1202       0.0697
7      0.3487      -0.2714      -0.0725      -0.3545      -0.5242      -0.4355
8      0.2407      -0.3159       0.7383       0.4329       0.0861      -0.1969
9      0.3847      -0.2533      -0.0078      -0.1468       0.0459      -0.1498

            7            8            9
1      0.1735      -0.1240      -0.0488
2      0.1386      -0.3032      -0.0079
3      0.0099      -0.0406      -0.0997
4     -0.4022      -0.1178       0.7060
5      0.7295       0.0075       0.0046
6     -0.3742       0.0925      -0.6780
7     -0.2854      -0.3408      -0.1089
8      0.1862      -0.1623       0.0505
9     -0.0251       0.8521       0.1225

                                 STD
         1            2            3            4            5            6
    0.6498       0.1771       0.0986       0.0879       0.0882       0.0890
```

```
         7           8           9
    0.0944      0.0994      0.1113

                                    PCT
         1           2           3           4           5           6
     0.520       0.660       0.754       0.816       0.865       0.913

         7           8           9
     0.947       0.978       1.000

                                       A
            1           2           3           4           5           6
1      0.7487     -0.2646      0.1274     -0.2471     -0.0728      0.5224
2      0.7625     -0.1245     -0.2568     -0.1610      0.5124     -0.1312
3      0.5956     -0.3032     -0.5133      0.5170     -0.1024      0.0990
4      0.7923      0.4532      0.0373      0.0891      0.0012      0.0755
5      0.6799      0.5646     -0.0674     -0.0154     -0.1875     -0.1177
6      0.7472      0.5119      0.1677      0.0830      0.0804      0.0456
7      0.7542     -0.3051     -0.0666     -0.2641     -0.3505     -0.2853
8      0.5206     -0.3552      0.6784      0.3225      0.0576     -0.1290
9      0.8319     -0.2848     -0.0071     -0.1094      0.0307     -0.0981

            7           8           9
1      0.0966     -0.0652     -0.0216
2      0.0772     -0.1596     -0.0035
3      0.0055     -0.0214     -0.0442
4     -0.2240     -0.0620      0.3127
5      0.4063      0.0039      0.0021
6     -0.2084      0.0487     -0.3003
7     -0.1589     -0.1794     -0.0482
8      0.1037     -0.0854      0.0224
9     -0.0140      0.4485      0.0543
```

Warning Errors

`IMSLS_100_DF`	Because the number of degrees of freedom in “covariances” and “n_degrees_freedom” is less than or equal to 0, 100 degrees of freedom will be used.
`IMSLS_COV_NOT_NONNEG_DEF`	“eigenvalues[#]” = #. One or more eigenvalues much less than zero are computed. The matrix “covariances” is not nonnegative definite. In order to continue computations of “eigenvalues” and “correlations,” these eigenvalues are treated as 0.
`IMSLS_FAILED_TO_CONVERGE`	The iteration for the eigenvalue failed to converge in 100 iterations before deflating.

factor_analysis

Extracts initial factor-loading estimates in factor analysis with rotation options.

Synopsis

#include `<imsls.h>`

float `*imsls_f_factor_analysis` (*int* `n_variables`,
    float `covariances[]`, *int* `n_factors`, ..., 0)

The type *double* function is `imsls_d_factor_analysis`.

Required Arguments

int `n_variables` (Input)
    Number of variables.

float `covariances[]` (Input)
    Array of length `n_variables*n_variables` containing the variance-covariance or correlation matrix.

int `n_factors` (Input)
    Number of factors in the model.

Return Value

An array of length `n_variables*n_factors` containing the matrix of factor loadings.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_factor_analysis` (*int* `n_variables`,
    float `covariances[]`, *int* `n_factors`,
    `IMSLS_MAXIMUM_LIKELIHOOD`, *int* `df_covariances`, *or*
    `IMSLS_PRINCIPAL_COMPONENT`, *or*
    `IMSLS_PRINCIPAL_FACTOR`, *or*
    `IMSLS_UNWEIGHTED_LEAST_SQUARES`,*or*
    `IMSLS_GENERALIZED_LEAST_SQUARES`, *int* df_covariances, *or*
    `IMSLS_IMAGE`, *or*
    `IMSLS_ALPHA`, *int* `df_covariances`,
    `IMSLS_UNIQUE_VARIANCES_INPUT`, *float* `unique_variances[]`,
    `IMSLS_UNIQUE_VARIANCES_OUTPUT`,
        float `unique_variances[]`,
    `IMSLS_MAX_ITERATIONS`, *int* `max_iterations`,
    `IMSLS_MAX_STEPS_LINE_SEARCH`,
        int `max_steps_line_search`,
    `IMSLS_CONVERGENCE_EPS`, *float* `convergence_eps`,
    `IMSLS_SWITCH_EXACT_HESSIAN`, *float* `switch_epsilon`,

```
IMSLS_EIGENVALUES, float **eigenvalues,
IMSLS_EIGENVALUES_USER, float eigenvalues[],
IMSLS_CHI_SQUARED_TEST, int *df, float *chi_squared,
          float *p_value,
IMSLS_TUCKER_RELIABILITY_COEFFICIENT, float *coefficient,
IMSLS_N_ITERATIONS, int *n_iterations,
IMSLS_FUNCTION_MIN, float *function_min,
IMSLS_LAST_STEP, float **last_step,
IMSLS_LAST_STEP_USER, float last_step[],
IMSLS_ORTHOMAX_ROTATION, float w, int norm, float **b,
          float **t,
IMSLS_ORTHOMAX_ROTATION_USER, float w, int norm, float b[],
          float t[],
IMSLS_ORTHOGONAL_PROCUSTES_ROTATION, float target[],
          float **b, float **t,
IMSLS_ORTHOGONAL_PROCUSTES_ROTATION_USER,
          float target[], float b[], float t[],
IMSLS_DIRECT_OBLIMIN_ROTATION, float w, int norm, float **b,
          float **t, float **factor_correlations,
IMSLS_DIRECT_OBLIMIN_ROTATION_USER, float w, int norm,
          float b[], float t[], float factor_correlations[],
IMSLS_OBLIQUE_PROMAX_ROTATION, float w, float power[],
          int norm, float  **target, float **b, float **t,
          float **factor_correlations,
IMSLS_OBLIQUE_PROMAX_ROTATION_USER, float w, float power[],
          nt norm, float  target[], float b[], float t[],
          loat factor_correlations[],
IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION, float w,
          float pivot[], int norm, float  **target, float **b,
          float **t, float **factor_correlations,
IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION_USER, float w,
          loat pivot[], int norm, float  target[], float b[],
          float t[], float factor_correlations[],
IMSLS_OBLIQUE_PROCRUSTES_ROTATION,  float  target[],
          float **b, float **t, float **factor_correlations,
IMSLS_OBLIQUE_PROCRUSTES_ROTATION_USER,  float  target[],
          float b[], float t[], float factor_correlations[],
IMSLS_FACTOR_STRUCTURE, float  **s, float  **fvar,
IMSLS_FACTOR_STRUCTURE_USER, float s[], float  fvar[],
IMSLS_COV_COL_DIM, int cov_col_dim,
IMSLS_RETURN_USER, float  factor_loadings[],
0)
```

Optional Arguments

`IMSLS_MAXIMUM_LIKELIHOOD`, *int* `df_covariances` (Input)

Maximum likelihood (common factor model) method used to obtain the estimates. Argument `df_covariances` is the number of degrees of freedom in covariances.

or

`IMSLS_PRINCIPAL_COMPONENT`

Principal component (principal component model) method used to obtain the estimates.

or

`IMSLS_PRINCIPAL_FACTOR`

Principal factor (common factor model) method used to obtain the estimates.

or

`IMSLS_UNWEIGHTED_LEAST_SQUARES`

Unweighted least-squares (common factor model) method used to obtain the estimates. This option is the default.

or

`IMSLS_GENERALIZED_LEAST_SQUARES`, *int* `df_covariances` (Input)

Generalized least-squares (common factor model) method used to obtain the estimates.

or

`IMSLS_IMAGE`

Image-factor analysis (common factor model) method used to obtain the estimates.

or

`IMSLS_ALPHA`, *int* `df_covariances` (Input)

Alpha-factor analysis (common factor model) method used to obtain the estimates. Argument `df_covariances` is the number of degrees of freedom in covariances.

`IMSLS_UNIQUE_VARIANCES_INPUT`, *float* `unique_variances[]` (Input)

Array of length `n_variables` containing the initial estimates of the unique variances.
Default: Initial estimates are taken as the constant
1 – `n_factors`/2 * `n_variables` divided by the diagonal elements of the inverse of `covariances`.

`IMSLS_UNIQUE_VARIANCES_OUTPUT`, *float* `unique_variances[]` (Output)

User-allocated array of length `n_variables` containing the estimated unique variances.

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)

Maximum number of iterations in the iterative procedure.
Default: `max_iterations` = 60

`IMSLS_MAX_STEPS_LINE_SEARCH`, *int* `max_steps_line_search` (Input)

Maximum number of step halvings allowed during any one iteration.
Default: `max_steps_line_search` = 10

IMSLS_CONVERGENCE_EPS, *float* convergence_eps (Input)
Convergence criterion used to terminate the iterations. For the unweighted least squares, generalized least squares or maximum likelihood methods, convergence is assumed when the relative change in the criterion is less than convergence_eps. For alpha-factor analysis, convergence is assumed when the maximum change (relative to the variance) of a uniqueness is less than convergence_eps.
Default: convergence_eps = 0.0001

IMSLS_SWITCH_EXACT_HESSIAN, *float* switch_epsilon (Input)
Convergence criterion used to switch to exact second derivatives. When the largest relative change in the unique standard deviation vector is less than switch_epsilon, exact second derivative vectors are used. Argument switch_epsilon is not used with the principal component, principal factor, image-factor analysis, or alpha-factor analysis methods.
Default: switch_epsilon = 0.1

IMSLS_EIGENVALUES, *float* **eigenvalues (Output)
The address of a pointer to an internally allocated array of length n_variables containing the eigenvalues of the matrix from which the factors were extracted.

IMSLS_EIGENVALUES_USER, *float* eigenvalues[] (Output)
Storage for array eigenvalues is provided by the user. See IMSLS_EIGENVALUES.

IMSLS_CHI_SQUARED_TEST, *int* *df, *float* *chi_squared, *float* *p_value (Output)
Number of degrees of freedom in chi-squared is df; chi_squared is the chi-squared test statistic for testing that n_factors common factors are adequate for the data; p_value is the probability of a greater chi-squared statistic.

IMSLS_TUCKER_RELIABILITY_COEFFICIENT, *float* *coefficient (Output)
Tucker reliability coefficient.

IMSLS_N_ITERATIONS, *int* *n_iterations (Output)
Number of iterations.

IMSLS_FUNCTION_MIN, *float* *function_min (Output)
Value of the function minimum.

IMSLS_LAST_STEP, *float* **last_step (Output)
Address of a pointer to an internally allocated array of length n_variables containing the updates of the unique variance estimates when convergence was reached (or the iterations terminated).

IMSLS_LAST_STEP_USER, *float* last_step[] (Output)
Storage for array last_step is provided by the user. See IMSLS_LAST_STEP.

`IMSLS_ORTHOMAX_ROTATION`, *float* `w`, *int* `norm`, *float* `**b`, *float* `**t` (Input/Output)
Nonnegative constant `w` defines the rotation. If `norm` =1, row normalization is performed. Otherwise, row normalization is not performed. `b` contains the address of a pointer to the internally allocated array of length `n_variables`*`n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors`*`n_factors` containing the rotation transformation matrix. `w` = 0.0 results in quartimax rotations, `w` = 1.0 results in varimax rotations, and `w` = `n_factors`/2.0 results in equamax rotations. Other nonnegative values of `w` may also be used, but the best values for `w` are in the range (0.0, 5 * `n_factors`).

`IMSLS_ORTHOMAX_ROTATION_USER`, *float* `w`, *int* `norm`, *float* `b[]`, *float* `t[]` (Input/Output)
Storage for b and t are provided by the user. See `IMSLS_ORTHOMAX_ROTATION.`

`IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION`, *float* `target[]`, *float* `**b`, *float* `**t` (Input/Output)
If specified, the `n_variables` by `n_factors` target matrix `target` will be used to compute an orthogonal Procrustes rotation of the factor-loading matrix. `b` contains the address of a pointer to the internally allocated array of length `n_variables`*`n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors`*`n_factors` containing the rotation transformation matrix.

`IMSLS_ORTHOGONAL_PROCRUTES_ROTATION_USER`, *float* `target[]`, *float* `b[]`, *float* `t[]` (Input/Output)
Storage for `b` and `t` are provided by the user. See `IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION`.

`IMSLS_DIRECT_OBLIMIN_ROTATION`, *float* `w` , *int* `norm`, *float* `**b`, *float* `**t`, *float* `**factor_correlations` (Input/Output)
Computes a direct oblimin rotation. Nonpositive constant `w` defines the rotation. If `norm` =1, row normalization is performed. Otherwise, row normalization is not performed. `b` contains the address of a pointer to the internally allocated array of length `n_variables`*`n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors`*`n_factors` containing the rotation transformation matrix. `factor_correlations` contains the address of a pointer to the internally allocated array of length `n_factors`*`n_factors` containing the factor correlations. The parameter `w` determines the type of direct oblimin rotation to be performed. In general `w` must be negative. `w` = 0.0 results in direct quartimin rotations. As `w` approaches negative infinity, the orthogonality among factors will increase.

IMSLS_DIRECT_OBLIMIN_ROTATION_USER, *float* w, *int* norm, *float* b[], *float* t[], *float* factor_correlations[] (Input/Output)
Storage for b, t and factor_correlations are provided by the user. See IMSLS_DIRECT_OBLIMIN_ROTATION.

IMSLS_OBLIQUE_PROMAX_ROTATION, *float* w, *float* power[], *int* norm, *float* **target, *float* **b, *float* **t, *float* **factor_correlations, (Input/Output)
Computes an oblique promax rotation of the factor loading matrix using a power vector. Nonnegative constant w defines the rotation. power, a vector of length n_factors containing the power vector. If norm =1, row (Kaiser) normalization is performed. Otherwise, row normalization is not performed. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix. factor_correlations contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the factor correlations. target contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the target matrix for rotation, derived from the orthomax rotation. w is used in the orthomax rotation, see the optional argument IMSLS_ORTHOMAX_ROTATION for common values of w.

All power[j] should be greater than 1.0, typically 4.0. Generally, the larger the values of power [j], the more oblique the solution will be.

IMSLS_OBLIQUE_PROMAX_ROTATION_USER, *float* w, *float* power[], *int* norm, *float* target[], *float* b[], *float* t[], *float* factor_correlations[], (Input/Output)
Storage for b, t, factor_correlations, and target are provided by the user. See IMSLS_OBLIQUE_PROMAX_ROTATION.

IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION, *float* w, *float* pivot[], *int* norm, *float* **target , *float* **b, *float* **t, *float* **factor_correlations, (Input/Output)
Computes an oblique pivotal promax rotation of the factor loading matrix using pivot constants. Nonnegative constant w defines the rotation. pivot, a vector of length n_factors containing the pivot constants. pivot[j] should be in the interval (0.0, 1.0). If norm =1, row (Kaiser) normalization is performed. Otherwise, row normalization is not performed. b contains the address of a pointer to the internally allocated array of length n_variables*n_factors containing the rotated factor loading matrix. t contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the rotation transformation matrix. factor_correlations contains the address of a pointer to the internally allocated array of length n_factors*n_factors containing the factor correlations. target

contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the target matrix for rotation, derived from the orthomax rotation. `w` is used in the orthomax rotation, see the optional argument `IMSLS_ORTHOMAX_ROTATION` for common values of `w`.

`IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION_USER`, *float* `w`, *float* `pivot[]`, *int* `norm`, *float* `target[]`, *float* `b[]`, *float* `t[]`, *float* `factor_correlations[]`, (Input/Output)
Storage for `b`, `t`, `factor_correlations`, and `target` are provided by the user. See `IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`.

`IMSLS_OBLIQUE_PROCRUSTES_ROTATION`, *float* `**target`, *float* `**b`, *float* `**t`, *float* `**factor_correlations` (Input/Output)
Computes an oblique procrustes rotation of the factor loading matrix using a target matrix. `target` is a hypothesized rotated factor loading matrix based upon prior knowledge with loadings chosen to the enhance interpretability. A simple structure solution will have most of the weights `target[i][j]` either zero or large in magnitude. `b` contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the rotated factor loading matrix. `t` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the rotation transformation matrix. `factor_correlations` contains the address of a pointer to the internally allocated array of length `n_factors*n_factors` containing the factor correlations.

`IMSLS_OBLIQUE_PROCRUSTES_ROTATION_USER`, *float* `target[]`, *float* `b[]`, *float* `t[]`, *float* `factor_correlations[]` (Input/Output)
Storage for `b`, `t`, and `factor_correlations` are provided by the user. See `IMSLS_PROCRUSTES_ROTATION`.

`IMSLS_FACTOR_STRUCTURE`, *float* `**s`, *float* `**fvar`, (Output)
Computes the factor structure and the variance explained by each factor. `s` contains the address of a pointer to the internally allocated array of length `n_variables*n_factors` containing the factor structure matrix. `fvar` contains the address of a pointer to the internally allocated array of length `n_factors` containing the variance accounted for by each of the `n_factors` rotated factors. A factor rotation matrix is used to compute the factor structure and the variance. One and only one rotation option argument can be specified.

`IMSLS_FACTOR_STRUCTURE_USER`, *float* `s[]`, *float* `fvar[]`, (Output)
`Storage for s, and fvar are provided by the user.`
See `IMSLS_FACTOR_STRUCTURE`.

`IMSLS_COV_COL_DIM`, *int* `cov_col_dim` (Input)
Column dimension of the matrix `covariances`.
Default: `cov_col_dim = n_variables`

`IMSLS_RETURN_USER`, *float* `factor_loadings[]` (Output)
User-allocated array of length `n_variables*n_factors` containing the unrotated factor loadings.

Description

Function `imsls_f_factor_analysis` computes factor loadings in exploratory factor analysis models. Models available in `imsls_f_factor_analysis` are the principal component model for factor analysis and the common factor model with additions to the common factor model in alpha-factor analysis and image analysis. Methods of estimation include principal components, principal factor, image analysis, unweighted least squares, generalized least squares, and maximum likelihood.

In the factor analysis model used for factor extraction, the basic model is given as $\Sigma = \Lambda\Lambda^T + \Psi$, where Σ is the $p \times p$ population covariance matrix, Λ is the $p \times k$ matrix of factor loadings relating the factors f to the observed variables x, and Ψ is the $p \times p$ matrix of covariances of the unique errors e. Here, p = `n_variables` and k = `n_factors`. The relationship between the factors, the unique errors, and the observed variables is given as $x = \Lambda f + e$, where in addition, the expected values of e, f, and x are assumed to be 0. (The sample means can be subtracted from x if the expected value of x is not 0.) It also is assumed that each factor has unit variance, the factors are independent of each other, and that the factors and the unique errors are mutually independent. In the common factor model, the elements of unique errors e also are assumed to be independent of one another so that the matrix Ψ is diagonal. This is not the case in the principal component model in which the errors may be correlated.

Further differences between the various methods concern the criterion that is optimized and the amount of computer effort required to obtain estimates. Generally speaking, the least-squares and maximum likelihood methods, which use iterative algorithms, require the most computer time with the principal factor, principal component and the image methods requiring much less time since the algorithms in these methods are not iterative. The algorithm in alpha-factor analysis is also iterative, but the estimates in this method generally require somewhat less computer effort than the least-squares and maximum likelihood estimates. In all methods, one eigensystem analysis is required on each iteration.

Principal Component and Principal Factor Methods

Both the principal component and principal factor methods compute the factor-loading estimates as

$$\hat{\Gamma}\hat{\Delta}^{-1/2}$$

where Γ and the diagonal matrix Δ are the eigenvectors and eigenvalues of a matrix. In the principal component model, the eigensystem analysis is performed on the sample covariance (correlation) matrix S, while in the principal factor model, the matrix $(S + \Psi)$ is used. If the unique error variances Ψ are not known in the principal factor mode, then `imsls_f_factor_analysis` obtains estimates for them.

The basic idea in the principal component method is to find factors that maximize the variance in the original data that is explained by the factors. Because this method allows the unique errors to be correlated, some factor analysts insist that the principal component method is not a factor analytic method. Usually, however, the estimates obtained by the principal component model and factor analysis model will be quite similar.

It should be noted that both the principal component and principal factor methods give different results when the correlation matrix is used in place of the covariance matrix. Indeed, any rescaling of the sample covariance matrix can lead to different estimates with either of these methods. A further difficulty with the principal factor method is the problem of estimating the unique error variances. Theoretically, these must be known in advance and be passed to `imsls_f_factor_analysis` using optional argument `IMSLS_UNIQUE_VARIANCES_INPUT`. In practice, the estimates of these parameters are produced by `imsls_f_factor_analysis` when `IMSLS_UNIQUE_VARIANCES_INPUT` is not specified. In either case, the resulting adjusted covariance (correlation) matrix

$$S - \hat{\psi}$$

may not yield the `n_factors` positive eigenvalues required for `n_factors` factors to be obtained. If this occurs, the user must either lower the number of factors to be estimated or give new unique error variance values.

Least-squares and Maximum Likelihood Methods

Unlike the previous two methods, the algorithm used to compute estimates in this section is iterative (see Jöreskog 1977). As with the principal factor model, the user may either initialize the unique error variances or allow `imsls_f_factor_analysis` to compute initial estimates. Unlike the principal factor method, `imsls_f_factor_analysis` optimizes the criterion function with respect to both Ψ and Γ. (In the principal factor method, Ψ is assumed to be known. Given Ψ, estimates for Λ may be obtained.)

The major difference between the methods discussed in this section is in the criterion function that is optimized. Let S denote the sample covariance (correlation) matrix, and let Σ denote the covariance matrix that is to be estimated by the factor model. In the unweighted least-squares method, also called the iterated principal factor method or the minres method (see Harman 1976, p. 177), the function minimized is the sum-of-squared differences between S and Σ. This is written as $\Phi_{ul} = 0.5$ (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 ϕ (Ψ, Λ (Ψ)), 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:

$$\rho = \frac{mM_0 - mM_k}{mM_0 - 1}$$

$$m = d - \frac{2p+5}{6} - \frac{2k}{6}$$

$$M_0 = \frac{-\ln(|S|)}{p(p-1)/2}$$

$$M_k = \frac{\phi}{\left((p-k)^2 - p - k\right)/2}$$

where $|S|$ is the determinant of covariances, p = `n_variables`, k = `n_variables`, ϕ is the optimized criterion, and d = `df_covariances`.

Image Analysis Method

The term *image analysis* is used here to denote the noniterative image method of Kaiser (1963). It is not the image analysis discussed by Harman (1976, p. 226). The image method (as well as the alpha-factor analysis method) begins with the notion that only a finite number from an infinite number of possible variables have been measured. The image factor pattern is calculated under the assumption that the ratio of the number of factors to the number of observed variables is near 0, so that a very good estimate for the unique error variances (for standardized variables) is given as 1 minus the squared multiple correlation of the variable under consideration with all variables in the covariance matrix.

First, the matrix $D^2 = (\text{diag}\,(S^{-1}))^{-1}$ is computed where the operator "diag" results in a matrix consisting of the diagonal elements of its argument and S is the sample covariance (correlation) matrix. Then, the eigenvalues Λ and eigenvectors Γ of the matrix $D^{-1}SD^{-1}$ are computed. Finally, the unrotated image-factor pattern is computed as $D\Gamma\,[(\Lambda - I)^2\Lambda^{-1}]^{1/2}$.

Alpha-factor Analysis Method

The alpha-factor analysis method of Kaiser and Caffrey (1965) finds factor-loading estimates to maximize the correlation between the factors and the complete universe of variables of interest. The basic idea in this method is that only a finite number of variables out of a much larger set of possible variables is observed. The population factors are linearly related to this larger set, while the observed factors are linearly related to the observed variables. Let f denote the factors obtainable from a finite set of observed random variables, and let ξ denote the factors obtainable from the universe of observable variables. Then, the alpha method attempts to find factor-loading estimates so as to maximize the correlation between f and ξ. In order to obtain these estimates, the iterative algorithm of Kaiser and Caffrey (1965) is used.

Rotation Methods

The `IMSLS_ORTHOMAX_ROTATION` optional argument performs an orthogonal rotation according to an orthomax criterion. In this analytic method of rotation, the criterion function

$$Q = \sum_i \sum_r \lambda_{ir}^4 - \frac{\gamma}{p} \sum_r \left[\sum_i \lambda_{ir}^2 \right]^2$$

is minimized by finding an orthogonal rotation matrix T such that $(\lambda_{ij}) = \Lambda = AT$ where A is the matrix of unrotated factor loadings. Here, $\gamma \geq 0$ is a user-specified constant (W) yielding a family of rotations, and p is the number of variables.

Kaiser (row) normalization can be performed on the factor loadings prior to rotation by specifying the parameter `norm =1`. In Kaiser normalization, the rows of `A` are first "normalized" by dividing each row by the square root of the sum of its squared elements (Harman 1976). After the rotation is complete, each row of `b` is "denormalized" by multiplication by its initial normalizing constant.

The method for optimizing Q proceeds by accumulating simple rotations where a simple rotation is defined to be one in which Q is optimized for two columns in Λ and for which the requirement that T be orthogonal is satisfied. `A` single iteration is defined to be such that each of the `n_factors`(`n_factors` – 1)/2 possible simple rotations is performed where `n_factors` is the number of factors. When the relative change in Q from one iteration to the next is less than `EPS` (the user-specified convergence criterion), the algorithm stops. `eps` = 0.0001 is usually sufficient. Alternatively, the algorithm stops when the user-specified maximum number of iterations, `max_iterations`, is reached. `max_iterations` = 30 is usually sufficient.

The parameter in the rotation, γ, is used to provide a family of rotations. When $\gamma = 0.0$, a direct quartimax rotation results. Other values of γ yield other rotations.

The `IMSLS_ORTHOGONAL_PROCRUSTES_ROTATION` optional argument performs orthogonal Procrustes rotation according to a method proposed by Schöneman (1966). Let k = `n_factors` denote the number of factors, p = `n_variables` denote the number of variables, A denote the $p \times k$ matrix of unrotated factor loadings, T denote the $k \times k$ orthogonal rotation matrix (orthogonality requires that $T^T T$ be a $k \times k$ identity matrix), and let X denote the target matrix. The basic idea in orthogonal Procrustes rotation is to find an orthogonal rotation matrix T such that $B = AT$ and T provides a least-squares fit between the target matrix X and the rotated loading matrix B. Schöneman's algorithm proceeds by finding the singular value decomposition of the matrix $A^T X = U\Sigma V^T$. The rotation matrix is computed as $T = UV^T$.

The `IMSLS_DIRECT_OBLIMIN_ROTATION` optional argument performs direct oblimin rotation. In this analytic method of rotation, the criterion function

$$Q = \sum_{r \neq s} \left[\sum_i \lambda_{ir}^2 \lambda_{is}^2 - \frac{\gamma}{p} \sum_i \lambda_{ir}^2 \sum_i \lambda_{is}^2 \right]$$

is minimized by finding a rotation matrix T such that $(\lambda_{ir}) = \Lambda = AT$ and $(T^T T)^{-1}$ is a correlation matrix. Here, $\gamma \leq 0$ 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 rotations is performed, where `n_factors` is the number of factors. When the relative change in Q from one iteration to the next is less than `eps` (the user-specified convergence criterion), the algorithm stops. `eps` = .0001 is usually sufficient. Alternatively, the algorithm stops when the user-specified maximum number of iterations, `max_iterations`, is reached. `max_iterations` = 30 is usually sufficient.

The parameter in the rotation, γ, is used to provide a family of rotations. Harman (1976) recommends that γ be strictly less than or equal to zero. When $\gamma = 0.0$, a direct quartimin rotation results. Other values of γ yield other rotations. Harman (1976) suggests that the direct quartimin rotations yield the most highly correlated factors while more orthogonal factors result as γ approaches $-\infty$.

`IMSLS_OBLIQUE_PROMAX_ROTATION`,
`IMSLS_OBLIQUE_PIVOTAL_PROMAX_ROTATION`,
`IMSLS_OBLIQUE_PROCRUSTES_ROTATION`, optional arguments performs oblique rotations using the Promax, pivotal Promax, or oblique Procrustes methods. In all of these methods, a target matrix *X* is first either computed or specified by the user. The differences in the methods relate to how the target matrix is first obtained.

Given a $p \times k$ target matrix, *X*, and a $p \times k$ orthogonal matrix of unrotated factor loadings, *A*, compute the rotation matrix *T* as follows: First regress each column of *A* on *X* yielding a $k \times k$ matrix β. Then, let $\gamma = \text{diag}(\beta^T \beta)$ 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]`$/b_{ij}$. This has the effects of greatly increasing in *X* those elements in *B* that are greater in magnitude than the pivot elements `pivot[i]`, and of greatly decreasing those elements that are less than `pivot[i]`.

In the oblique Procrustes method, the elements of *X* are specified by the user as input to the routine via the `target` argument. No orthogonal rotation is performed in the oblique Procrustes method.

Factor Structure and Variance

The `IMSLS_FACTOR_STRUCTURE` optional argument computes the factor structure matrix (the matrix of correlations between the observed variables and the hypothesized factors) and the variance explained by each of the factors (for orthogonal rotations). For oblique rotations, `IMSLS_FACTOR_STRUCTURE` computes a measure of the importance of the factors, the sum of the squared elements in each column.

Let Δ denote the diagonal matrix containing the elements of the variance of the original data along its diagonal. The estimated factor structure matrix *S* is computed as

$$S = \Delta^{-\frac{1}{2}} A (T^{-1})^T$$

while the elements of `fvar` are computed as the diagonal elements of

$$S^T \Delta^{\frac{1}{2}} AT$$

If the factors were obtained from a correlation matrix (or the factor variances for standardized variables are desired), then the variances should all be 1.0.

Comments

1. Function `imsls_f_factor_analysis` makes no attempt to solve for `n_factors`. In general, if `n_factors` is not known in advance, several different values of `n_factors` should be used and the most reasonable value kept in the final solution.
2. Iterative methods are generally thought to be superior from a theoretical point of view, but in practice, often lead to solutions that differ little from the noniterative methods. For this reason, it is usually suggested that a noniterative method be used in the initial stages of the factor analysis and that the iterative methods be used when issues such as the number of factors have been resolved.
3. Initial estimates for the unique variances can be input. If the iterative methods fail for these values, new initial estimates should be tried. These can be obtained by use of another factoring method. (Use the final estimates from the new method as the initial estimates in the old method.)

Examples

Example 1

In this example, factor analysis is performed for a nine-variable matrix using the default method of unweighted least squares.

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

main()
{
#define N_VARIABLES 9
#define N_FACTORS   3
    float *a;

    float covariances[N_VARIABLES][N_VARIABLES] = {
        1.0,   0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0,   0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0,   0.355, 0.27,  0.254, 0.452, 0.219, 0.504,
        0.471, 0.506, 0.355, 1.0,   0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27,  0.691, 1.0,   0.679, 0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0,   0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0,   0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0,   0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68,  0.47,  1.0};

                    /* Perform analysis */
    a = imsls_f_factor_analysis (9, covariances, 3, 0);
```

```
                        /* Print results */
    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
        a, 0);

    free(a);
}
```

Output

```
         Unrotated Loadings
                1           2           3
1      0.7018     -0.2316      0.0796
2      0.7200     -0.1372     -0.2082
3      0.5351     -0.2144     -0.2271
4      0.7907      0.4050      0.0070
5      0.6532      0.4221     -0.1046
6      0.7539      0.4842      0.1607
7      0.7127     -0.2819     -0.0701
8      0.4835     -0.2627      0.4620
9      0.8192     -0.3137     -0.0199
```

Example 2

The following data were originally analyzed by Emmett (1949). There are 211 observations on 9 variables. Following Lawley and Maxwell (1971), three factors are obtained by the method of maximum likelihood.

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

main()
{
#define N_VARIABLES 9
#define N_FACTORS   3
    float *a;
    float *evals;
    float chi_squared, p_value, reliability_coef, function_min;
    int   chi_squared_df, n_iterations;
    float uniq[N_VARIABLES];

    float covariances[N_VARIABLES][N_VARIABLES] = {
        1.0,   0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0,   0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0,   0.355, 0.27,  0.254, 0.452, 0.219, 0.504,
        0.471, 0.506, 0.355, 1.0,   0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27,  0.691, 1.0,   0.679, 0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0,   0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0,   0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0,   0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68,  0.47,  1.0};

                            /* Perform analysis */
    a = imsls_f_factor_analysis (9, covariances, 3,
        IMSLS_MAXIMUM_LIKELIHOOD,           210,
        IMSLS_SWITCH_EXACT_HESSIAN,         0.01,
        IMSLS_CONVERGENCE_EPS,              0.000001,
        IMSLS_MAX_ITERATIONS,               30,
        IMSLS_MAX_STEPS_LINE_SEARCH,        10,
        IMSLS_EIGENVALUES,                  &evals,
        IMSLS_UNIQUE_VARIANCES_OUTPUT,      uniq,
        IMSLS_CHI_SQUARED_TEST,
            &chi_squared_df,
            &chi_squared,
            &p_value,
        IMSLS_TUCKER_RELIABILITY_COEFFICIENT, &reliability_coef,
        IMSLS_N_ITERATIONS,                 &n_iterations,
        IMSLS_FUNCTION_MIN,                 &function_min,
        0);

                        /* Print results */
    imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
        a, 0);
    imsls_f_write_matrix("Eigenvalues", 1, N_VARIABLES, evals, 0);
    imsls_f_write_matrix("Unique Error Variances", 1, N_VARIABLES,
        uniq, 0);
    printf("\n\nchi_squared_df =    %d\n", chi_squared_df);
    printf("chi_squared =       %f\n", chi_squared);
    printf("p_value =           %f\n\n", p_value);
    printf("reliability_coef = %f\n", reliability_coef);
    printf("function_min =      %f\n", function_min);
    printf("n_iterations =      %d\n", n_iterations);

    free(evals);
    free(a);
}
```

Output

```
         Unrotated Loadings
            1           2           3
1      0.6642     -0.3209      0.0735
2      0.6888     -0.2471     -0.1933
3      0.4926     -0.3022     -0.2224
4      0.8372      0.2924     -0.0354
5      0.7050      0.3148     -0.1528
6      0.8187      0.3767      0.1045
7      0.6615     -0.3960     -0.0777
8      0.4579     -0.2955      0.4913
9      0.7657     -0.4274     -0.0117

                             Eigenvalues
         1           2           3           4           5           6
     0.063       0.229       0.541       0.865       0.894       0.974

         7           8           9
     1.080       1.117       1.140

                       Unique Error Variances
         1           2           3           4           5           6
    0.4505      0.4271      0.6166      0.2123      0.3805      0.1769

         7           8           9
    0.3995      0.4615      0.2309


chi_squared_df =    12
```

```
chi_squared =       7.149356
p_value =           0.847588

reliability_coef = 1.000000
function_min =      0.035017
n_iterations =      5
```

Example 3

This example is a continuation of example 1 and illustrates the use of the `IMSLS_FACTOR_STRUCTURE` optional argument when the structure and an index of factor importance for obliquely rotated loadings are desired. A direct oblimin rotation is used to compute the factors, derived from nine variables and using $\gamma = -1$. Note in this example that the elements of `fvar` are not variances since the rotation is oblique.

```
#include <stdio.h>
#include <imsls.h>
#include <stdlib.h>
void main()
{
#define N_VARIABLES 9
#define N_FACTORS   3
    float *a;
    float w= -1.0;
    int   norm=1;
    float *b, *t, *fcor;
    float *s, *fvar;
    float covariances[9][9] = {
        1.0,   0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639,
        0.523, 1.0,   0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645,
        0.395, 0.479, 1.0,   0.355, 0.27,  0.254, 0.452, 0.219, 0.504,
        0.471, 0.506, 0.355, 1.0,   0.691, 0.791, 0.443, 0.285, 0.505,
        0.346, 0.418, 0.27,  0.691, 1.0,   0.679, 0.383, 0.149, 0.409,
        0.426, 0.462, 0.254, 0.791, 0.679, 1.0,   0.372, 0.314, 0.472,
        0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0,   0.385, 0.68,
        0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0,   0.47,
        0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68,  0.47,  1.0};

                    /* Perform analysis */
  a = imsls_f_factor_analysis (9, (float *)covariances, 3,
      IMSLS_MAXIMUM_LIKELIHOOD,           210,
      IMSLS_SWITCH_EXACT_HESSIAN,         0.01,
      IMSLS_CONVERGENCE_EPS,              0.00001,
      IMSLS_MAX_ITERATIONS,               30,
      IMSLS_MAX_STEPS_LINE_SEARCH,        10,
      IMSLS_DIRECT_OBLIMIN_ROTATION, w, norm, &b, &t, &fcor,
      IMSLS_FACTOR_STRUCTURE, &s, &fvar,
      0);

                  /* Print results */

  imsls_f_write_matrix("Unrotated Loadings", N_VARIABLES, N_FACTORS,
      a, 0);
  imsls_f_write_matrix("Rotated Loadings", N_VARIABLES, N_FACTORS,
      b, 0);
    imsls_f_write_matrix("Transformation Matrix", N_FACTORS, N_FACTORS,
        t, 0);
    imsls_f_write_matrix("Factor Correlation Matrix", N_FACTORS, N_FACTORS,
        fcor, 0);
    imsls_f_write_matrix("Factor Structure",  N_VARIABLES,
        N_FACTORS,s,0);
    imsls_f_write_matrix("Factor Variance", 1, N_FACTORS, fvar, 0);
}
```

Output

```
          Unrotated Loadings
              1           2           3
1        0.6642     -0.3209      0.0735
2        0.6888     -0.2471     -0.1933
3        0.4926     -0.3022     -0.2224
4        0.8372      0.2924     -0.0354
5        0.7050      0.3148     -0.1528
6        0.8187      0.3767      0.1045
7        0.6615     -0.3960     -0.0777
8        0.4579     -0.2955      0.4913
9        0.7657     -0.4274     -0.0117

           Rotated Loadings
              1           2           3
1        0.1128     -0.5144      0.2917
2        0.1847     -0.6602     -0.0018
3        0.0128     -0.6354     -0.0585
4        0.7797     -0.1751      0.0598
5        0.7147     -0.1813     -0.0959
6        0.8520      0.0039      0.1820
7        0.0354     -0.6844      0.1510
8        0.0276     -0.0941      0.6824
9        0.0729     -0.7100      0.2493

         Transformation Matrix
              1           2           3
1         0.611      -0.462       0.203
2         0.923       0.813      -0.249
3         0.042       0.728       1.050

       Factor Correlation Matrix
              1           2           3
1         1.000      -0.427       0.217
2        -0.427       1.000      -0.411
3         0.217      -0.411       1.000

           Factor Structure
              1           2           3
1        0.3958     -0.6824      0.5275
2        0.4662     -0.7383      0.3094
3        0.2714     -0.6169      0.2052
4        0.8675     -0.5326      0.3011
5        0.7713     -0.4471      0.1339
6        0.8899     -0.4347      0.3656
7        0.3605     -0.7616      0.4398
8        0.2161     -0.3861      0.7271
9        0.4302     -0.8435      0.5568
```

```
      Factor Variance
    1           2           3
2.170       2.560       0.914
```

Warning Errors

`IMSLS_VARIANCES_INPUT_IGNORED`	When using the `IMSLS_PRINCIPAL_COMPONENT` option, the unique variances are assumed to be zero. Input for `IMSLS_UNIQUE_VARIANCES_INPUT` is ignored.
`IMSLS_TOO_MANY_ITERATIONS`	Too many iterations. Convergence is assumed.
`IMSLS_NO_DEG_FREEDOM`	There are no degrees of freedom for the significance testing.
`IMSLS_TOO_MANY_HALVINGS`	Too many step halvings. Convergence is assumed.
`IMSLS_NO_ROTATION`	`n_factors = 1`. No rotation is possible.
`IMSLS_SVD_ERROR`	An error occurred in the singular value decomposition of tran(A)*X. The rotation matrix, T, may not be correct.

Fatal Errors

`IMSLS_HESSIAN_NOT_POS_DEF`	The approximate Hessian is not semi-definite on iteration #. The computations cannot proceed. Try using different initial estimates.
`IMSLS_FACTOR_EVAL_NOT_POS`	"eigenvalues[#]" = #. An eigenvalue corresponding to a factor is negative or zero. Either use different initial estimates for "unique_variances" or reduce the number of factors.
`IMSLS_COV_NOT_POS_DEF`	"covariances" is not positive semi-definite. The computations cannot proceed.
`IMSLS_COV_IS_SINGULAR`	The matrix "covariances" is singular. The computations cannot continue because variable # is linearly related to the remaining variables.

`IMSLS_COV_EVAL_ERROR`	An error occurred in calculating the eigenvalues of the adjusted (inverse) covariance matrix. Check “covariances.”
`IMSLS_ALPHA_FACTOR_EVAL_NEG`	In alpha factor analysis on iteration #, eigenvalue # is #. As all eigenvalues corresponding to the factors must be positive, either the number of factors must be reduced or new initial estimates for “unique_variances” must be given.
`IMSLS_RANK_LESS_THAN`	The rank of TRAN(A)*`target` = #. This must be greater than or equal to `n_factors` = #.

discriminant_analysis

Performs a linear or a quadratic discriminant function analysis among several known groups.

Synopsis

#include `<imsls.h>`

void `imsls_f_discriminant_analysis` (*int* `n_rows`, *int* `n_variables`,
    float `*x`, *int* `n_groups`, ..., 0)

The type *double* function is `imsls_d_discriminant_analysis`.

Required Arguments

int `n_rows` (Input)
    Number of rows of `x` to be processed.

int `n_variables` (Input)
    Number of variables to be used in the discrimination.

float `*x` (Input)
    Array of size `n_rows` by `n_variables` + 1 containing the data. The first `n_variables` columns correspond to the variables, and the last column (column `n_variables`) contains the group numbers. The groups must be numbered 1, 2, ..., `n_groups`.

int `n_groups` (Input)
    Number of groups in the data.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_discriminant_analysis` (*int* `n_rows`, *int* `n_variables`,
    float `*x`, *int* `n_groups`,

IMSLS_X_COL_DIM, *int* x_col_dim,
IMSLS_X_INDICES, *int* igrp, *int* ind[], *int* ifrq, *int* iwt,
IMSLS_METHOD, *int* method,
IMSLS_IDO, *int* ido,
IMSLS_ROWS_ADD,
IMSLS_ROWS_DELETE,
IMSLS_PRIOR_EQUAL,
IMSLS_PRIOR_PROPORTIONAL,
IMSLS_PRIOR_INPUT, *float* prior_input[],
IMSLS_PRIOR_OUTPUT, *float* **prior_output
IMSLS_PRIOR_OUTPUT_USER, *float* prior_output[]
IMSLS_GROUP_COUNTS, *int* **gcounts,
IMSLS_GROUP_COUNTS_USER, *int* gcounts[]
IMSLS_MEANS, *float* **means,
IMSLS_MEANS_USER, *float* means[],
IMSLS_COV, *float* **covariances,
IMSLS_COV_USER, *float* covariances[],
IMSLS_COEF, *float* **coefficients
IMSLS_COEF_USER, *float* coefficients[],
IMSLS_CLASS_MEMBERSHIP, *int* **class_membership,
IMSLS_CLASS_MEMBERSHIP_USER, *int* class_membership[],
IMSLS_CLASS_TABLE, *float* **class_table,
IMSLS_CLASS_TABLE_USER, *float* class_table[],
IMSLS_PROB, *float* **prob,
IMSLS_PROB_USER, *float* prob[],
IMSLS_MAHALANOBIS, *float* **d2,
IMSLS_MAHALANOBIS_USER, *float* d2[],
IMSLS_STATS, *float* **stats,
IMSLS_STATS_USER, *float* stats[],
IMSLS_N_ROWS_MISSING, *int* *nrmiss,
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of array x.
Default: x_col_dim = n_variables + 1

IMSLS_X_INDICES, *int* igrp, *int* ind[], *int* ifrq, *int* iwt (Input)
Each of the four arguments contains indices indicating column numbers of x in which particular types of data are stored. Columns are numbered 0 … x_col_dim – 1.

Parameter igrp contains the index for the column of x in which the group numbers are stored.

Parameter ind contains the indices of the variables to be used in the analysis.

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

Defaults: `igrp` = `n_variables`, `ind[]` = 0, 1, ..., `n_variables` − 1, `ifrq` = −1, and `iwt` = −1

`IMSLS_METHOD`, *int* `method` (Input)
Method of discrimination. The method chosen determines whether linear or quadratic discrimination is used, whether the group covariance matrices are computed (the pooled covariance matrix is always computed), and whether the leaving-out-one or the reclassification method is used to classify each observation.

method	**discrimination method**	**covariances computed**	**classification method**
1	linear	pooled, group	reclassification
2	quadratic	pooled, group	reclassification
3	linear	pooled	reclassification
4	linear	pooled, group	leaving-out-one
5	quadratic	pooled, group	leaving-out-one
6	linear	pooled	leaving-out-one

In the leaving-out-one method of classification, the posterior probabilities are adjusted so as to eliminate the effect of the observation from the sample statistics prior to its classification. In the classification method, the effect of the observation is not eliminated from the classification function.

When optional argument `IMSLS_IDO` is specified, the following rules for mixing methods apply; Methods 1, 2, 4, and 5 can be intermixed, as can methods 3 and 6. Methods 1, 2, 4, and 5 *cannot* be intermixed with methods 3 and 6.

Default: `method` = 1

`IMSLS_IDO`, *int* `ido` (Input)
Processing option. See Comments 3 and 4 for more information.

ido	**Action**
0	This is the only invocation; all the data are input at once. (Default)
1	This is the first invocation with this data; additional calls will be made. Initialization and updating for the `n_rows` observations of `x` will be performed.

ido	Action
2	This is an intermediate invocation; updating for the `n_rows` observations of `x` will be performed.
3	All statistics are updated for the `n_rows` observations. The discriminant functions and other statistics are computed.
4	The discriminant functions are used to classify each of the `n_rows` observations of `x`.
5	The covariance matrices are computed, and workspace is released. No further call to `discriminant_analysis` with `ido` greater than 1 should be made without first calling `discriminant_analysis` with `ido` = 1.
6	Workspace is released. No further calls to `discriminant_analysis` with `ido` greater than 1 should be made without first calling `discriminant_analysis` with `ido` = 1. Invocation with this option is not required if a call has already been made with `ido` = 5.

Default: `ido` = 0

`IMSLS_ROWS_ADD`, *or*
`IMSLS_ROWS_DELETE` (Input)

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

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

`IMSLS_PRIOR_EQUAL`, *or*
`IMSLS_PRIOR_PROPORTIONAL`, *or*
`IMSLS_PRIOR_INPUT`, *float* `prior_input[]` (Input)

By default, (or if `IMSLS_PRIOR_EQUAL` is specified), equal prior probabilities are calculated as 1.0/`n_groups`.

If `IMSLS_PRIOR_PROPORTIONAL` is specified, prior probabilities are calculated to be proportional to the sample size in each group.

If `IMSLS_PRIOR_INPUT` is specified, then array `prior_input` is an array of length `n_groups` containing the prior probabilities for each group, such that the sum of all prior probabilities is equal to 1.0. Prior probabilities are not used if `ido` is equal to 1, 2, 5, or 6.

`IMSLS_PRIOR_OUTPUT`, *float* `**prior_output` (Output)

Address of a pointer to an array of length `n_groups` containing the most recently calculated or input prior probabilities. If `IMSLS_PRIOR_PROPORTIONAL` is specified, every element of `prior_output` is equal to −1 until a call is made with `ido` equal to 0 or 3, at which point the priors are calculated. Note that subsequent calls to discriminant_analysis with IMSLS_PRIOR_PROPORTIONAL specified, and ido not equal to 0 or 3 will result in the elements of prior_output being reset to –1.

IMSLS_PRIOR_OUTPUT_USER, *float* prior_output[] (Output)
: Storage for array prior_output is provided by the user. See IMSLS_PRIOR_OUTPUT.

IMSLS_GROUP_COUNTS, *int* **gcounts (Output)
: Address of a pointer to an integer array of length n_groups containing the number of observations in each group. Array gcounts is updated when ido is equal to 0, 1, or 2.

IMSLS_GROUP_COUNTS_USER, *int* gcounts[] (Output)
: Storage for integer array gcounts is provided by the user. See IMSLS_GROUP_COUNTS.

IMSLS_MEANS, *float* **means (Output)
: Address of a pointer to an array of size n_groups by n_variables. The *i*-th row of means contains the group *i* variable means. Array means is updated when ido is equal to 0, 1, 2, or 5. The means are *unscaled* until a call is made with ido = 5. where the unscaled means are calculated as $\Sigma w_i f_i x_i$ and the scaled means as

$$\frac{\sum w_i f_i x_i}{\sum w_i f_i}$$

where x_i is the value of the *i*-th observation, w_i is the weight of the *i*-th observation, and f_i is the frequency of the *i*-th observation.

IMSLS_MEANS_USER, *float* means[] (Output)
: Storage for array means is provided by the user. See IMSLS_MEANS.

IMSLS_COV, *float* **covariances (Output)
: Address of a pointer to an array of size g by n variables by n_variables containing the within-group covariance matrices (methods 1, 2, 4, and 5 only) as the first *g*-1 matrices, and the pooled covariance matrix as the *g*-th matrix (that is, the first n_variables * n_variables elements comprise the group 1 covariance matrix, the next n_variables * n_variables elements comprise the group 2 covariance, ..., and the last n_variables * n_variables elements comprise the pooled covariance matrix). If method is 3 or 6 then *g* is equal to 1. Otherwise*, g* is equal to n_groups + 1. Argument cov is updated when ido is equal to 0, 1, 2, 3, or 5.

IMSLS_COV_USER, *float* covariances[] (Output)
: Storage for array covariances is provided by the user. See IMSLS_COVARIANCES.

IMSLS_COEF, *float* **coefficients (Output)

Address of a pointer to an array of size n_groups by (n_variables + 1) containing the linear discriminant coefficients. The first column of coefficients contains the constant term, and the remaining columns contain the variable coefficients. Row $i - 1$ of coefficients corresponds to group i, for $i = 1, 2, ...,$ n_variables + 1. Array coefficients are always computed as the linear discriminant function coefficients even when quadratic discrimination is specified.

Array coefficients is updated when ido is equal to 0 or 3.

IMSLS_COEF_USER, *float* coefficients[] (Output)

Storage for array coefficients is provided by the user. See IMSLS_COEFFICIENTS.

IMSLS_CLASS_MEMBERSHIP, *int* **class_membership (Output)

Address of a pointer to an integer array of length n_rows containing the group to which the observation was classified. Array class_membership is updated when ido is equal to 0 or 4.

If an observation has an invalid group number, frequency, or weight when the leaving-out-one method has been specified, then the observation is not classified and the corresponding elements of class_membership (and prob, see IMSLS_PROB) are set to zero.

IMSLS_CLASS_MEMBERSHIP_USER, *int* class_membership[] (Ouput)

Storage for array class_membership is provided by the user. See IMSLS_CLASS_MEMBERSHIP.

IMSLS_CLASS_TABLE, *float* **class_table (Output)

Address of a pointer to an array of size n_groups by n_groups containing the classification table. Array class_table is updated when ido is equal to 0, 1, or 4. Each observation that is classified and has a group number 1.0, 2.0, ..., n_groups is entered into the table. The rows of the table correspond to the known group membership. The columns refer to the group to which the observation was classified. Classification results accumulate with each call to imsls_f_discriminant_analysis with ido equal to 4. For example, if two calls with ido equal to 4 are made, the elements in class_table sum to the total number of valid observations in the two calls.

IMSLS_CLASS_TABLE_USER, *float* class_table[] (Output)

Storage for array class_table is provided by the user. See IMSLS_CLASS_TABLE.

IMSLS_PROB, *float* **prob (Output)

Address of a pointer to an array of size n_rows by n_groups containing the posterior probabilities for each observation. Argument prob is updated when ido is equal to 0 or 4.

IMSLS_PROB_USER, *float* prob[] (Output)

Storage for array prob is provided by the user. See IMSLS_PROB.

IMSLS_MAHALANOBIS, *float* **d2 (Output)

Address of a pointer to an array of size n_groups by n_groups containing the Mahalanobis distances

$$D_{ij}^2$$

between the group means. Argument d2 is updated when ido is equal to 0 or 3.

For linear discrimination, the Mahalanobis distance is computed using the pooled covariance matrix. Otherwise, the Mahalanobis distance

$$D_{ij}^2$$

between group means *i* and *j* is computed using the within covariance matrix for group *i* in place of the pooled covariance matrix.

IMSLS_MAHALANOBIS_USER, *float* d2[] (Output)

Storage for array d2 is provided by the user. See IMSLS_MAHALANOBIS.

IMSLS_STATS, *float* **stats (Output)

Address of a pointer to an array of length 4 + 2 × (n_groups + 1) containing various statistics of interest. Array stats is updated when ido is equal to 0, 1, 3, or 5. The first element of stats is the sum of the degrees of freedom for the within-covariance matrices. The second, third, and fourth elements of stats correspond to the chi-squared statistic, its degrees of freedom, and the probability of a greater chi-squared, respectively, of a test of the homogeneity of the within-covariance matrices (not computed if method is equal to 3 or 6). The fifth through 5 + n_groups elements of stats contain the log of the determinants of each group's covariance matrix (not computed if method is equal to 3 or 6) and of the pooled covariance matrix (element 4 + n_groups). Finally, the last n_groups + 1 elements of stats contain the sum of the weights within each group, and in the last position, the sum of the weights in all groups.

IMSLS_STATS_USER, *float* stats[] (Output)

Storage for array stats is provided by the user. See IMSLS_STATS_USER.

IMSLS_N_ROWS_MISSING, *int* *nrmiss (Output)

Number of rows of data encountered in calls to discriminant_analysis containing missing values (NaN) for the classification, group, weight, and/or frequency variables. If a row of data contains a missing value (NaN) for any of these variables, that row is excluded from the computations.

Array nrmiss is updated when ido is equal to 0, 1, 2, or 3.

Comments

1. Common choices for the Bayesian prior probabilities are given by:
   `prior_input[i]` = 1.0/`n_groups` (equal priors)
   `prior_input[i]` = `gcounts`/`n_rows` (proportional priors)
   `prior_input[i]` = Past history or subjective judgment.
   In all cases, the priors should sum to 1.0.

2. Two passes of the data are made. In the first pass, the statistics required to compute the discriminant functions are obtained (`ido` equal to 1, 2, and 3). In the second pass, the discriminant functions are used to classify the observations. When `ido` is equal to 0, all of the data are memory resident, and both passes are made in one call to `imsls_f_discriminant_analysis`. When `ido` > 0 (optional argument `IMSLS_IDO` is specified), a third call to `imsls_f_discriminant_analysis` involving no data is required with `ido` equal to 5 or 6.

3. Here are a few rules and guidelines for the correct value of `ido` in a series of calls:

   1 Calls with `ido` = 0 or `ido` = 1 may be made at any time, subject to rule 2. These calls indicate that a new analysis is to begin, and therefore allocate memory and destroy all statistics from previous calls.

   2 Each series of calls to `imsls_f_discriminant_analysis` which begins with `ido` = 1 must end with `ido` equal to 5 or 6 to ensure the proper release of workspace, subject to rule 3.

   3 `ido` may not be 4 or 5 before a call with `ido` = 3 has been made.

   4 `ido` may not be 2, 3, 4, 5, or 6
   a) Immediately after a call with `ido` = 0.
   b) Before a call with `ido` = 1 has been made.
   c) Immediately after a call with `ido` equal to 5 or 6 has been made.

The following is a valid sequence of `ido`'s:

ido	Explanation
0	Data Set A: Perform a complete analysis. All data to be used in the analysis must be present in `x`. Since cleanup of workspace is automatic for `ido` = 0, no further calls are necessary.
1	Data Set B: Begin analysis. The `n_rows` observations in `x` are used for initialization.
2	Data Set B: Continue analysis. New observations placed in `x` are added to (or deleted from, see `IMSLS_ROWS_DELETE`) the analysis.

ido	Explanation
2	Data Set B: Continue analysis. `n_rows` new observations placed in `x` are added to (or deleted from, see `IMSLS_ROWS_DELETE`) the analysis.
3	Data Set B: Continue analysis. `n_rows` new observations are added (or deleted) and discriminant functions and other statistics are computed.
4	Data Set B: Classification of each of the `n_rows` observations in the current `x` matrix.
5	Data Set B: End analysis. Covariance matrices are computed and workspace is released. This analysis could also have been ended by choosing `ido` = 6
1	Data Set C: Begin analysis. Note that for this call to be valid the previous call must have been made with `ido` equal to 5 or 6.
3	Data Set C: Continue analysis.
4	Data Set C: Continue analysis.
3	Data Set C: Continue analysis.
6	Data Set C: End analysis.

4. Because of the internal workspace allocation and saved variables, function `imsls_f_discriminant_analysis` must complete the analysis of a data set before beginning processing of the next data set.

Return Value

The return value is void.

Description

Function `imsls_f_discriminant_analysis` performs discriminant function analysis using either linear or quadratic discrimination. The output includes a measure of distance between the groups, a table summarizing the classification results, a matrix containing the posterior probabilities of group membership for each observation, and the within-sample means and covariance matrices. The linear discriminant function coefficients are also computed.

By default (or if optional argument `IMSLS_IDO` is specified with `ido` = 0) all observations are input during one call, a method of operation that has the advantage of simplicity. Alternatively, one or more rows of observations can be input during separate calls. This method does not require that all observations be memory resident, a significant advantage with large data sets. Note, however, that the algorithm requires two passes of the data. During the first pass the discriminant functions are computed while in the second pass, the observations are classified. Thus, with the second method of operation, the data will usually need to be input twice.

Because both methods result in the same operations being performed, the algorithm is discussed as if only a few observations are input during each call. The operations performed during each call depend upon the `ido` parameter.

The `ido` = 1 step is the initialization step. "Private" internally allocated saved variables corresponding to `means`, `class_table`, and `covariances` are initialized to zero, and other program parameters are set (copies of these private variables are written to the corresponding output variables upon return from the function call, assuming `ido` values such that the results are to be returned). Parameters `n_rows`, `x`, and `method` can be changed from one call to the next *within* the two sets {1, 2, 4, 5} and {3, 6} but not *between* these sets when `ido` > 1. That is, do not specify `method` = 1 in one call and `method` = 3 in another call without first making a call with `ido` = 1.

After initialization has been performed in the `ido` = 1 step, the within-group means are updated for all valid observations in `x`. Observations with invalid group numbers are ignored, as are observation with missing values. The *LU* factorization of the covariance matrices are updated by adding (or deleting) observations via Givens rotations.

The `ido` = 2 step is used solely for adding or deleting observations from the model as in the above paragraph.

The `ido` = 3 step begins by adding all observations in `x` to the means and the factorizations of the covariance matrices. It continues by computing some statistics of interest: the linear discriminant functions, the prior probabilities (by default, or if `IMSLS_PROPORTIONAL_PRIORS` is specified), the log of the determinant of each of the covariance matrices, a test statistic for testing that all of the within-group covariance matrices are equal, and a matrix of Mahalanobis distances between the groups. The matrix of Mahalanobis distances is computed via the pooled covariance matrix when linear discrimination is specified; the row covariance matrix is used when the discrimination is quadratic.

Covariance matrices are defined as follows: Let N_i denote the sum of the frequencies of the observations in group i and M_i denote the number of observations in group i. Then, if S_i denotes the within-group i covariance matrix,

$$S_i = \frac{1}{N_i - 1} \sum_{j=1}^{M_i} w_j f_j \left(x_j - \bar{x}\right)\left(x_j - \bar{x}\right)^T$$

Where w_j is the weight of the j-th observation in group i, f_j is the frequency, x_j is the j-th observation column vector (in group i), and $\bar{x}$ denotes the mean vector of the observations in group i. The mean vectors are computed as

$$\bar{x} = (\frac{1}{W_i}) \sum_{j=1}^{M_i} w_j f_j x_j \qquad \text{where } W_i = \sum_{j=1}^{M_i} w_j f_j$$

Given the means and the covariance matrices, the linear discriminant function for group i is computed as:

$$z_i = \ln\left(p_i\right) - 0.5 \bar{x}_i^T S_p^{-1} \bar{x}_i + x^T S_p^{-1} \bar{x}_i$$

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:

$$D_{ij}^2 = \left(\bar{x}_i - \bar{x}_j\right)^T S^{-1} \left(\bar{x}_i - \bar{x}_j\right)$$

Finally, the asymptotic chi-squared test for the equality of covariance matrices is computed as follows (Morrison 1976, p. 252):

$$\gamma = C^{-1} \sum_{i=1}^{k} n_i \left\{ \ln\left(\left|S_p\right|\right) - \ln\left(\left|S_i\right|\right) \right\}$$

where n_i is the number of degrees of freedom in the i-th sample covariance matrix, k is the number of groups, and

$$C^{-1} = \frac{1 - 2p^2 + 3p - 1}{6\left(p+1\right)\left(k-1\right)} \left(\sum_{i=1}^{k} \frac{1}{n_i} - \frac{1}{\sum_j n_j} \right)$$

where p is the number of variables.

When `ido` = 4, the estimated posterior probability of each observation x belonging to group is computed using the prior probabilities and the sample mean vectors and estimated covariance matrices under a multivariate normal assumption. Under quadratic discrimination, the within-group covariance matrices are used to compute the estimated posterior probabilities. The estimated posterior probability of an observation x belonging to group i is

$$\hat{q}_i\left(x\right) = \frac{\exp\left(-0.5 D_i^2\left(x\right)\right)}{\sum_{j=1}^{k} \exp\left(-0.5 D_j^2\left(x\right)\right)}$$

where

$$D_i^2\left(x\right) = \begin{cases} \left(x - \bar{x}_i\right)^T S_i^{-1} \left(x - \bar{x}_i\right) + \ln\left|S_i\right| - 2\ln\left(p_i\right) & \text{METHOD = 1 or 2} \\ \left(x - \bar{x}_i\right)^T S_p^{-1} \left(x - \bar{x}_i\right) - 2\ln\left(p_i\right) & \text{METHOD = 3} \end{cases}$$

For the leaving-out-one method of classification (`method` equal to 4, 5 or 6), the sample mean vector and sample covariance matrices in the formula for

$$D_i^2$$

are adjusted so as to remove the observation x from their computation. For linear discrimination (`method` equal to 1, 2, 4, or 6), the linear discriminant function coefficients are actually used to compute the same posterior probabilities.

Using the posterior probabilities, each observation in *x* is classified into a group; the result is tabulated in the matrix `class_table` and saved in the vector `class_membership`. Matrix `class_table` is not altered at this stage if `x[i][x_group]` (by default, `x_igrp` = 0; see optional argument `IMSLS_INDICES`) contains a group number that is out of range. If the reclassification method is specified, then all observations with no missing values in the `n_variables` classification variables are classified. When the leaving-out-one method is used, observations with invalid group numbers, weights, frequencies, or classification variables are not classified. Regardless of the frequency, a 1 is added (or subtracted) from `class_table` for each row of `x` that is classified and contains a valid group number.

When `method` > 3, adjustment is made to the posterior probabilities to remove the effect of the observation in the classification rule. In this adjustment, each observation is presumed to have a weight of `x[i][iwt]` if `iwt` > −1 (and a weight of 1.0 if `iwt` = −1), and a frequency of 1.0. See Lachenbruch (1975, p. 36) for the required adjustment.

Finally, when `ido` = 5, the covariance matrices are computed from their *LU* factorizations. Internally allocated and saved variables are cleaned up at this step (`ido` equal to 5 or 6).

Example 1

The following example uses liner discrimination with equal prior probabilities on Fisher’s (1936) iris data. This example illustrates the execution of `imsls_f_discriminant_analysis` when one call is made (i.e. using the default of `ido` = 0).

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

main() {
    int   n_groups = 3;
    int   nrow, nvar, ncol, nrmiss;
    float *x, *xtemp;
    float *prior_out, *means, *cov, *coef;
    float *table, *d2, *stats, *prob;
    int   *counts, *cm;
    static int perm[5] = {1, 2, 3, 4, 0};

    /* Retrieve the Fisher Iris Data Set */
    xtemp = imsls_f_data_sets(3, IMSLS_N_OBSERVATIONS, &nrow,
        IMSLS_N_VARIABLES, &ncol, 0);
    nvar = ncol - 1;

    /* Move the group column to end of the the matrix */
    x = imsls_f_permute_matrix(nrow, ncol, xtemp, perm,
        IMSLS_PERMUTE_COLUMNS, 0);
    free(xtemp);

    imsls_f_discriminant_analysis (nrow, nvar, x, n_groups,
        IMSLS_METHOD, 3,
        IMSLS_GROUP_COUNTS, &counts,
        IMSLS_COEF, &coef,
        IMSLS_MEANS, &means,
        IMSLS_STATS, &stats,
        IMSLS_CLASS_MEMBERSHIP, &cm,
        IMSLS_CLASS_TABLE, &table,
        IMSLS_PROB, &prob,
        IMSLS_MAHALANOBIS, &d2,
        IMSLS_COV, &cov,
        IMSLS_PRIOR_OUTPUT, &prior_out,
        IMSLS_N_ROWS_MISSING, &nrmiss,
        IMSLS_PRIOR_EQUAL,
        IMSLS_METHOD, 3, 0);

    imsls_i_write_matrix("Counts", 1, n_groups, counts, 0);
    imsls_f_write_matrix("Coef", n_groups, nvar+1, coef, 0);
    imsls_f_write_matrix("Means", n_groups, nvar, means, 0);
    imsls_f_write_matrix("Stats", 12, 1, stats, 0);
    imsls_i_write_matrix("Membership", 1, nrow, cm, 0);
    imsls_f_write_matrix("Table", n_groups, n_groups, table, 0);
    imsls_f_write_matrix("Prob", nrow, n_groups, prob, 0);
    imsls_f_write_matrix("D2", n_groups, n_groups, d2, 0);
    imsls_f_write_matrix("Covariance", nvar, nvar, cov, 0);
    imsls_f_write_matrix("Prior OUT", 1, n_groups, prior_out, 0);
    printf("\nnrmiss = %3d\n", nrmiss);

    free(means);
    free(stats);
    free(counts);
    free(coef);
    free(cm);
    free(table);
    free(prob);
    free(d2);
    free(prior_out);
    free(cov);
}
```

Output

```
   Counts
  1    2    3
 50   50   50

                              Coef
            1            2            3            4            5
1       -86.3         23.5         23.6        -16.4        -17.4
2       -72.9         15.7          7.1          5.2          6.4
3      -104.4         12.4          3.7         12.8         21.1

                        Means
            1            2            3            4
1       5.006        3.428        1.462        0.246
2       5.936        2.770        4.260        1.326
3       6.588        2.974        5.552        2.026

     Stats
 1         147
 2  ..........
 3  ..........
 4  ..........
 5  ..........
 6  ..........
 7  ..........
 8         -10
 9          50
10          50
11          50
12         150

                             Membership
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2

61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 2  2  2  2  2  2  2  2  2  2  3  2  2  2  2  2  2  2  2  2

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 2  2  2  3  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   3   3   3   3   3   3   3   3   3

148  149  150
  3    3    3

                Table
            1           2           3
1          50           0           0
2           0          48           2
3           0           1          49

                 Prob
              1           2           3
  1       1.000       0.000       0.000
  2       1.000       0.000       0.000
  3       1.000       0.000       0.000
  4       1.000       0.000       0.000
  5       1.000       0.000       0.000
  6       1.000       0.000       0.000
  7       1.000       0.000       0.000
  8       1.000       0.000       0.000
  9       1.000       0.000       0.000
 10       1.000       0.000       0.000
 11       1.000       0.000       0.000
 12       1.000       0.000       0.000
 13       1.000       0.000       0.000
```

```
14       1.000       0.000       0.000
15       1.000       0.000       0.000
16       1.000       0.000       0.000
17       1.000       0.000       0.000
18       1.000       0.000       0.000
19       1.000       0.000       0.000
20       1.000       0.000       0.000
21       1.000       0.000       0.000
22       1.000       0.000       0.000
23       1.000       0.000       0.000
24       1.000       0.000       0.000
25       1.000       0.000       0.000
26       1.000       0.000       0.000
27       1.000       0.000       0.000
28       1.000       0.000       0.000
29       1.000       0.000       0.000
30       1.000       0.000       0.000
31       1.000       0.000       0.000
32       1.000       0.000       0.000
33       1.000       0.000       0.000
34       1.000       0.000       0.000
35       1.000       0.000       0.000
36       1.000       0.000       0.000
37       1.000       0.000       0.000
38       1.000       0.000       0.000
39       1.000       0.000       0.000
40       1.000       0.000       0.000
41       1.000       0.000       0.000
42       1.000       0.000       0.000
43       1.000       0.000       0.000
44       1.000       0.000       0.000
45       1.000       0.000       0.000
46       1.000       0.000       0.000
47       1.000       0.000       0.000
48       1.000       0.000       0.000
49       1.000       0.000       0.000
50       1.000       0.000       0.000
51       0.000       1.000       0.000
52       0.000       0.999       0.001
53       0.000       0.996       0.004
54       0.000       1.000       0.000
55       0.000       0.996       0.004
56       0.000       0.999       0.001
57       0.000       0.986       0.014
58       0.000       1.000       0.000
59       0.000       1.000       0.000
60       0.000       1.000       0.000
61       0.000       1.000       0.000
62       0.000       0.999       0.001
63       0.000       1.000       0.000
64       0.000       0.994       0.006
65       0.000       1.000       0.000
66       0.000       1.000       0.000
67       0.000       0.981       0.019
68       0.000       1.000       0.000
69       0.000       0.960       0.040
70       0.000       1.000       0.000
71       0.000       0.253       0.747
72       0.000       1.000       0.000
```

```
 73       0.000       0.816       0.184
 74       0.000       1.000       0.000
 75       0.000       1.000       0.000
 76       0.000       1.000       0.000
 77       0.000       0.998       0.002
 78       0.000       0.689       0.311
 79       0.000       0.993       0.007
 80       0.000       1.000       0.000
 81       0.000       1.000       0.000
 82       0.000       1.000       0.000
 83       0.000       1.000       0.000
 84       0.000       0.143       0.857
 85       0.000       0.964       0.036
 86       0.000       0.994       0.006
 87       0.000       0.998       0.002
 88       0.000       0.999       0.001
 89       0.000       1.000       0.000
 90       0.000       1.000       0.000
 91       0.000       0.999       0.001
 92       0.000       0.998       0.002
 93       0.000       1.000       0.000
 94       0.000       1.000       0.000
 95       0.000       1.000       0.000
 96       0.000       1.000       0.000
 97       0.000       1.000       0.000
 98       0.000       1.000       0.000
 99       0.000       1.000       0.000
100       0.000       1.000       0.000
101       0.000       0.000       1.000
102       0.000       0.001       0.999
103       0.000       0.000       1.000
104       0.000       0.001       0.999
105       0.000       0.000       1.000
106       0.000       0.000       1.000
107       0.000       0.049       0.951
108       0.000       0.000       1.000
109       0.000       0.000       1.000
110       0.000       0.000       1.000
111       0.000       0.013       0.987
112       0.000       0.002       0.998
113       0.000       0.000       1.000
114       0.000       0.000       1.000
115       0.000       0.000       1.000
116       0.000       0.000       1.000
117       0.000       0.006       0.994
118       0.000       0.000       1.000
119       0.000       0.000       1.000
120       0.000       0.221       0.779
121       0.000       0.000       1.000
122       0.000       0.001       0.999
123       0.000       0.000       1.000
124       0.000       0.097       0.903
125       0.000       0.000       1.000
126       0.000       0.003       0.997
127       0.000       0.188       0.812
128       0.000       0.134       0.866
129       0.000       0.000       1.000
130       0.000       0.104       0.896
131       0.000       0.000       1.000
```

```
132       0.000       0.001       0.999
133       0.000       0.000       1.000
134       0.000       0.729       0.271
135       0.000       0.066       0.934
136       0.000       0.000       1.000
137       0.000       0.000       1.000
138       0.000       0.006       0.994
139       0.000       0.193       0.807
140       0.000       0.001       0.999
141       0.000       0.000       1.000
142       0.000       0.000       1.000
143       0.000       0.001       0.999
144       0.000       0.000       1.000
145       0.000       0.000       1.000
146       0.000       0.000       1.000
147       0.000       0.006       0.994
148       0.000       0.003       0.997
149       0.000       0.000       1.000
150       0.000       0.018       0.982

                 D2
           1           2           3
1        0.0        89.9       179.4
2       89.9         0.0        17.2
3      179.4        17.2         0.0

                   Covariance
           1           2           3           4
1     0.2650      0.0927      0.1675      0.0384
2     0.0927      0.1154      0.0552      0.0327
3     0.1675      0.0552      0.1852      0.0427
4     0.0384      0.0327      0.0427      0.0419

                   Prior OUT
         1           2           3
    0.3333      0.3333      0.3333

nrmiss =   0
```

Example 2

Continuing with Fisher's iris data, the example below computes the quadratic discriminant functions using values of `IDO` greater than 0. In the first loop, all observations are added to the functions, one at a time. In the second loop, each of the observations is classified, one by one, using the leaving-out-one method.

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

main() {
    int   n_groups = 3;
    int   nrow, nvar, ncol, i, nrmiss;
    float *x, *xtemp;
    float *prior_out, *means, *cov, *coef;
    float *table, *d2, *stats, *prob;
    int   *counts, *cm;
    static int perm[5] = {1, 2, 3, 4, 0};
```

```
/* Retrieve the Fisher Iris Data Set */
xtemp = imsls_f_data_sets(3, IMSLS_N_OBSERVATIONS, &nrow,
    IMSLS_N_VARIABLES, &ncol, 0);
nvar = ncol - 1;

/* Move the group column to end of the the matrix */
x = imsls_f_permute_matrix(nrow, ncol, xtemp, perm,
    IMSLS_PERMUTE_COLUMNS, 0);
free(xtemp);

prior_out = (float *) malloc(n_groups*sizeof(float));
counts    = (int *)   malloc(n_groups*sizeof(int));
means     = (float *) malloc(n_groups*nvar*sizeof(float));
cov       = (float *) malloc(nvar*nvar*(ngroups+1)*sizeof(float));
coef      = (float *) malloc(n_groups*(nvar+1)*sizeof(float));
table     = (float *) malloc(n_groups*n_groups*sizeof(float));
d2        = (float *) malloc(n_groups*n_groups*sizeof(float));
stats     = (float *) malloc((4+2*(n_groups+1))*sizeof(float));
cm        = (int *)   malloc(nrow*sizeof(int));
prob      = (float *) malloc(nrow*n_groups*sizeof(float));

/*Initialize Analysis*/
imsls_f_discriminant_analysis (0, nvar, x, n_groups,
     IMSLS_IDO, 1,
     IMSLS_METHOD, 2, 0);

/*Add In Each Observation*/
for (i=0;i<nrow;i=i+1) {
  imsls_f_discriminant_analysis (1, nvar, (x+i*ncol), n_groups,
     IMSLS_IDO, 2, 0);
}

/*Remove observation 0 from the analysis */
imsls_f_discriminant_analysis (1, nvar, (x+0), n_groups,
     IMSLS_ROWS_DELETE,
     IMSLS_IDO, 2, 0);

/*Add observation 0 back into the analysis */
imsls_f_discriminant_analysis (1, nvar, (x+0), n_groups,
     IMSLS_IDO, 2, 0);

/*Compute statistics*/
imsls_f_discriminant_analysis (0, nvar, x, n_groups,
     IMSLS_PRIOR_PROPORTIONAL,
     IMSLS_PRIOR_OUTPUT_USER, prior_out,
     IMSLS_IDO, 3, 0);

imsls_f_write_matrix("Prior OUT", 1, n_groups, prior_out, 0);

/*Classify One observation at a time, using proportional priors*/
for (i=0;i<nrow;i=i+1) {
  imsls_f_discriminant_analysis (1, nvar, (x+i*ncol), n_groups,
     IMSLS_IDO, 4,
     IMSLS_CLASS_MEMBERSHIP_USER, (cm+i),
     IMSLS_PROB_USER, (prob+i*n_groups), 0);
}

/*Compute covariance matrices and release internal workspace*/
```

```
    imsls_f_discriminant_analysis (0, nvar, x, n_groups,
         IMSLS_IDO, 5,
         IMSLS_COV_USER, cov,
         IMSLS_GROUP_COUNTS_USER, counts,
         IMSLS_COEF_USER, coef,
         IMSLS_MEANS_USER, means,
         IMSLS_STATS_USER, stats,
         IMSLS_CLASS_TABLE_USER, table,
         IMSLS_MAHALANOBIS_USER, d2,
         IMSLS_N_ROWS_MISSING, &nrmiss, 0);

    imsls_i_write_matrix("Counts", 1, n_groups, counts, 0);
    imsls_f_write_matrix("Coef", n_groups, nvar+1, coef, 0);
    imsls_f_write_matrix("Means", n_groups, nvar, means, 0);
    imsls_f_write_matrix("Stats", 12, 1, stats, 0);
    imsls_i_write_matrix("Membership", 1, nrow, cm, 0);
    imsls_f_write_matrix("Table", n_groups, n_groups, table, 0);
    imsls_f_write_matrix("Prob", nrow, n_groups, prob, 0);
    imsls_f_write_matrix("D2", n_groups, n_groups, d2, 0);
    imsls_f_write_matrix("Covariance", nvar, nvar, cov, 0);
    printf("\nnrmiss = %3d\n", nrmiss);

    free(means);
    free(stats);
    free(counts);
    free(coef);
    free(cm);
    free(table);
    free(prob);
    free(d2);
    free(prior_out);
    free(cov);

}
```

Output

```
            Prior OUT
        1           2           3
    0.3333      0.3333      0.3333

   Counts
  1    2    3
 50   50   50

                           Coef
           1           2           3           4           5
1      -86.3        23.5        23.6       -16.4       -17.4
2      -72.9        15.7         7.1         5.2         6.4
3     -104.4        12.4         3.7        12.8        21.1

                     Means
           1           2           3           4
1      5.006       3.428       1.462       0.246
2      5.936       2.770       4.260       1.326
3      6.588       2.974       5.552       2.026

     Stats
 1       147.0
 2       143.8
 3        20.0
 4         0.0
 5       -13.1
 6       -10.9
 7        -8.9
 8       -10.0
 9        50.0
10        50.0
11        50.0
12       150.0

                                            Membership
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2

61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 2  2  2  2  2  2  2  2  2  2  3  2  2  2  2  2  2  2  2  2

81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
 2  2  2  3  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115
  2   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131
  3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3

132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147
  3   3   2   3   3   3   3   3   3   3   3   3   3   3   3   3

148 149 150
  3   3   3

              Table
           1           2           3
1         50           0           0
2          0          48           2
3          0           1          49

               Prob
             1           2           3
  1      1.000       0.000       0.000
  2      1.000       0.000       0.000
  3      1.000       0.000       0.000
  4      1.000       0.000       0.000
  5      1.000       0.000       0.000
  6      1.000       0.000       0.000
  7      1.000       0.000       0.000
  8      1.000       0.000       0.000
  9      1.000       0.000       0.000
 10      1.000       0.000       0.000
 11      1.000       0.000       0.000
 12       1.000       0.000       0.000
 13       1.000       0.000       0.000
 14       1.000       0.000       0.000
 15       1.000       0.000       0.000
 16       1.000       0.000       0.000
 17       1.000       0.000       0.000
 18       1.000       0.000       0.000
 19       1.000       0.000       0.000
 20       1.000       0.000       0.000
 21       1.000       0.000       0.000
 22       1.000       0.000       0.000
 23       1.000       0.000       0.000
 24       1.000       0.000       0.000
 25       1.000       0.000       0.000
 26       1.000       0.000       0.000
 27       1.000       0.000       0.000
 28       1.000       0.000       0.000
 29       1.000       0.000       0.000
 30       1.000       0.000       0.000
 31       1.000       0.000       0.000
 32       1.000       0.000       0.000
 33       1.000       0.000       0.000
 34       1.000       0.000       0.000
 35       1.000       0.000       0.000
 36       1.000       0.000       0.000
 37       1.000       0.000       0.000
 38       1.000       0.000       0.000
 39       1.000       0.000       0.000
 40       1.000       0.000       0.000
 41       1.000       0.000       0.000
 42       1.000       0.000       0.000
 43       1.000       0.000       0.000
 44       1.000       0.000       0.000
 45       1.000       0.000       0.000
 46       1.000       0.000       0.000
 47       1.000       0.000       0.000
 48       1.000       0.000       0.000
 49       1.000       0.000       0.000
 50       1.000       0.000       0.000
 51       0.000       1.000       0.000
 52       0.000       1.000       0.000
 53       0.000       0.998       0.002
 54       0.000       0.997       0.003
 55       0.000       0.997       0.003
 56       0.000       0.989       0.011
 57       0.000       0.995       0.005
 58       0.000       1.000       0.000
 59       0.000       1.000       0.000
 60       0.000       0.994       0.006
 61       0.000       1.000       0.000
 62       0.000       0.999       0.001
 63       0.000       1.000       0.000
 64       0.000       0.988       0.012
 65       0.000       1.000       0.000
 66       0.000       1.000       0.000
 67       0.000       0.973       0.027
 68       0.000       1.000       0.000
 69       0.000       0.813       0.187
 70       0.000       1.000       0.000
 71       0.000       0.336       0.664
 72       0.000       1.000       0.000
 73       0.000       0.699       0.301
 74       0.000       0.972       0.028
 75       0.000       1.000       0.000
 76       0.000       1.000       0.000
 77       0.000       0.998       0.002
 78       0.000       0.861       0.139
 79       0.000       0.992       0.008
 80       0.000       1.000       0.000
 81       0.000       1.000       0.000
 82       0.000       1.000       0.000
 83       0.000       1.000       0.000
 84       0.000       0.154       0.846
 85       0.000       0.943       0.057
 86       0.000       0.996       0.004
 87       0.000       0.999       0.001
 88       0.000       0.999       0.001
 89       0.000       1.000       0.000
 90       0.000       0.999       0.001
 91       0.000       0.981       0.019
 92       0.000       0.997       0.003
 93       0.000       1.000       0.000
 94       0.000       1.000       0.000
 95       0.000       0.999       0.001
 96       0.000       1.000       0.000
 97       0.000       1.000       0.000
 98       0.000       1.000       0.000
 99       0.000       1.000       0.000
100       0.000       1.000       0.000
101       0.000       0.000       1.000
102       0.000       0.000       1.000
103       0.000       0.000       1.000
104       0.000       0.006       0.994
105       0.000       0.000       1.000
106       0.000       0.000       1.000
107       0.000       0.004       0.996
108       0.000       0.000       1.000
109       0.000       0.000       1.000
110       0.000       0.000       1.000
111       0.000       0.006       0.994
112       0.000       0.001       0.999
113       0.000       0.000       1.000
114       0.000       0.000       1.000
115       0.000       0.000       1.000
116       0.000       0.000       1.000
117       0.000       0.033       0.967
118       0.000       0.000       1.000
119       0.000       0.000       1.000
120       0.000       0.041       0.959
121       0.000       0.000       1.000
122       0.000       0.000       1.000
123       0.000       0.000       1.000
124       0.000       0.028       0.972
125       0.000       0.001       0.999
126       0.000       0.007       0.993
127       0.000       0.057       0.943
128       0.000       0.151       0.849
129       0.000       0.000       1.000
```

```
130        0.000        0.020        0.980
131        0.000        0.000        1.000
132        0.000        0.009        0.991
133        0.000        0.000        1.000
134        0.000        0.605        0.395
135        0.000        0.000        1.000
136        0.000        0.000        1.000
137        0.000        0.000        1.000
138        0.000        0.050        0.950
139        0.000        0.141        0.859
140        0.000        0.000        1.000
141        0.000        0.000        1.000
142        0.000        0.000        1.000
143        0.000        0.000        1.000
144        0.000        0.000        1.000
145        0.000        0.000        1.000
146        0.000        0.000        1.000
147        0.000        0.000        1.000
148        0.000        0.001        0.999
149        0.000        0.000        1.000
150        0.000        0.061        0.939

                 D2
          1            2            3
1       0.0        323.1        706.1
2     103.2          0.0         17.9
3     168.8         13.8          0.0


                   Covariance
          1            2            3            4
1      0.1242       0.0992       0.0164       0.0103
2      0.0992       0.1437       0.0117       0.0093
3      0.0164       0.0117       0.0302       0.0061
4      0.0103       0.0093       0.0061       0.0111

nrmiss =    0
```

Warning Errors

`IMSLS_BAD_OBS_1`	In call #, row # of the data matrix, “x”, has group number = #. The group number must be an integer between 1.0 and “n_groups” = #, inclusively. This observation will be ignored.
`IMSLS_BAD_OBS_2`	The leaving out one method is specified but this observation does not have a valid group number (Its group number is #.). This observation (row #) is ignored.
`IMSLS_BAD_OBS_3`	The leaving out one method is specified but this observation does not have a valid weight or it does not have a valid frequency. This observation (row #) is ignored.

`IMSLS_COV_SINGULAR_3`	The group # covariance matrix is singular. “stats[1]” cannot be computed. “stats[1]” and “stats[3]” are set to the missing value code (NaN).

Fatal Errors

`IMSLS_BAD_IDO_1`	“ido” = #. Initial allocations must be performed by making a call to discriminant_analysis with “ido” = 1.
`IMSLS_BAD_IDO_2`	“ido” = #. A new analysis may not begin until the previous analysis is terminated with “ido” equal to 5 or 6.
`IMSLS_COV_SINGULAR_1`	The variance-covariance matrix for population number # is singular. The computations cannot continue.
`IMSLS_COV_SINGULAR_2`	The pooled variance-covariance matrix is singular. The computations cannot continue.
`IMSLS_COV_SINGULAR_4`	A variance-covariance matrix is singular. The index of the first zero element is equal to #.

Chapter 10: Survival and Reliability Analysis

Routines

Usage Notes

The functions described in this chapter have primary application in the areas of reliability and life testing, but they may find application in any situation in which analysis of binomial events over time is of interest. Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), Gross and Clark (1975), Lawless (1982), and Chiang (1968) and Tanner and Wong (1984) are references for discussing the models and methods desribed in this chapter.

Function `imsls_f_kaplan_meier_estimates` (page 654) 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` (page 660) computes the parameter estimates in a proportional hazards model.

Function `imsls_f_survival_glm` (page 673) fits any of several generalized linear models for survival data, and `imsls_f_survival_estimates` (page 697) computes estimates of survival probabilities based upon the same models. Function `imsls_f_nonparam_hazard_rate` (page 703) performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.
Function `imsls_f_life_tables` (page 712) computes and (optionally) prints an actuarial table based either upon a cohort followed over time or a cross-section of a population.

kaplan_meier_estimates

Computes Kaplan-Meier estimates of survival probabilities in stratified samples.

Synopsis

#include `<imsls.h>`

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

The type *double* function is `imsls_d_kaplan_meier_estimates`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `ncol` (Input)
Number of columns in `x`.

float `x[]` (Input)
Two-dimensional data array of size `n_observations*ncol`.

Return Value

Pointer to an array of length `n_observations*2`. The first column contains the estimated survival probabilities, and the second column contains Greenwood's estimate of the standard deviation of these probabilities. If the *i*-th observation contains censor codes out of range or if a variable is missing, then the corresponding elements of the return value are set to missing (NaN, not a number). Similarly, if an element in the return value is not defined, then it is set to missing.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_kaplan_meier_estimates` (*int* `n_observations`, *int* `ncol`, *float* `x[]`,
`IMSLS_RETURN_USER`, *float* `table[]`,

IMSLS_PRINT,
IMSLS_X_RESPONSE_COL, *int* irt,
IMSLS_CENSOR_CODES_COL, *int* icen,
IMSLS_FREQ_RESPONSE_COL_COL, *int* ifrq,
IMSLS_STRATUM_NUMBER_COL, *int* igrp,
IMSLS_SORTED,
IMSLS_N_MISSING, *int* *nrmiss,
0)

Optional Arguments

IMSLS_RETURN_USER, *float* table[] (Output)
User supplied storage of an array of length n_observations*2 containing the estimated survival probabilities and their associated standard deviations. See Return Value section.

IMSLS_PRINT, (Input)
Print Kaplan-Meier estimates of survival probabilities in stratified samples.

IMSLS_X_RESPONSE_COL, *int* irt (Input)
Column index for the response times in the data array, x. The interpretation of these times as either right-censored or exact failure times depends on IMSLS_CENSOR_CODES_COL.
Default: irt = 0.

IMSLS_CENSOR_CODES_COL, *int* icen (Input)
Column index for the optional censoring codes in the data array, x. If x[i, icen]= 0, the failure time x[i, irt] is treated as an exact time of failure. Otherwise it is treated as a right-censored time.
Default: It is assumed that there is no censor code column in x. All observations are assumed to be exact failure times.

IMSLS_FREQ_RESPONSE_COL_COL, *int* ifrq (Input)
Column index for the number of responses associated with each row in the data array, x.
Default: It is assumed that there is no frequency response column in x. Each observation in the data array is assumed to be for a single failure.

IMSLS_STRATUM_NUMBER_COL, *int* igrp (Input)
Column index for the stratum number for each observation in the data array, x. Column igrp of x contains a unique value for each stratum in the data. Kaplan-Meier estimates are computed within each stratum.
Default: It is assumed that there is no stratum number column in x. The data is assumed to come from one stratum.

IMSLS_SORTED, (Input)
If this option is used, column irt of x is assumed to be sorted in ascending order within each stratum. Otherwise, a detached sort is conducted prior to analysis. If sorting is performed, all censored

individuals are assumed to follow tied failures.
Default: Column `irt` of `x` is not sorted.

`IMSLS_N_MISSING`, *int* `*nrmiss` (Output)
Number of rows of data in `x` containing missing values.

Description

Function `imsls_f_kaplan_meier_estimates` computes Kaplan-Meier (or product-limit) estimates of survival probabilities for a sample of failure times that can be right censored or exact times. A survival probability $S(t)$ is defined as $1 - F(t)$, where $F(t)$ is the cumulative distribution function of the failure times (t). Greenwood's estimate of the standard errors of the survival probability estimates are also computed. (See Kalbfleisch and Prentice, 1980, pages 13 and 14.)

Let (t_i, δ_i), for $i = 1,\ldots, n$ denote the failure censoring times and the censoring codes for the n observations in a single sample. Here, $t_i = \mathtt{x}_{i-1,\, irt}$ is a failure time if δ_i is 0, where $\delta_i = \mathtt{x}_{i-1,\, icen}$. Also, t_i is a right censoring time if δ_i is 1. Rows in `x` containing values other than 0 or 1 for δ_i are ignored. Let the number of observations in the sample that have not failed by time $s_{(t)}$ be denoted by $n_{(t)}$, where $s_{(t)}$ is an ordered (from smallest to largest) listing of the distinct failure times (censoring times are omitted). Then the Kaplan-Meier estimate of the survival probabilities is a step function, which in the interval from $s_{(t)}$ to $s_{(i+1)}$ (including the lower endpoint) is given by

$$\hat{S}(t) = \prod_{j=1}^{i} \left(\frac{n_{(j)} - d_{(j)}}{n_{(j)}} \right)$$

where $d_{(j)}$ denotes the number of failures occurring at time $s_{(j)}$, and $n_{(\varphi)}$ is the number of observation that have not failed prior to$s_{(j)}$.

Note that one row of X may correspond to more than one failed (or censored) observation when the frequency option is in effect (`ifrq` is specified). The Kaplan-Meier estimate of the survival probability prior to time $s_{(1)}$ is 1.0, while the Kaplan-Meier estimate of the survival probability after the last failure time is not defined.

Greenwood's estimate of the variance of

$$\hat{S}(t)$$

in the interval from $s_{(i)}$ to $s_{(i+1)}$ is given as

$$\text{est. var}(\hat{S}(t)) = \hat{S}^2(t) \sum_{j=1}^{i} \frac{d_{(j)}}{n_{(j)}(n_{(j)} - d_{(j)})}$$

Function `imsls_f_kaplan_meier_estimates` computes the single sample estimates of the survival probabilities for all samples of data included in `x` during a single call. This is accomplished through the `igrp` column of `x`, which if present, must contain a distinct code for each sample of observations. If `igrp` is not specified, there is no grouping column, and all observations are assumed to come from the same sample.

When failures and right-censored observations are tied and the data are to be sorted by `imsls_f_kaplan_meier_estimates` (`IMSLS_SORTED` optional argument is not used), `imsls_f_kaplan_meier_estimates` assumes that the time of censoring for the tied-censored observations is immediately after the tied failure (within the same sample). When the `IMSLS_SORTED` optional argument is used, the data are assumed to be sorted from smallest to largest according to column `irt` of `x` within each stratum. Furthermore, a small increment of time is assumed (theoretically) to elapse between the failed and censored observations that are tied (in the same sample). Thus, when the `IMSLS_SORTED` optional argument is used, the user must sort all of the data in `x` from smallest to largest according to column `irt` (and column `igrp`, if present). By appropriate sorting of the observations, the user can handle censored and failed observations that are tied in any manner desired.

The `IMSLS_PRINT` option prints life tables. One table for each stratum is printed. In addition to the survival probabilities at each failure point, the following is also printed: the number of individuals remaining at risk, Greenwood's estimate of the standard errors for the survival probabilities, and the Kaplan-Meier log-likelihood. The Kaplan-Meier log-likelihood is computed as:

$$\ell = \sum_j d_{(j)} \ln d_{(j)} + (n_{(j)} - d_{(j)}) \ln(n_{(j)} - d_{(j)}) - n_{(j)} \ln n_{(j)}$$

where the sum is with respect to the distinct failure times $s_{(j)}$, $d_{(j)}$.

Example

The following example is taken from Kalbfleisch and Prentice (1980, page 1). The first column in *x* contains the death/censoring times for rats suffering from vaginal cancer. The second column contains information as to which of two forms of treatment were provided, while the third column contains the censoring code. Finally, the fourth column contains the frequency of each observation. The product-limit estimates of the survival probabilities are computed for both groups with one call to `imsls_f_kaplan_meier_estimates`.

Function `imsls_f_kaplan_meier_estimates` could have been called with the `IMSLS_SORTED` optional argument if the censored observations had been sorted with respect to the failure time variable. `IMSLS_PRINT` option is used to print the life tables.

```
#include "imsls.h"

void main ()
{
  int icen = 2, ifrq = 3, igrp = 1, ncol = 4, n_observations = 33;
  float x[] = {
    143, 5, 0, 1,
    164, 5, 0, 1,
    188, 5, 0, 2,
    190, 5, 0, 1,
    192, 5, 0, 1,
    206, 5, 0, 1,
    209, 5, 0, 1,
    213, 5, 0, 1,
    216, 5, 0, 1,
    220, 5, 0, 1,
    227, 5, 0, 1,
    230, 5, 0, 1,
    234, 5, 0, 1,
    246, 5, 0, 1,
    265, 5, 0, 1,
    304, 5, 0, 1,
    216, 5, 1, 1,
    244, 5, 1, 1,
    142, 7, 0, 1,
    156, 7, 0, 1,
    163, 7, 0, 1,
    198, 7, 0, 1,
    205, 7, 0, 1,
    232, 7, 0, 2,
    233, 7, 0, 4,
    239, 7, 0, 1,
    240, 7, 0, 1,
    261, 7, 0, 1,
    280, 7, 0, 2,
    296, 7, 0, 2,
    323, 7, 0, 1,
    204, 7, 1, 1,
    344, 7, 1, 1
  };

  imsls_f_kaplan_meier_estimates (n_observations, ncol, x,
                             IMSLS_PRINT,
                             IMSLS_FREQ_RESPONSE_COL_COL, ifrq,
                             IMSLS_CENSOR_CODES_COL, icen,
                             IMSLS_STRATUM_NUMBER_COL, igrp,
                             0);
}
```

Output

```
              Kaplan Meier Survival Probabilities
                 For Group Value = 5

     Number      Number                 Survival     Estimated
    at risk     Failing        Time  Probability    Std. Error
         19           1         143      0.94737      0.051228

         18           1         164      0.89474      0.070406

         17           2         188      0.78947      0.093529

         15           1         190      0.73684       0.10102

         14           1         192      0.68421       0.10664
         13           1         206      0.63158       0.11066

         12           1         209      0.57895       0.11327

         11           1         213      0.52632       0.11455

         10           1         216      0.47368       0.11455

          8           1         220      0.41447       0.11452

          7           1         227      0.35526       0.11243

          6           1         230      0.29605       0.10816

          5           1         234      0.23684       0.10145

          3           1         246      0.15789      0.093431

          2           1         265     0.078947      0.072792

          1           1         304            0  ............

Total number in group    =      19
Total number failing     =      17
Product Limit Likelihood = -49.1692

                  Kaplan Meier Survival Probabilities
                     For Group Value = 7

      Number      Number                  Survival     Estimated
     at risk     Failing         Time  Probability    Std. Error
         21           1         142      0.95238      0.046471

         20           1         156      0.90476      0.064056

         19           1         163      0.85714       0.07636

         18           1         198      0.80952      0.085689

         16           1         205      0.75893      0.094092

         15           2         232      0.65774       0.10529

         13           4         233      0.45536       0.11137

          9           1         239      0.40476       0.10989

          8           1         240      0.35417       0.10717

          7           1         261      0.30357       0.10311

          6           2         280      0.20238      0.090214

          4           2         296      0.10119      0.067783

          2           1         323     0.050595      0.049281

Total number in group    =      21
Total number failing     =      19
Product Limit Likelihood = -50.4277
```

prop_hazards_gen_lin

Analyzes survival and reliability data using Cox's proportional hazards model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_prop_hazards_gen_lin` (*int* `n_observations`,
int `n_columns`, *float* `x[]`, *int* `nef`, *int* `n_var_effects[]`,
int `indices_effects[]`, *int* `max_class`, *int* `*ncoef`, ..., 0)

The type *double* function is `imsls_d_prop_hazards_gen_lin`.

Required Arguments

int `n_observations` (Input)
Number of observations.

int `n_columns` (Input)
Number of columns in `x`.

float `x[]` (Input)
Array of length `n_observations` * `n_columns` containing the data. When optional argument `itie` = 1, the observations in `x` must be grouped by stratum and sorted from largest to smallest failure time within each stratum, with the strata separated.

int `nef` (Input)
Number of effects in the model. In addition to effects involving classification variables, simple covariates and the product of simple covariates are also considered effects.

int `n_var_effects[]` (Input)
Array of length `nef` containing the number of variables associated with each effect in the model.

int `indices_effects[]` (Input)
Index array of length `n_var_effects[0]` + … + `n_var_effects[nef-1]` containing the column indices of `x` associated with each effect. The first `n_var_effects[0]` elements of `indices_effects` contain the column indices of `x` for the variables in the first effect. The next `n_var_effects[1]` elements in `indices_effects` contain the column indices for the second effect, etc.

int `max_class` (Input)
An upper bound on the total number of different values found among the

classification variables in `x`. For example, if the model consisted of two class variables, one with the values {1, 2, 3, 4} and a second with the values {0, 1}, then then the total number of different classification values is 4+2=6, and max_class >= 6.

int `*ncoef` (Output)
Number of estimated coefficients in the model.

Return Value

Pointer to an array of length `ncoef`*4, `coef`, containing the parameter estimates and associated statistics.

Column	Statistic
1	Coefficient estimate $\hat{\beta}$
2	Estimated standard deviation of the estimated coefficient.
3	Asymptotic normal score for testing that the coefficient is zero against the two-sided alternative.
4	*p*-value associated with the normal score in column 3.

Synopsis with Optional Arguments

#include `<imsls.h>`

```
float * imsls_f_prop_hazards_gen_lin (int n_observations,
        int n_columns, float x[], int nef, int n_var_effects[],
        int indices_effects[], int max_class, int *ncoef,
        IMSLS_RETURN_USER, float cov[],
        IMSLS_PRINT_LEVEL, int iprint,
        IMSLS_MAX_ITERATIONS, int max_iterations,
        IMSLS_CONVERGENCE_EPS, float eps,
        IMSLS_RATIO, float ratio,
        IMSLS_X_RESPONSE_COL, int irt,
        IMSLS_CENSOR_CODES_COL, int icen,
        IMSLS_STRATIFICATION_COL, int istrat,
        IMSLS_CONSTANT_COL, int ifix,
        IMSLS_FREQ_RESPONSE_COL, int ifrq,
        IMSLS_TIES_OPTION, int itie,
        IMSLS_MAXIMUM_LIKELIHOOD, float algl,
        IMSLS_N_MISSING, int *nrmiss,
        IMSLS_STATISTICS, float **case,
        IMSLS_STATISTICS_USER, float case[],
        IMSLS_X_MEAN, float **xmean,
        IMSLS_X_MEAN_USER, float xmean[],
        IMSLS_VARIANCE_COVARIANCE_MATRIX, float **cov,
        IMSLS_VARIANCE_COVARIANCE_MATRIX_USER, float cov[],
        IMSLS_INITIAL_EST_INPUT, float in_coef[],
        IMSLS_UPDATE, float **gr,
        IMSLS_UPDATE_USER, float gr[],
        IMSLS_DUMP, int n_class_var, int index_class_var[],
        IMSLS_STRATUM_NUMBER, int **igrp,
        IMSLS_STRATUM_NUMBER_USER, int igrp[],
        IMSLS_CLASS_VARIABLES, int **n_class_values,
               float **class_values,
        IMSLS_CLASS_VARIABLES_USER, int n_class_values[],
               float class_values[],
        0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *float* `coef[]` (Output)
If specified, coef is an array of length `ncoef`*4 containing the parameter estimates and associated statistics. See Return Value.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option. Default: `iprint` = 0.

Iprint	**Action**
0	No printing is performed.
1	Printing is performed, but observational statistics are not printed.
2	All output statistics are printed.

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations. `max_iterations` = 30 will usually be sufficient. Use `max_iterations` = 0 to compute the Hessian and gradient, stored in `cov` and `gr`, at the initial estimates. When `max_iterations` = 0, `IMSLS_INITIAL_EST_INPUT` must be used.
Default: `max_iterations` = 30.

`IMSLS_CONVERGENCE_EPS`, *float* `eps` (Input)
Convergence criterion. Convergence is assumed when the relative change in `algl` from one iteration to the next is less than `eps`. If `eps` is zero, `eps` = 0.0001 is assumed.
Default: `eps` = 0.0001.

`IMSLS_RATIO`, *float* `ratio` (Input)
Ratio at which a stratum is split into two strata.
Default: `ratio` = 1000.0.
Let

$$r_k = \exp(z_k \hat{\beta} + w_k)$$

be the observation proportionality constant, where z_k is the design row vector for the k-th observation and w_k is the optional fixed parameter specified by $x_{k,\,\mathtt{ifix}}$. Let $r_{\min}$ be the minimum value r_k in a stratum, where, for failed observations, the minimum is over all times less than or equal to the time of occurrence of the k-th observation. Let $r_{\max}$ be the maximum value of r_k for the remaining observations in the group. Then, if $r_{\min}$ > `ratio` $r_{\max}$, the observations in the group are divided into two groups at *k*. `ratio` = 1000 is usually a good value. Set `ratio` = –1.0 if no division into strata is to be made.

`IMSLS_X_RESPONSE_COL`, *int* `irt` (Input)
Column index in `x` containing the response variable. For point observations, $x_{i,\,irt}$ contains the time of the *i*-th event. For right-censored observations, $x_{i,\,irt}$ contains the right-censoring time. Note that because `imsls_f_prop_hazards_gen_lin` only uses the order of the events, negative "times" are allowed.
Default: `irt` = 0.

`IMSLS_CENSOR_CODES_COL`, *int* `icen` (Input)
Column index in `x` containing the censoring code for each observation.
Default: A censoring code of 0 is assumed for all observations.

$x_{i,icen}$	**Censoring**
0	Exact censoring time $x_{i,\,irt}$.
1	Right censored. The exact censoring time is greater than $x_{i,\,irt}$.

`IMSLS_STRATIFICATION_COL`, *int* `istrat` (Input)
Column number in `x` containing the stratification variable. Column `istrat` in `x` contains a unique number for each stratum. The risk set for an observation is determined by its stratum.
Default: All observations are considered to be in one stratum.

`IMSLS_CONSTANT_COL`, *int* `ifix` (Input)
Column index in `x` containing a constant, w_i, to be added to the linear response. The linear response is taken to be $w_i + z_i\hat{\beta}$
where w_i is the observation constant, z_i is the observation design row vector, and $\hat{\beta}$ is the vector of estimated parameters. The "fixed" constant allows one to test hypotheses about parameters via the log-likelihoods.
Default: w_i is assumed to be 0 for all observations.

`IMSLS_FREQ_RESPONSE_COL`, *int* `ifrq` (Input)
Column index in `x` containing the number of responses for each observation.
Default: A response frequency of 1 for each observation is assumed.

`IMSLS_TIES_OPTION`, *int* `itie` (Input)
Method for handling ties. Default: `itie` = 0.

Itie	Method
0	Breslow's approximate method.
1	Failures are assumed to occur in the same order as the observations input in x. The observations in x must be sorted from largest to smallest failure time within each stratum, and grouped by stratum. All observations are treated as if their failure/censoring times were distinct when computing the log-likelihood.

IMSLS_MAXIMUM_LIKELIHOOD, *float* *algl (Output)
The maximized log-likelihood.

IMSLS_N_MISSING, *int* *nrmiss (Output)
Number of rows of data in X that contain missing values in one or more columns irt, ifrq, ifix, icen, istrat, index_class_var, or indices_effects of x.

IMSLS_STATISTICS, *float* **case (Output)
Address of a pointer to an array of length n_observations * 5 containing the case statistics for each observation.

Column	Statistic
1	Estimated survival probability at the observation time.
2	Estimated observation influence or leverage.
3	A residual estimate.
4	Estimated cumulative baseline hazard rate.
5	Observation proportionality constant.

IMSLS_STATISTICS_USER, *float* case[] (Output)
Storage for case is provided by the user. See IMSLS_STATISTICS.

IMSLS_X_MEAN, *float* **xmean (Output)
Address of a pointer to an array of length ncoef containing the means of the design variables.

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

IMSLS_VARIANCE_COVARIANCE_MATRIX, *float* **cov (Output)
Address of a pointer to an array of length ncoef*ncoef containing the estimated asymptotic variance-covariance matrix of the parameters. For max_iterations = 0, the return value is the inverse of the Hessian of the negative of the log-likelihood, computed at the estimates input in in_coef.

IMSLS_VARIANCE_COVARIANCE_MATRIX_USER, *float* cov[] (Output)
Storage for cov is provided by the user. See IMSLS_VARIANCE_COVARIANCE_MATRIX.

IMSLS_INITIAL_EST_INPUT, *float* *in_coef (Input)
An array of length ncoef containing the initial estimates on input to prop_hazards_gen_lin.
Default: all initial estimates are taken to be 0.

IMSLS_UPDATE, *float* **gr (Output)
Address of a pointer to an array of length ncoef containing the last parameter updates (excluding step halvings). For max_iterations = 0, gr contains the inverse of the Hessian times the gradient vector computed at the estimates input in in_coef.

IMSLS_UPDATE_USER, *float* gr[] (Output)
Storage for gr is provided by the user. See IMSLS_UPDATE.

IMSLS_DUMP, *int* n_class_var, *int* index_class_var[] (Input)
Variable n_class_var is the number of classification variables. Dummy variables are generated for classification variables using the dummy_method = IMSLS_LEAVE_OUT_LAST of the IMSLS_DUMMY option of imsls_f_regressors_for_glm function (see Chapter 2, Regression). Argument index_class_var is an index array of length n_class_var containing the column numbers of x that are the classification variables. (if n_class_var is is equal to zero, index_class_var is not used).
Default: n_class_var = 0.

IMSLS_STRATUM_NUMBER, *int* **igrp (Output)
Address of a pointer to an array of length n_observations giving the stratum number used for each observation. If ratio is not −1.0, additional "strata" (other than those specified by column istrat of x) may be generated. igrp also contains a record of the generated strata. See the description section for more detail.

IMSLS_STRATUM_NUMBER_USER, *int* igrp[] (Output)
Storage for igrp is provided by the user. See IMSLS_STRATUM_NUMBER.

IMSLS_CLASS_VARIABLES, *int* **n_class_values, *float* **class_values (Output)
n_class_values is an address of a pointer to an array of length n_class_var containing the number of values taken by each classification variable. n_class_values[*i*] is the number of distinct values for the *i*-th classification variable. class_values is an address of a pointer to an array of length n_class_values[0] + n_class_values[1] + … + n_class_values[n_class_var-1] containing the distinct values of the classification variables. The first n_class_values[0] elements of class_values contain the values

for the first classification variable, the next `n_class_values[1]` elements contain the values for the second classification variable, etc.

`IMSLS_CLASS_VARIABLES_USER`, *int* `n_class_values[]`, *float* `class_values[]` (Output)
Storage for `n_class_values` and `class_values` is provided by the user. The length of `class_values` will not be known in advance, use `max_class` as the maximum length of `class_values`. See `IMSLS_CLASS_VARIABLES`.

Description

Function `imsls_f_prop_hazards_gen_lin` computes parameter estimates and other statistics in Proportional Hazards Generalized Linear Models. These models were first proposed by Cox (1972). Two methods for handling ties are allowed in `imsls_f_prop_hazards_gen_lin`. Time-dependent covariates are not allowed. The user is referred to Cox and Oakes (1984), Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), or Lawless (1982), among other texts, for a thorough discussion of the Cox proportional hazards model.

Let $\lambda(t, z_i)$ represent the hazard rate at time t for observation number i with covariables contained as elements of row vector z_i. The basic assumption in the proportional hazards model (the proportionality assumption) is that the hazard rate can be written as a product of a time varying function $\lambda_0(t)$, which depends only on time, and a function $f(z_i)$, which depends only on the covariable values. The function $f(z_i)$ used in `imsls_f_prop_hazards_gen_lin` is given as $f(z_i) = \exp(w_i + \beta z_i)$ where w_i is a fixed constant assigned to the observation, and β is a vector of coefficients to be estimated. With this function one obtains a hazard rate $\lambda(t, z_i) = \lambda_0(t) \exp(w_i + \beta z_i)$. The form of $\lambda_0(t)$ is not important in proportional hazards models.

The constants w_i may be known theoretically. For example, the hazard rate may be proportional to a known length or area, and the w_i can then be determined from this known length or area. Alternatively, the w_i may be used to fix a subset of the coefficients β (say, β_1) at specified values. When w_i is used in this way, constants $w_i = \beta_1 z_{1i}$ are used, while the remaining coefficients in β are free to vary in the optimization algorithm. If user-specified constants are not desired, the user should set `ifix` to 0 so that $w_i = 0$ will be used.

With this definition of $\lambda(t, z_i)$, the usual partial (or marginal, see Kalbfleisch and Prentice (1980)) likelihood becomes

$$L = \prod_{i=1}^{n_d} \frac{\exp(w_i + \beta z_i)}{\sum_{j \in R(t_i)} \exp(w_j + \beta z_j)}$$

where $R(t_i)$ denotes the set of indices of observations that have not yet failed at time t_i (the risk set), t_i denotes the time of failure for the i-th observation, n_d is the total number of observations that fail. Right-censored observations (i.e., observations that are known to have survived to time t_i, but for which no time of failure is known) are incorporated into the likelihood through the risk set $R(t_i)$. Such observations never appear in the numerator of the likelihood. When `itie` = 0, all observations that are censored at time t_i are not included in $R(t_i)$, while all observations that fail at time t_i are included in $R(t_i)$.

If it can be assumed that the dependence of the hazard rate upon the covariate values remains the same from stratum to stratum, while the time-dependent term, $\lambda_0(t)$, may be different in different strata, then `imsls_f_prop_hazards_gen_lin` allows the incorporation of strata into the likelihood as follows. Let k index the m = `istrat` strata. Then, the likelihood is given by

$$L_s = \prod_{k=1}^{m}\left[\prod_{i=1}^{n_k}\frac{\exp(w_{ki}+\beta z_{ki})}{\sum_{j\in R(t_{ki})}\exp(w_{kj}+\beta z_{kj})}\right]$$

In `imsls_f_prop_hazards_gen_lin`, the log of the likelihood is maximized with respect to the coefficients β. A quasi-Newton algorithm approximating the Hessian via the matrix of sums of squares and cross products of the first partial derivatives is used in the initial iterations (the “Q-N” method in the output). When the change in the log-likelihood from one iteration to the next is less than `100*eps`, Newton-Raphson iteration is used (the “N-R” method). If, during any iteration, the initial step does not lead to an increase in the log-likelihood, then step halving is employed to find a step that will increase the log-likelihood.

Once the maximum likelihood estimates have been computed, `imsls_f_prop_hazards_gen_lin` computes estimates of a probability associated with each failure. Within stratum k, an estimate of the probability that the i-th observation fails at time t_i given the risk set $R(t_{ki})$ is given by

$$p_{ki} = \frac{\exp(w_{ki}+z_{ki}\beta)}{\sum_{j\in R(t_{ki})}\exp(w_{kj}+z_{kj}\beta)}$$

A diagnostic “influence” or “leverage” statistic is computed for each noncensored observation as:

$$l_{ki} = -g'_{ki}H_s^{-1}g'_{ki}$$

where H_s is the matrix of second partial derivatives of the log-likelihood, and

$$g'_{ki}$$

is computed as:

$$g'_{ki} = z_{ki} - \frac{z_{ki}\exp(w_{ki}+z_{ki}\beta)}{\sum_{j\in R(t_{ki})}\exp(w_{kj}+z_{kj}\beta)}$$

Influence statistics are not computed for censored observations.

A "residual" is computed for each of the input observations according to methods given in Cox and Oakes (1984, page 108). Residuals are computed as

$$r_{ki} = \exp(w_{ki} + z_{ki}\hat{\beta}) \sum_{j \in R(t_{ki})} \frac{d_{kj}}{\sum_{l \in R(t_{kj})} \exp(w_{kl} + z_{kl}\hat{\beta})}$$

where d_{kj} is the number of tied failures in group k at time t_{kj}. Assuming that the proportional hazards assumption holds, the residuals should approximate a random sample (with censoring) from the unit exponential distribution. By subtracting the expected values, centered residuals can be obtained. (The j-th expected order statistic from the unit exponential with censoring is given as

$$e_j = \sum_{l \le j} \tfrac{1}{h-l+1}$$

where h is the sample size, and censored observations are not included in the summation.)

An estimate of the cumulative baseline hazard within group k is given as

$$\hat{H}_{k0}(t_{ik}) = \sum_{t_{kj} \le t_{ki}} \frac{d_{kj}}{\sum_{l \in R(t_{kj})} \exp(w_{kl} + z_{kl}\hat{\beta})}$$

The observation proportionality constant is computed as

$$\exp(w_{ki} + z_{ki}\hat{\beta})$$

Programming Notes

1. The covariate vectors z_{ki} are computed from each row of the input matrix `x` via function `imsls_f_regressors_for_glm` (see Chapter 2, Regression). Thus, class variables are easily incorporated into the z_{ki}. The reader is referred to the document for `imsls_f_regressors_for_glm` in the regression chapter for a more detailed discussion.
   Note that `imsls_f_prop_hazards_gen_lin` calls `imsls_f_regressors_for_glm` with `dummy_method` = `IMSLS_LEAVE_OUT_LAST` of the `IMSLS_DUMMY` option.

2. The average of each of the explanatory variables is subtracted from the variable prior to computing the product $z_{ki}\beta$. Subtraction of the mean values has no effect on the computed log-likelihood or the estimates since the constant term occurs in both the numerator and denominator of the likelihood. Subtracting the mean values does help to avoid invalid exponentiation in the algorithm and may also speed convergence.

3. Function `imsls_f_prop_hazards_gen_lin` allows for two methods of handling ties. In the first method (`itie` = 1), the user is allowed to break ties in any manner desired. When this method is used, it is assumed that the user has sorted the rows in `x` from largest to smallest with respect to the failure/censoring times $\texttt{x}_{i,\,\texttt{irt}}$ within each stratum (and

across strata), with tied observations (failures or censored) broken in the manner desired. The same effect can be obtained with `itie` = 0 by adding (or subtracting) a small amount from each of the tied observations failure/ censoring times $t_i = x_{i,\,\texttt{irt}}$ 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 ρ = `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:

$$\ln(\lambda) = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \alpha_i + \gamma_j$$

where α_i and γ_j correspond to dummy variables generated from column indices 5 and 6 of `x`, respectively, x_1 corresponds to column index 2, x_2 corresponds to column index 3, and x_3 corresponds to column index 4 of `x`.

```
#include "imsls.h"

#define NOBS 40
#define NCOL 7
#define NCLVAR 2
#define NEF 5

void main ()
{
  int icen = 1, iprint = 2, maxcl = 6, ncoef;
  int indef[NEF] = { 2, 3, 4, 5, 6 };
  int nvef[NEF] = { 1, 1, 1, 1, 1 };
  int indcl[NCLVAR] = { 5, 6 };
  float *coef, ratio = 10000.0;
  float x[NOBS * NCOL] = {
    411, 0, 7, 64, 5, 1, 0,
    126, 0, 6, 63, 9, 1, 0,
    118, 0, 7, 65, 11, 1, 0,
    92, 0, 4, 69, 10, 1, 0,
    8, 0, 4, 63, 58, 1, 0,
    25, 1, 7, 48, 9, 1, 0,
    11, 0, 7, 48, 11, 1, 0,
    54, 0, 8, 63, 4, 2, 0,
    153, 0, 6, 63, 14, 2, 0,
    16, 0, 3, 53, 4, 2, 0,
    56, 0, 8, 43, 12, 2, 0,
    21, 0, 4, 55, 2, 2, 0,
    287, 0, 6, 66, 25, 2, 0,
    10, 0, 4, 67, 23, 2, 0,
    8, 0, 2, 61, 19, 3, 0,
    12, 0, 5, 63, 4, 3, 0,
    177, 0, 5, 66, 16, 4, 0,
    12, 0, 4, 68, 12, 4, 0,
    200, 0, 8, 41, 12, 4, 0,
    250, 0, 7, 53, 8, 4, 0,
    100, 0, 6, 37, 13, 4, 0,
    999, 0, 9, 54, 12, 1, 1,
    231, 1, 5, 52, 8, 1, 1,
    991, 0, 7, 50, 7, 1, 1,
    1, 0, 2, 65, 21, 1, 1,
    201, 0, 8, 52, 28, 1, 1,
    44, 0, 6, 70, 13, 1, 1,
    15, 0, 5, 40, 13, 1, 1,
    103, 1, 7, 36, 22, 2, 1,
    2, 0, 4, 44, 36, 2, 1,
    20, 0, 3, 54, 9, 2, 1,
    51, 0, 3, 59, 87, 2, 1,
    18, 0, 4, 69, 5, 3, 1,
    90, 0, 6, 50, 22, 3, 1,
    84, 0, 8, 62, 4, 3, 1,
    164, 0, 7, 68, 15, 4, 1,
    19, 0, 3, 39, 4, 4, 1,
    43, 0, 6, 49, 11, 4, 1,
    340, 0, 8, 64, 10, 4, 1,
    231, 0, 7, 67, 18, 4, 1
  };

  coef = imsls_f_prop_hazards_gen_lin (NOBS, NCOL, x, NEF,
                              nvef, indef, maxcl, &ncoef,
                              IMSLS_PRINT_LEVEL, iprint,
                              IMSLS_CENSOR_CODES_COL, icen,
                              IMSLS_RATIO, ratio,
                              IMSLS_DUMMY, NCLVAR, &indcl[0], 0);
}
```

Output

```
                    Initial Estimates
      1         2         3         4         5         6         7
 0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000

Method  Iteration  Step size  Maximum scaled     Log
                              coef. update      likelihood
  Q-N        0                                     -102.4
  Q-N        1      1.0000          0.5034         -91.04
  Q-N        2      1.0000          0.5782         -88.07
  N-R        3      1.0000          0.1131         -87.92
  N-R        4      1.0000         0.06958         -87.89
  N-R        5      1.0000       0.0008145         -87.89

Log-likelihood                -87.88778

                 Coefficient Statistics
    Coefficient      Standard    Asymptotic    Asymptotic
                        error   z-statistic       p-value
1        -0.585         0.137        -4.272         0.000
2        -0.013         0.021        -0.634         0.526
3         0.001         0.012         0.064         0.949
4        -0.367         0.485        -0.757         0.449
5        -0.008         0.507        -0.015         0.988
```

```
6         1.113         0.633         1.758         0.079
7         0.380         0.406         0.936         0.349

                  Asymptotic Coefficient Covariance
              1             2             3             4             5
1       0.01873      0.000253     0.0003345      0.005745       0.00975
2                   0.0004235    -4.12e-005     -0.001663    -0.0007954
3                                 0.0001397     0.0008111     -0.001831
4                                                   0.235       0.09799
5                                                                0.2568

              6             7
1      0.004264      0.002082
2     -0.003079     -0.002898
3     0.0005995      0.001684
4        0.1184       0.03735
5        0.1253      -0.01944
6        0.4008       0.06289
7                     0.1647

                             Case Analysis
      Survival     Influence      Residual    Cumulative         Prop.
   Probability                                    hazard      constant
 1        0.00          0.04          2.05          6.10          0.34
 2        0.30          0.11          0.74          1.21          0.61
 3        0.34          0.12          0.36          1.07          0.33
 4        0.43          0.16          1.53          0.84          1.83
 5        0.96          0.56          0.09          0.05          2.05
 6        0.74  ............          0.13          0.31          0.42
 7        0.92          0.37          0.03          0.08          0.42
 8        0.59          0.26          0.14          0.53          0.27
 9        0.26          0.12          1.20          1.36          0.88
10        0.85          0.15          0.97          0.17          5.76
11        0.55          0.31          0.21          0.60          0.36
12        0.74          0.21          0.96          0.31          3.12
13        0.03          0.06          3.02          3.53          0.86
14        0.94          0.09          0.17          0.06          2.71
15        0.96          0.16          1.31          0.05         28.89
16        0.89          0.23          0.59          0.12          4.82
17        0.18          0.09          2.62          1.71          1.54
18        0.89          0.19          0.33          0.12          2.68
19        0.14          0.23          0.72          1.96          0.37
20        0.05          0.09          1.66          2.95          0.56
21        0.39          0.22          1.17          0.94          1.25
22        0.00          0.00          1.73         21.11          0.08
23        0.08  ............          2.19          2.52          0.87
24        0.00          0.00          2.46          8.89          0.28
25        0.99          0.31          0.05          0.01          4.28
26        0.11          0.17          0.34          2.23          0.15
27        0.66          0.25          0.16          0.41          0.38
28        0.87          0.22          0.15          0.14          1.02
29        0.39  ............          0.45          0.94          0.48
30        0.98          0.25          0.06          0.02          2.53
31        0.77          0.26          1.03          0.26          3.90
32        0.63          0.35          1.80          0.46          3.88
33        0.82          0.26          1.06          0.19          5.47
34        0.47          0.26          1.65          0.75          2.21
35        0.51          0.32          0.39          0.67          0.58
```

```
36          0.22          0.18          0.49          1.53          0.32
37          0.80          0.26          1.08          0.23          4.77
38          0.70          0.16          0.26          0.36          0.73
39          0.01          0.23          0.87          4.66          0.19
40          0.08          0.20          0.81          2.52          0.32

                           Last Coefficient Update
          1             2            3            4            5             6
-1.296e-008   2.269e-009  -5.894e-009  -4.782e-007  -1.787e-007   1.509e-007

          7
 4.327e-008

                               Covariate Means
          1             2            3            4            5             6
       5.65         56.58        15.65         0.35         0.28          0.13

          7
       0.53

Distinct Values For Each Class Variable
Variable 1:          1           2           3           4

Variable 2:          0           1

                     Stratum Numbers For Each Observation
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
20
 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1

21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
40
 1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
1

Number of Missing Values          0
```

survival_glm

Analyzes censored survival data using a generalized linear model.

Synopsis

#include <imsls.h>

int imsls_f_survival_glm (*int* n_observations, *int* n_class,
    int n_continuous, *int* model, *float* x[], ..., 0)

The type *double* function is imsls_d_survival_glm.

Required Arguments

int n_observations (Input)
    Number of observations.

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

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

int `model` (Input)
Argument `model` specifies the model used to analyze the data.

model	PDF of the Response Variable
0	Exponential
1	Linear hazard
2	Log-normal
3	Normal
4	Log-logistic
5	Logistic
6	Log least extreme value
7	Least extreme value
8	Log extreme value
9	Extreme value
10	Weibull

See the “Description” section for more information about these models.

float `x[]` (Input)
Array of size `n_observations` by (`n_class` + `n_continuous`) + *m* containing data for the independent variables, dependent variable, and optional parameters.

The columns must be ordered such that the first `n_class` columns contain data for the class variables, the next `n_continuous` columns contain data for the continuous variables, and the next column contains the response variable. The final (and optional) *m* – 1 columns contain the optional parameters.

Return Value

An integer value indicating the number of estimated coefficients in the model.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `imsls_f_survival_glm` (*int* `n_observations`, *int* `n_class`,
int `n_continuous`, *int* `model`, *float* `x[]`,
`IMSLS_X_COL_CENSORING`, *int* `icen`, *int* `ilt`, *int* `irt`,
`IMSLS_X_COL_DIM`, *int* `x_col_dim`,

IMSLS_X_COL_FREQUENCIES, *int* ifrq,
IMSLS_X_COL_FIXED_PARAMETER, *int* ifix,
IMSLS_X_COL_VARIABLES, *int* iclass[], *int* icontinuous[],
    int iy
IMSLS_EPS, *float* eps,
IMSLS_MAX_ITERATIONS, *int* max_iterations,
IMSLS_INTERCEPT,
IMSLS_NO_INTERCEPT,
IMSLS_INFINITY_CHECK, *int* lp_max
IMSLS_NO_INFINITY_CHECK
IMSLS_EFFECTS, *int* n_effects, *int* n_var_effects[],
    int indices_effects,
IMSLS_INITIAL_EST_INTERNAL,
IMSLS_INITIAL_EST_INPUT, *int* n_coef_input,
    float estimates[],
IMSLS_MAX_CLASS, *int* max_class,
IMSLS_CLASS_INFO, *int* **n_class_values,
    float **class_values,
IMSLS_CLASS_INFO_USER, *int* n_class_values[],
    float class_values[],
IMSLS_COEF_STAT, *float* **coef_statistics,
IMSLS_COEF_STAT_USER, *float* coef_statistics[],
IMSLS_CRITERION, *float* *criterion,
IMSLS_COV, *float* **cov,
IMSLS_COV_USER, *float* cov[],
IMSLS_MEANS, *float* **means,
IMSLS_MEANS_USER, *float* means[],
IMSLS_CASE_ANALYSIS, *float* **case_analysis,
IMSLS_CASE_ANALYSIS_USER, *float* case_analysis[],
IMSLS_LAST_STEP, *float* **last_step,
IMSLS_LAST_STEP_USER, *float* last_step[],
IMSLS_OBS_STATUS, *int* **obs_status,
IMSLS_OBS_STATUS_USER, *int* obs_status[],
IMSLS_ITERATIONS, *int* *n, *float* **iterations,
IMSLS_ITERATIONS_USER, *int* *n, *float* iterations[],
IMSLS_SURVIVAL_INFO, *Imsls_f_survival* **survival_info
IMSLS_N_ROWS_MISSING, *int* *n_rows_missing,
0)

Optional Arguments

IMSLS_X_COL_DIM, *int* x_col_dim (Input)
Column dimension of input array x.
Default: x_col_dim = n_class + n_continuous + 1

IMSLS_X_COL_CENSORING, *int* icen, *int* ilt, *int* irt (Input)
Parameter icen is the column in x containing the censoring code for each observation.

x [*i*] [icen]	**Censoring type**
0	Exact failure at x [*i*] [irt]
1	Right Censored. The response is greater than x [*i*] [irt].
2	Left Censored. The response is less than or equal to x [*i*] [irt].
3	Interval Censored. The response is greater than x [*i*] [irt], but less than or equal to x [*i*] [ilt].

Parameter ilt is the column number of x containing the upper endpoint of the failure interval for interval- and left-censored observations. If there are no left-censored or interval-censored observations, ilt should be set to −1.

Parameter irt is the column number of x containing the lower endpoint of the failure interval for interval- and right-censored observations. If there are no left-censored or interval-censored observations, irt should be set to −1.

Exact failure times are specified in column iy of x. By default, iy is column n_class + n_continuous of x. The default can be changed if keyword IMSLS_X_COL_VARIABLES is specified.

Note that it is allowable to set iy = irt, since a row with an iy value will never have an irt value, and vice versa. This use is illustrated in Example 2.

IMSLS_FREQUENCIES, *int* ifrq (Input)
Column number of x containing the frequency of response for each observation.

IMSLS_FIXED_PARAMETER, *int* ifix (Input)
Column number in x containing a fixed parameter for each observation that is added to the linear response prior to computing the model parameter. The “fixed” parameter allows one to test hypothesis about the parameters via the log-likelihoods.

IMSLS_X_COL_VARIABLES *int* iclass[], *int* icontinuous[], *int* iy (Input)
This keyword allows specification of the variables to be used in the analysis, and overrides the default ordering of variables described for input argument x. Columns are numbered from 0 to x_col_dim – 1. To avoid errors, always specify the keyword IMSLS_X_COL_DIM when using this keyword.

Argument `iclass` is an index vector of length `n_class` containing the column numbers of `x` that correspond to classification variables.

Argument `icontinuous` is an index vector of length `n_continuous` containing the column numbers of `x` that correspond to continuous variables.

Argument `iy` corresponds to the column of `x` which contains the dependent variable.

`IMSLS_EPS`, *float* `eps` (Input)
Argument `eps` is the convergence criterion. Convergence is assumed when the maximum relative change in any coefficient estimate is less than `eps` from one iteration to the next or when the relative change in the log-likelihood, criterion, from one iteration to the next is less than `eps`/100.0.
Default: `eps` = 0.001

`IMSLS_MAX_ITERATIONS`, *int* `max_iterations` (Input)
Maximum number of iterations. Use `max_iterations` = 0 to compute the Hessian, stored in `cov`, and the Newton step, stored in `last_step`, at the initial estimates (The initial estimates must be input. Use keyword `IMSLS_INITIAL_EST_INPUT`).
Default: `max_iterations` = 30

`IMSLS_INTERCEPT`, *or*
`IMSLS_NO_INTERCEPT`,
By default, or if `IMSLS_INTERCEPT` is specified, the intercept is automatically included in the model. If `IMSLS_NO_INTERCEPT` is specified, there is no intercept in the model (unless otherwise provided for by the user).

`IMSLS_INFINITY_CHECK`, *int* `lp_max` (Input)
Remove a right- or left-censored observation from the log-likelihood whenever the probability of the observation exceeds 0.995. At convergence, use linear programming to check that all removed observations actually have infinite linear response

$$z_i\hat{\beta}$$

`obs_status` [*i*] is set to 2 if the linear response is infinite (See optional argument `IMSLS_OBS_STATUS`). If not all removed observations have infinite linear response, re-compute the estimates based upon the observations with finite

$$z_i\hat{\beta}$$

Parameter `lp_max` is the maximum number of observations that can be handled in the linear programming. Setting
`lp_max` = `n_observations` is always sufficient.
Default: No infinity checking; `lp_max` = 0

`IMSLS_NO_INFINITY_CHECK`
: Iterates without checking for infinite estimates. This option is the default.

`IMSLS_EFFECTS`, *int* `n_effects`, *int* `n_var_effects[]`, *int* `indices_effects[]` (Input)
: Use this keyword to specify the effects in the model.

  Variable `n_effects` is the number of effects (sources of variation) in the model. Variable `n_var_effects` is an array of length `n_effects` containing the number of variables associated with each effect in the model.

  Argument `indices_effects` is an index array of length `n_var_effects` [0] + `n_var_effects` [1] + … + `n_var_effects` [`n_effects` − 1]. The first `n_var_effects` [0] elements give the column numbers of `x` for each variable in the first effect. The next `n_var_effects`[1] elements give the column numbers for each variable in the second effect. The last `n_var_effects` [`n_effects` − 1] elements give the column numbers for each variable in the last effect.

`IMSLS_INITIAL_EST_INTERNAL`, *or*
`IMSLS_INITIAL_EST_INPUT`, *int* `n_coef_input`, *float* `estimates[]` (Input)
: By default, or if `IMSLS_INIT_INTERNAL` is specified, then unweighted linear regression is used to obtain initial estimates. If `IMSLS_INITIAL_EST_INPUT` is specified, then the `n_coef_input` elements of `estimates` contain initial estimates of the parameters (which requires that the user know the number of coefficients in the model prior to the call to `survival_glm`). See optional argument `IMSLS_COEF_STAT` for a description of the "nuisance" parameter, which is the first element of array `estimates`.

`IMSLS_MAX_CLASS`, *int* `max_class` (Input)
: An upper bound on the sum of the number of distinct values taken on by each classification variable. Internal workspace usage can be significantly reduced with an appropriate choice of `max_class`.
  Default: `max_class` = `n_observations` * `n_class`

`IMSLS_CLASS_INFO`, *int* `**n_class_values`, *float* `**class_values` (Output)
: Argument `n_class_values` is the address of a pointer to the internally allocated array of length `n_class` containing the number of values taken by each classification variable; the *i*-th classification variable has `n_class_values` [*i*] distinct values. Argument `class_values` is the address of a pointer to the internally allocated array of length

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

containing the distinct values of the classification variables in ascending order. The first `n_class_values` [0] elements of `class_values` contain the values for the first classification variables, the next `n_class_values` [1] elements contain the values for the second classification variable, etc.

`IMSLS_CLASS_INFO_USER`, *int* `n_class_values[]`, *float* `class_values[]` (Output)
Storage for arrays `n_class_values` and `class_values` is provided by the user. See `IMSLS_CLASS_INFO`.

`IMSLS_COEF_STAT`, *float* `**coef_statistics` (Output)
Address of a pointer to an internally allocated array of size `n_coefficients` * 4 containing the parameter estimates and associated statistics:

Column	Statistic
0	Coefficient estimate.
1	Estimated standard deviation of the estimated coefficient.
2	Asymptotic normal score for testing that the coefficient is zero.
3	The *p*-value associated with the normal score in Column 2.

When present in the model, the first coefficient in `coef_statistics` is the estimate of the "nuisance" parameter, and the remaining coefficients are estimates of the parameters associated with the "linear" model, beginning with the intercept, if present. Nuisance parameters are as follows:

model	
0	No nuisance parameter
1	Coefficient of the quadratic term in time, θ
2-9	Scale parameter, σ
10	Shape parameter, θ

`IMSLS_COEF_STAT_USER`, *float* `coef_statistics[]` (Output)
Storage for array `coef_statistics` is provided by the user. See `IMSLS_COEF_STAT`.

`IMSLS_CRITERION`, *float* `*criterion` (Output)
Optimized criterion. The criterion to be maximized is a constant plus the log-likelihood.

`IMSLS_COV`, *float* `**cov` (Output)
Address of a pointer to the internally allocated array of size

n_coefficients by n_coefficients containing the estimated asymptotic covariance matrix of the coefficients. For max_iterations = 0, this is the Hessian computed at the initial parameter estimates.

IMSLS_COV_USER, *float* cov[] (Ouput)
Storage for array cov is provided by the user. See IMSLS_COV.

IMSLS_MEANS, *float* **means (Output)
Address of a pointer to the internally allocated array containing the means of the design variables. The array is of length n_coefficients – *m* if IMSLS_NO_INTERCEPT is specified, and of length n_coefficients – *m* – 1 otherwise. Here, *m* is equal to 0 if model = 0, and equal to 1 otherwise.

IMSLS_MEANS_USER, *float* means[] (Output)
Storage for array means is provided by the user. See IMSLS_MEANS.

IMSLS_CASE_ANALYSIS, *float* **case_statistics (Output)
Address of a pointer to the internally allocated array of size n_observations by 5 containing the case analysis below:

Column	Statistic
0	Estimated predicted value.
1	Estimated influence or leverage.
2	Estimated residual.
3	Estimated cumulative hazard.
4	Non-censored observations: Estimated density at the observation failure time and covariate values.
	Censored observations: The corresponding estimated probability.

If max_iterations = 0, case_statistics is an array of length n_observations containing the estimated probability (for censored observations) or the estimated density (for non-censored observations)

IMSLS_CASE_ANALYSIS_USER, *float* case_statistics[] (Output)
Storage for array case_statistics is provided by the user. See IMSLS_CASE_ANALYSIS.

IMSLS_LAST_STEP, *float* **last_step (Output)
Address of a pointer to the internally allocated array of length n_coefficients containing the last parameter updates (excluding step halvings). Parameter last_step is computed as the inverse of the matrix of second partial derivatives times the vector of first partial derivatives of the log-likelihood. When max_iterations = 0, the derivatives are computed at the initial estimates.

`IMSLS_LAST_STEP_USER`, *float* `last_step[]` (Output)
Storage for array `last_step` is provided by the user. See `IMSLS_LAST_STEP`.

`IMSLS_OBS_STATUS`, *int* `**obs_status` (Output)
Address of a pointer to the internally allocated array of length `n_observations` indicating which observations are included in the extended likelihood.

`obs_status` [*i*]	**Status of Observation**
0	Observation *i* is in the likelihood
1	Observation *i* cannot be in the likelihood because it contains at least one missing value in `x`.
2	Observation *i* is not in the likelihood. Its estimated parameter is infinite.

`IMSLS_OBS_STATUS_USER`, *int* `obs_status[]` (Output)
Storage for array `obs_status` is provided by the user. See `IMSLS_OBS_STATUS`.

`IMSLS_ITERATIONS`, *int* `*n`, *float* `**iterations` (Output)
Address of a pointer to the internally allocated array of size, `n` by 5 containing information about each iteration of the analysis, where `n` is equal to the number of iterations.

column	**statistic**
0	Method of iteration
	Q-N Step = 0
	N-R Step = 1
1	Iteration number
2	Step size
3	Maximum scaled coefficient update
4	Log-likelihood

`IMSLS_ITERATIONS_USER`, *int* `*n`, *float* `iterations[]` (Output)
Storage for array iterations is provided by the user. See `IMSLS_ITERATIONS`.

`IMSLS_SURVIVAL_INFO`, *Imsls_f_survival* `**survival_info` (Output)
Address of the pointer to an internally allocated structure of type *Imsls_f_survival* containing information about the survival analysis. This structure is required input for function `imsls_f_survival_estimates`.

`IMSLS_N_ROWS_MISSING`, *int* `*n_rows_missing` (Output)
Number of rows of data that contain missing values in one or more of the

following vectors or columns of `x`: `iy`, `icen`, `ilt`, `irt`, `ifrq`, `ifix`, `iclass`, `icontinuous`, or `indices_effects`.

Comments

1. Dummy variables are generated for the classification variables as follows: An ascending list of all distinct values of each classification variable is obtained and stored in `class_values`. Dummy variables are then generated for each but the last of these distinct values. Each dummy variable is zero unless the classification variable equals the list value corresponding to the dummy variable, in which case the dummy variable is one. See keyword `IMSLS_LEAVE_OUT_LAST` for optional argument `IMSLS_DUMMY` in `imsls_f_regressors_for_glm` (Chapter 2. "Regression").
2. The "product" of a classification variable with a covariate yields dummy variables equal to the product of the covariate with each of the dummy variables associated with the classification variable.
3. The "product" of two classification variables yields dummy variables in the usual manner. Each dummy variable associated with the first classification variable multiplies each dummy variable associated with the second classification variable. The resulting dummy variables are such that the index of the second classification variable varies fastest.

Description

Function `imsls_f_survival_glm` computes the maximum likelihood estimates of parameters and associated statistics in generalized linear models commonly found in survival (reliability) analysis. Although the terminology used will be from the survival area, the methods discussed have applications in many areas of data analysis, including reliability analysis and event history analysis. These methods can be used anywhere a random variable from one of the discussed distributions is parameterized via one of the models available in `imsls_f_survival_glm`. Thus, while it is not advisable to do so, standard multiple linear regression can be performed by routine `imsls_f_survival_glm`. Estimates for any of 10 standard models can be computed. Exact, left-censored, right-censored, or interval-censored observations are allowed (note that left censoring is the same as interval censoring with the left endpoint equal to the left endpoint of the support of the distribution).

Let $\eta = x^T\beta$ be the linear parameterization, where x is a design vector obtained by `imsls_f_survival_glm` via function `imsls_f_regressors_for_glm` from a row of `x`, and β is a vector of parameters associated with the linear model. Let T denote the random response variable and $S(t)$ denote the probability that $T > t$. All models considered also allow a fixed parameter w_i for observation i (input in column `ifix` of `x`). Use of this parameter is discussed below. There also may be nuisance parameters $\theta > 0$, or $\sigma > 0$ to be estimated (along with β) in the various models. Let Φ denote the cumulative normal distribution. The survival models available in `imsls_f_survival_glm` are:

model	Name	*S* (*t*)
0	Exponential	$\exp\left[-t\exp\left(w_i+\eta\right)\right]$
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	$\{1+\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}^{-1}$
5	Logistic	$\{1+\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}^{-1}$
6	Log least extreme value	$\exp\{-\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}$
7	Least extreme value	$\exp\{-\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}$
8	Log extreme value	$1-\exp\{-\exp\left(\frac{\ln(t)-\eta-w_i}{\sigma}\right)\}$
9	Extreme value	$1-\exp\{-\exp\left(\frac{t-\eta-w_i}{\sigma}\right)\}$
10	Weibull	$\exp\{-\left[\frac{t}{\exp\left(w_i+\eta\right)}\right]^{\theta}\}$

Note that the log-least-extreme-value model is a reparameterization of the Weibull model. Moreover, models 0, 1, 2, 4, 6, 8, and 10 require that $T > 0$, while all of the remaining models allow any value for T, $-\infty < T < \infty$.

Each row vector in the data matrix can represent a single observation; or, through the use of vector frequencies, each row can represent several observations. Also note that classification variables and their products are easily incorporated into the models via the usual regression-type specifications.

The constant parameter W_i is input in `x` and may be used for a number of purposes. For example, if the parameter in an exponential model is known to depend upon the size of the area tested, volume of a radioactive mass, or population density, etc., then a multiplicative factor of the exponential parameter $\lambda = \exp(x\beta)$ may be known apriori. This factor can be input in W_i (W_i is the log of the factor).

An alternate use of W_i is as follows: It may be that $\lambda = \exp(x_1\beta_1 + x_2\beta_2)$, where β_2 is known. Letting $W_i = x_2\beta_2$, estimates for β_1 can be obtained via `imsls_f_survival_glm` with the known fixed values for β_2. Standard methods can then be used to test hypothesis about β_1 via computed log-likelihoods.

Computational Details

The computations proceed as follows:

1. The input parameters are checked for consistency and validity.

   - Estimates of the means of the "independent" or design variables are computed. Means are computed as

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

2. If initial estimates are not provided by the user (see optional argument `IMSLS_INITIAL_EST_INPUT`), the initial estimates are calculated as follows:

   - Models 2-10

   A. Kaplan-Meier estimates of the survival probability,

$$\hat{S}(t)$$

   at the upper limit of each failure interval are obtained. (Because upper limits are used, interval- and left-censored data are assumed to be exact failures at the upper endpoint of the failure interval.) The Kaplan-Meier estimate is computed under the assumption that all failure distributions are identical (i.e., all β's but the intercept, if present, are assumed to be zero).

   B. If there is an intercept in the model, a simple linear regression is performed predicting

$$S^{-1}\left(\hat{S}(t)\right) - w_i = \alpha + \phi t'$$

   where t' is computed at the upper endpoint of each failure interval, $t' = t$ in models 3, 5, 7, and 9, and $t' = \ln(t)$ in models 2, 4, 6, 8, and 10, and w_i is the fixed constant, if present.

   If there is no intercept in the model, then α is fixed at zero, and the model

$$S^{-1}\left(\hat{S}(t)\right) - \hat{\phi}t' - w_i = x^T\beta$$

is fit instead. In this model, the coefficients β are used in place of the location estimate α above. Here

$$\hat{\phi}$$

is estimated from the simple linear regression with α = 0.

C. If the intercept is in the model, then in log-location-scale models (models 1-8),

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

and the initial estimate of the intercept is assumed to be $\hat{\alpha}$.

In the Weibull model

$$\hat{\theta} = 1/\hat{\phi}$$

and the intercept is assumed to be $\hat{\alpha}$.

Initial estimates of all parameters β, other than the intercept, are assumed to be zero.

If there is no intercept in the model, the scale parameter is estimated as above, and the estimates

$$\hat{\beta}$$

from Step 2 are used as initial estimates for the β's.

- Models 0 and 1

For the exponential models (`model` = 0 or 1), the "average total time on" test statistic is used to obtain an estimate for the intercept. Specifically, let T_t denote the total number of failures divided by the total time on test. The initial estimates for the intercept is then $\ln(T_t)$. Initial estimates for the remaining parameters β are assumed to be zero, and if `model` = 1, the initial estimate for the linear hazard parameter θ is assumed to be a small positive number. When the intercept is not in the model, the initial estimate for the parameter θ is assumed to be a small positive number, and initial estimates of the parameters β are computed via multiple linear regression as in Part A.

3. A quasi-Newton algorithm is used in the initial iterations based on a Hessian estimate

$$\hat{H}_{\kappa_j \kappa_l} = \sum_i l'_{i\alpha_j i\alpha_l}$$

where $l'_{i\alpha j}$ is the partial derivative of the i-th term in the log-likelihood with respect to the parameter α_j, and a_j denotes one of the parameter to be estimated.

When the relative change in the log-likelihood from one iteration to the next is 0.1 or less, exact second partial derivatives are used for the Hessian so the Newton-Rapheson iteration is used.

If the initial step size results in an increase in the log-likelihood, the full step is used. If the log-likelihood decreases for the initial step size, the step size is halved, and a check for an increase in the log-likelihood performed. Step-halving is performed (as a simple line search) until an increase in the log-likelihood is detected, or until the step size becomes very small (the initial step size is 1.0).

4. Convergence is assumed when the maximum relative change in any coefficient update from one iteration to the next is less than `eps` or when the relative change in the log-likelihood from one iteration to the next is less than `eps`/100. Convergence is also assumed after `maxit` iterations or when step halving leads to a very small step size with no increase in the log-likelihood.
5. If requested (see optional argument `IMSLS_INFINITY_CHECK`), then the methods of Clarkson and Jennrich (1988) are used to check for the existence of infinite estimates in

$$\eta_i = x_i^T \beta$$

As an example of a situation in which infinite estimates can occur, suppose that observation j is right-censored with $t_j > 15$ in a normal distribution model in which the mean is

$$\mu_j = x_j^T \beta = \eta_j$$

where x_j is the observation design vector. If the design vector x_j for parameter β_m is such that $x_{jm} = 1$ and $x_{im} = 0$ for all $i \neq j$, then the optimal estimate of β_m occurs at

$$\hat{\beta}_m = \infty$$

leading to an infinite estimate of both β_m and η_j. In `imsls_f_survival_glm`, such estimates can be "computed".

In all models fit by `imsls_f_survival_glm`, infinite estimates can only occur when the optimal estimated probability associated with the left- or right-censored observation is 1. If infinity checking is on, left- or right-censored observations that have estimated probability greater than 0.995 at some point during the iterations are excluded from the log-likelihood, and the iterations proceed with a log-likelihood based on the remaining observations. This allows convergence of the algorithm when the maximum relative change in the estimated coefficients is small and also allows for a more precise determination of observations with infinite

$$\eta_i = x_i^T \beta$$

At convergence, linear programming is used to ensure that the eliminated observations have infinite η_i. If some (or all) of the removed observations should not have been removed (because their estimated η_i's must be finite), then the iterations are restarted with a log-likelihood based upon the finite η_i observations. See Clarkson and Jennrich (1988) for more details.

When infinity checking is turned off (see optional argument `IMSLS_NO_INFINITY_CHECK`), no observations are eliminated during the iterations. In this case, the infinite estimates occur, some (or all) of the coefficient estimates

$$\hat{\beta}$$

will become large, and it is likely that the Hessian will become (numerically) singular prior to convergence.

6. The case statistics are computed as follows: Let I_i (θ_i)denote the log-likelihood
of the i-th observation evaluated at θ_i, let I'_i denote the vector of derivatives of
I_i with respect to all parameters, $I'_{\eta,i}$ denote the derivative of I_i with respect to $\eta = x^T\beta$, H denote the Hessian, and E denote expectation. Then the columns of `case_statistics` are:

A. Predicted values are computed as E (T/x) according to standard formulas. If model is 4 or 8, and if $s \geq 1$, then the expected values cannot be computed because they are infinite.

B. Following Cook and Weisberg (1982), the influence (or leverage) of the i-th observation is assumed to be

$$\left(I'_i\right)^T H^{-1} I'_i$$

This quantity is a one-step approximation of the change in the estimates when the i-th observation is deleted (ignoring the nuisance parameters).

C. The "residual" is computed as $I'_{\eta,i}$.

D. The cumulative hazard is computed at the observation covariate values and, for interval observations, the upper endpoint of the failure interval. The cumulative hazard also can be used as a "residual" estimate. If the model is correct, the cumulative hazards should follow a standard exponential distribution. See Cox and Oakes (1984).

Programming Notes

Indicator (dummy) variables are created for the classification variables using function `imsls_f_regressors_for_glm` (Chapter 2, "Regression") using keyword `IMSLS_LEAVE_OUT_LAST` as the argument to the `IMSLS_DUMMY` optional argument.

Examples

Example 1

This example is taken from Lawless (1982, p. 287) and involves the mortality of patients suffering from lung cancer. An exponential distribution is fit for the model

$$\eta = \mu + \alpha_i + \gamma_k + \beta_6 x_3 + \beta_7 x_4 + \beta_8 x_5$$

where α_i is associated with a classification variable with four levels, and γ_k is associated with a classification variable with two levels. Note that because the computations are performed in single precision, there will be some small variation in the estimated coefficients across different machine environments.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,    0.0,    6.0,   63.0,    9.0,  126.0,    0.0,
        1.0,    0.0,    7.0,   65.0,   11.0,  118.0,    0.0,
        1.0,    0.0,    4.0,   69.0,   10.0,   92.0,    0.0,
        1.0,    0.0,    4.0,   63.0,   58.0,    8.0,    0.0,
        1.0,    0.0,    7.0,   48.0,    9.0,   25.0,    1.0,
        1.0,    0.0,    7.0,   48.0,   11.0,   11.0,    0.0,
        2.0,    0.0,    8.0,   63.0,    4.0,   54.0,    0.0,
        2.0,    0.0,    6.0,   63.0,   14.0,  153.0,    0.0,
        2.0,    0.0,    3.0,   53.0,    4.0,   16.0,    0.0,
        2.0,    0.0,    8.0,   43.0,   12.0,   56.0,    0.0,
        2.0,    0.0,    4.0,   55.0,    2.0,   21.0,    0.0,
        2.0,    0.0,    6.0,   66.0,   25.0,  287.0,    0.0,
        2.0,    0.0,    4.0,   67.0,   23.0,   10.0,    0.0,
        3.0,    0.0,    2.0,   61.0,   19.0,    8.0,    0.0,
        3.0,    0.0,    5.0,   63.0,    4.0,   12.0,    0.0,
        4.0,    0.0,    5.0,   66.0,   16.0,  177.0,    0.0,
        4.0,    0.0,    4.0,   68.0,   12.0,   12.0,    0.0,
        4.0,    0.0,    8.0,   41.0,   12.0,  200.0,    0.0,
        4.0,    0.0,    7.0,   53.0,    8.0,  250.0,    0.0,
        4.0,    0.0,    6.0,   37.0,   13.0,  100.0,    0.0,
        1.0,    1.0,    9.0,   54.0,   12.0,  999.0,    0.0,
        1.0,    1.0,    5.0,   52.0,    8.0,  231.0,    1.0,
        1.0,    1.0,    7.0,   50.0,    7.0,  991.0,    0.0,
        1.0,    1.0,    2.0,   65.0,   21.0,    1.0,    0.0,
        1.0,    1.0,    8.0,   52.0,   28.0,  201.0,    0.0,
        1.0,    1.0,    6.0,   70.0,   13.0,   44.0,    0.0,
        1.0,    1.0,    5.0,   40.0,   13.0,   15.0,    0.0,
        2.0,    1.0,    7.0,   36.0,   22.0,  103.0,    1.0,
        2.0,    1.0,    4.0,   44.0,   36.0,    2.0,    0.0,
        2.0,    1.0,    3.0,   54.0,    9.0,   20.0,    0.0,
        2.0,    1.0,    3.0,   59.0,   87.0,   51.0,    0.0,
        3.0,    1.0,    4.0,   69.0,    5.0,   18.0,    0.0,
        3.0,    1.0,    6.0,   50.0,   22.0,   90.0,    0.0,
        3.0,    1.0,    8.0,   62.0,    4.0,   84.0,    0.0,
        4.0,    1.0,    7.0,   68.0,   15.0,  164.0,    0.0,
        4.0,    1.0,    3.0,   39.0,    4.0,   19.0,    0.0,
        4.0,    1.0,    6.0,   49.0,   11.0,   43.0,    0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   n_coef;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    float *coef_stat;
    char *fmt = "%12.4f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_COEF_STAT, &coef_stat,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);
}
```

Output

```
                 Coefficient Statistics
 coefficient         s.e.           z            p
     -1.1027       1.3091     -0.8423       0.3998
     -0.3626       0.4446     -0.8156       0.4149
      0.1271       0.4863      0.2613       0.7939
      0.8690       0.5861      1.4825       0.1385
      0.2697       0.3882      0.6948       0.4873
     -0.5400       0.1081     -4.9946       0.0000
     -0.0090       0.0197     -0.4594       0.6460
     -0.0034       0.0117     -0.2912       0.7710
```

Example 2

This example is the same as Example 1, but more optional arguments are demonstrated.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,    0.0,    6.0,   63.0,    9.0,  126.0,    0.0,
        1.0,    0.0,    7.0,   65.0,   11.0,  118.0,    0.0,
        1.0,    0.0,    4.0,   69.0,   10.0,   92.0,    0.0,
        1.0,    0.0,    4.0,   63.0,   58.0,    8.0,    0.0,
        1.0,    0.0,    7.0,   48.0,    9.0,   25.0,    1.0,
        1.0,    0.0,    7.0,   48.0,   11.0,   11.0,    0.0,
        2.0,    0.0,    8.0,   63.0,    4.0,   54.0,    0.0,
        2.0,    0.0,    6.0,   63.0,   14.0,  153.0,    0.0,
        2.0,    0.0,    3.0,   53.0,    4.0,   16.0,    0.0,
        2.0,    0.0,    8.0,   43.0,   12.0,   56.0,    0.0,
        2.0,    0.0,    4.0,   55.0,    2.0,   21.0,    0.0,
        2.0,    0.0,    6.0,   66.0,   25.0,  287.0,    0.0,
        2.0,    0.0,    4.0,   67.0,   23.0,   10.0,    0.0,
        3.0,    0.0,    2.0,   61.0,   19.0,    8.0,    0.0,
        3.0,    0.0,    5.0,   63.0,    4.0,   12.0,    0.0,
        4.0,    0.0,    5.0,   66.0,   16.0,  177.0,    0.0,
        4.0,    0.0,    4.0,   68.0,   12.0,   12.0,    0.0,
        4.0,    0.0,    8.0,   41.0,   12.0,  200.0,    0.0,
        4.0,    0.0,    7.0,   53.0,    8.0,  250.0,    0.0,
        4.0,    0.0,    6.0,   37.0,   13.0,  100.0,    0.0,
        1.0,    1.0,    9.0,   54.0,   12.0,  999.0,    0.0,
        1.0,    1.0,    5.0,   52.0,    8.0,  231.0,    1.0,
        1.0,    1.0,    7.0,   50.0,    7.0,  991.0,    0.0,
        1.0,    1.0,    2.0,   65.0,   21.0,    1.0,    0.0,
        1.0,    1.0,    8.0,   52.0,   28.0,  201.0,    0.0,
        1.0,    1.0,    6.0,   70.0,   13.0,   44.0,    0.0,
        1.0,    1.0,    5.0,   40.0,   13.0,   15.0,    0.0,
        2.0,    1.0,    7.0,   36.0,   22.0,  103.0,    1.0,
        2.0,    1.0,    4.0,   44.0,   36.0,    2.0,    0.0,
        2.0,    1.0,    3.0,   54.0,    9.0,   20.0,    0.0,
        2.0,    1.0,    3.0,   59.0,   87.0,   51.0,    0.0,
        3.0,    1.0,    4.0,   69.0,    5.0,   18.0,    0.0,
        3.0,    1.0,    6.0,   50.0,   22.0,   90.0,    0.0,
        3.0,    1.0,    8.0,   62.0,    4.0,   84.0,    0.0,
        4.0,    1.0,    7.0,   68.0,   15.0,  164.0,    0.0,
        4.0,    1.0,    3.0,   39.0,    4.0,   19.0,    0.0,
        4.0,    1.0,    6.0,   49.0,   11.0,   43.0,    0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   n_coef;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    int   n, *ncv, nrmiss, *obs;
    float *iterations, *cv, criterion;
    float *coef_stat, *casex;
    char *fmt = "%12.4f";
    char *fmt2 = "%4d%4d%6.4f%8.4f%8.1f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};
    static char *clabels2[] = {"", "Method", "Iteration", "Step Size",
        "Coef Update", "Log-Likelihood"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_COEF_STAT, &coef_stat,
        IMSLS_ITERATIONS, &n, &iterations,
        IMSLS_CASE_ANALYSIS, &casex,
        IMSLS_CLASS_INFO, &ncv, &cv,
        IMSLS_OBS_STATUS, &obs,
        IMSLS_CRITERION, &criterion,
        IMSLS_N_ROWS_MISSING, &nrmiss,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);

    imsls_f_write_matrix("Iteration Information", n, 5, iterations,
        IMSLS_WRITE_FORMAT, fmt2,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels2, 0);

    printf("\nLog-Likelihood = %12.5f\n", criterion);

    imsls_f_write_matrix("Case Analysis", 1, n_observations, casex,
        IMSLS_WRITE_FORMAT, fmt,
        0);

    imsls_f_write_matrix(
        "Distinct Values for Classification Variable 1",
        1, ncv[0], &cv[0], IMSLS_NO_COL_LABELS, 0);

    imsls_f_write_matrix(
        "Distinct Values for Classification Variable 2",
        1, ncv[1], &cv[ncv[0]], IMSLS_NO_COL_LABELS, 0);

    imsls_i_write_matrix("Observation Status", 1, n_observations,
        obs, 0);

    printf("\nNumber of Missing Values = %2d\n", nrmiss);
}
```

Output

```
                  Coefficient Statistics
 coefficient          s.e.            z            p
     -1.1027        1.3091      -0.8423       0.3998
     -0.3626        0.4446      -0.8156       0.4149
      0.1271        0.4863       0.2613       0.7939
      0.8690        0.5861       1.4825       0.1385
      0.2697        0.3882       0.6948       0.4873
     -0.5400        0.1081      -4.9946       0.0000
     -0.0090        0.0197      -0.4594       0.6460
     -0.0034        0.0117      -0.2912       0.7710

                    Iteration Information
Method  Iteration  Step Size  Coef Update  Log-Likelihood
     0          0     ......     ........          -224.0
     0          1     1.0000       0.9839          -213.4
     1          2     1.0000       3.6033          -207.3
     1          3     1.0000      10.1236          -204.3
     1          4     1.0000       0.1430          -204.1
     1          5     1.0000       0.0117          -204.1
```

```
Log-Likelihood =   -204.13916

                           Case Analysis
          1             2             3             4             5
   262.6884        0.0450       -0.5646        1.5646        0.0008

          6             7             8             9            10
   153.7777        0.0042        0.1806        0.8194        0.0029

         11            12            13            14            15
   270.5347        0.0482        0.5638        0.4362        0.0024

         16            17            18            19            20
    55.3168        0.0844       -0.6631        1.6631        0.0034

         21            22            23            24            25
    61.6845        0.3765        0.8703        0.1297        0.0142

         26            27            28            29            30
   230.4414        0.0025       -0.1085        0.1085        0.8972

         31            32            33            34            35
   232.0135        0.1960        0.9526        0.0474        0.0041

         36            37            38            39            40
   272.8432        0.1677        0.8021        0.1979        0.0030

 Distinct Values for Classification Variable 1
        1           2           3           4

Distinct Values for Classification Variable 2
                    0           1

                               Observation Status
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

Number of Missing Values =  0
```

Example 3

In this example, the same data and model as example 1 are used, but `max_iterations` is set to zero iterations with model coefficients restricted such that $\mu = -1.25$, $\beta_6 = -0.6$, and the remaining six coefficients are equal to zero. A chi-squared statistic, with 8 degrees of freedom for testing the coefficients is specified as above (versus the alternative that it is not as specified), can be computed, based on the output, as

$$\chi^2 = g^T \hat{\Sigma}^{-1} g$$

where

$$\hat{\Sigma}$$

is output in `cov`. The resulting test statistic, $\chi^2 = 6.107$, based upon no iterations is comparable to likelihood ratio test that can be computed from the log-likelihood output in this example (–206.6835) and the log-likelihood output in Example 2 (–204.1392).

$$\chi^2_{LR} = 2\left(206.6835 - 204.1392\right) = 5.0886$$

Neither statistic is significant at the $\alpha = 0.05$ level.

```
#include <imsls.h>

main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,     5.0,  411.0,     0.0,
        1.0,    0.0,    6.0,   63.0,     9.0,  126.0,     0.0,
        1.0,    0.0,    7.0,   65.0,    11.0,  118.0,     0.0,
        1.0,    0.0,    4.0,   69.0,    10.0,   92.0,     0.0,
        1.0,    0.0,    4.0,   63.0,    58.0,    8.0,     0.0,
        1.0,    0.0,    7.0,   48.0,     9.0,   25.0,     1.0,
        1.0,    0.0,    7.0,   48.0,    11.0,   11.0,     0.0,
        2.0,    0.0,    8.0,   63.0,     4.0,   54.0,     0.0,
        2.0,    0.0,    6.0,   63.0,    14.0,  153.0,     0.0,
        2.0,    0.0,    3.0,   53.0,     4.0,   16.0,     0.0,
        2.0,    0.0,    8.0,   43.0,    12.0,   56.0,     0.0,
        2.0,    0.0,    4.0,   55.0,     2.0,   21.0,     0.0,
        2.0,    0.0,    6.0,   66.0,    25.0,  287.0,     0.0,
        2.0,    0.0,    4.0,   67.0,    23.0,   10.0,     0.0,
        3.0,    0.0,    2.0,   61.0,    19.0,    8.0,     0.0,
        3.0,    0.0,    5.0,   63.0,     4.0,   12.0,     0.0,
        4.0,    0.0,    5.0,   66.0,    16.0,  177.0,     0.0,
        4.0,    0.0,    4.0,   68.0,    12.0,   12.0,     0.0,
        4.0,    0.0,    8.0,   41.0,    12.0,  200.0,     0.0,
        4.0,    0.0,    7.0,   53.0,     8.0,  250.0,     0.0,
        4.0,    0.0,    6.0,   37.0,    13.0,  100.0,     0.0,
        1.0,    1.0,    9.0,   54.0,    12.0,  999.0,     0.0,
        1.0,    1.0,    5.0,   52.0,     8.0,  231.0,     1.0,
        1.0,    1.0,    7.0,   50.0,     7.0,  991.0,     0.0,
        1.0,    1.0,    2.0,   65.0,    21.0,    1.0,     0.0,
        1.0,    1.0,    8.0,   52.0,    28.0,  201.0,     0.0,
        1.0,    1.0,    6.0,   70.0,    13.0,   44.0,     0.0,
        1.0,    1.0,    5.0,   40.0,    13.0,   15.0,     0.0,
        2.0,    1.0,    7.0,   36.0,    22.0,  103.0,     1.0,
        2.0,    1.0,    4.0,   44.0,    36.0,    2.0,     0.0,
        2.0,    1.0,    3.0,   54.0,     9.0,   20.0,     0.0,
        2.0,    1.0,    3.0,   59.0,    87.0,   51.0,     0.0,
        3.0,    1.0,    4.0,   69.0,     5.0,   18.0,     0.0,
        3.0,    1.0,    6.0,   50.0,    22.0,   90.0,     0.0,
        3.0,    1.0,    8.0,   62.0,     4.0,   84.0,     0.0,
        4.0,    1.0,    7.0,   68.0,    15.0,  164.0,     0.0,
        4.0,    1.0,    3.0,   39.0,     4.0,   19.0,     0.0,
        4.0,    1.0,    6.0,   49.0,    11.0,   43.0,     0.0,
        4.0,    1.0,    8.0,   64.0,    10.0,  340.0,     0.0,
        4.0,    1.0,    7.0,   67.0,    18.0,  231.0,     0.0};
    int   n_observations = 40;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    int   n_coef_input = 8;
    static float estimates[8] = {-1.25, 0.0, 0.0, 0.0,
        0.0, -0.6, 0.0, 0.0};

    int   n_coef;
    float *coef_stat, *means, *cov;
    float criterion, *last_step;

    char *fmt = "%12.4f";
    static char *clabels[] = {"", "coefficient", "s.e.", "z", "p"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous, model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_INITIAL_EST_INPUT, n_coef_input, estimates,
        IMSLS_MAX_ITERATIONS, 0,
        IMSLS_COEF_STAT, &coef_stat,
        IMSLS_MEANS, &means,
        IMSLS_COV, &cov,
        IMSLS_CRITERION, &criterion,
        IMSLS_LAST_STEP, &last_step,
        0);

    imsls_f_write_matrix("Coefficient Statistics", n_coef, 4,
        coef_stat,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels,
        0);

    imsls_f_write_matrix("Covariate Means", 1, n_coef-1, means, 0);

    imsls_f_write_matrix("Hessian", n_coef, n_coef, cov,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_PRINT_UPPER,
        0);

    printf("\nLog-Likelihood = %12.5f\n", criterion);

    imsls_f_write_matrix("Newton-Raphson Step", 1, n_coef, last_step,
        IMSLS_WRITE_FORMAT, fmt, 0);

}
```

Output

```
                Coefficient Statistics
 coefficient         s.e.            z            p
     -1.2500       1.3773      -0.9076       0.3643
      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         1.8969       -0.0906       -0.1641       -0.1681        0.0778
2                       0.1839        0.0996        0.1191        0.0358
3                                     0.2808        0.1264       -0.0226
4                                                   0.6003        0.0460
5                                                                 0.1641

               6             7             8
1        -0.0818       -0.0235       -0.0012
2        -0.0005       -0.0008        0.0006
3         0.0104        0.0005       -0.0021
4         0.0193       -0.0016        0.0007
5         0.0060       -0.0040        0.0017
6         0.0125        0.0000        0.0003
7                       0.0005       -0.0001
8                                     0.0001

Log-Likelihood =   -206.68349

                       Newton-Raphson Step
             1             2             3             4             5
      0.1706       -0.3365        0.1333        1.2967        0.2985

             6             7             8
      0.0625       -0.0112       -0.0026
```

Warning Errors

`IMSLS_CONVERGENCE_ASSUMED_1`	Too many step halvings. Convergence is assumed.
`IMSLS_CONVERGENCE_ASSUMED_2`	Too many step iterations. Convergence is assumed.
`IMSLS_NO_PREDICTED_1`	"estimates[0]" > 1.0. The expected value for the log logistic distribution ("model" = 4) does not exist. Predicted values will not be calculated.
`IMSLS_NO_PREDICTED_2`	"estimates[0]" > 1.0. The expected value for the log extreme value distribution("model" = 8) does not

	exist. Predicted values will not be calculated.
`IMSLS_NEG_EIGENVALUE`	The Hessian has at least one negative eigenvalue. An upper bound on the absolute value of the minimum eigenvalue is # corresponding to variable index #.
`IMSLS_INVALID_FAILURE_TIME_4`	“x[#][“ilt”= #]” = # and “x[#][“irt”= #]” = #. The censoring interval has length 0.0. The censoring code for this observation is being set to 0.0.

Fatal Error

`IMSLS_MAX_CLASS_TOO_SMALL`	The number of distinct values of the classification variables exceeds “max_class” = #.
`IMSLS_TOO_FEW_COEF`	`IMSLS_INITIAL_EST_INPUT` is specified, and “n_coef_input” = #. The model specified requires # coefficients.
`IMSLS_TOO_FEW_VALID_OBS`	“n_observations” = # and “n_rows_missing” = #. “n_observations”– ”n_rows_missing” must be greater than or equal to 2 in order to estimate the coefficients.
`IMSLS_SVGLM_1`	For the exponential model (“model” = 0) with “n_effects” = # and no intercept, “n_coef” has been determined to equal 0. With no coefficients in the model, processing cannot continue.
`IMSLS_INCREASE_LP_MAX`	Too many observations are to be deleted from the model. Either use a different model or increase the workspace.
`IMSLS_INVALID_DATA_8`	“n_class_values[#]” = #. The number of distinct values for each classification variable must be greater than one.

survival_estimates

Estimates survival probabilities and hazard rates for the various parametric models.

Synopsis

#include `<imsls.h>`

int `*imsls_f_survival_estimates` (*Imsls_f_survival* `*survival_info`, *int* `n_observations`, *float* `xpt[]`, *float* `time`, *int* `npt`, *float* `delta`, ..., 0)

The type *double* function is `imsls_d_survival_estimates`.

Required Arguments

Imsls_f_survival `*survival_info` (Input)
Pointer to structure of type *Imsls_f_survival* containing the estimated survival coefficients and other related information. See `imsls_f_survival_glm`.

int `n_observations` (Input)
Number of observations for which estimates are to be calculated.

float `xpt[]` (Input)
Array `xpt` is an array of size `n_observations` by `x_col_dim` containing the groups of covariates for which estimates are desired, where `x_col_dim` is described in the documentation for `imsls_f_survival_glm`. The covariates must be specified exactly as in the call to `imsls_f_survival_glm` which produced `survival_info`.

float `time` (Input)
Beginning of the time grid for which estimates are desired. Survival probabilities and hazard rates are computed for each covariate vector over the grid of time points `time` + *i**`delta` for *i* = 0, 1, …, `npt` − 1.

int `npt` (Input)
Number of points on the time grid for which survival probabilities are desired.

float `delta` (Input)
Increment between time points on the time grid.

Return Value

An array of size `npt` by (2 ∗ `n_observations` + 1) containing the estimated survival probabilities for the covariate groups specified in `xpt`. Column 0 contains the survival time. Columns 1 and 2 contain the estimated survival probabilities and hazard rates, respectively, for the covariates in the first row of xpt. In general, the survival and hazard for row *i* of xpt is contained in columns $2i - 1$ and $2i$, respectively, for $i = 1, 2, \ldots,$ npt.

Synopsis with Optional Arguments

#include <imsls.h>

int *imsls_f_survival_estimates (*Imsls_f_survival* survival_info,
    int n_observations, *float* xpt[], *float* time, *int* npt,
    float delta,
    IMSLS_XBETA, *float* **xbeta,
    IMSLS_XBETA_USER, *float* xbeta[],
    IMSLS_RETURN_USER, *float* sprob[],
    0)

Optional Arguments

IMSLS_XBETA, *float* **xbeta (Output)
Address of a pointer to an array of length n_observations containing the estimated linear response

$$w + x\hat{\beta}$$

for each row of xpt.

IMSLS_XBETA_USER, *float* xbeta[] (Output)
Storage for array xbeta is provided by the user. See IMSLS_XBETA.

IMSLS_RETURN_USER, *float* sprob[] (Output)
User supplied array of size npt by (2 * n_observations + 1) containing the estimated survival probabilities for the covariate groups specified in xpt. Column 0 contains the survival time. Columns 1 and 2 contain the estimated survival probabilities and hazard rates, respectively, for the covariates in the first row of xpt. In general, the survival and hazard for row *i* of xpt is contained in columns $2i - 1$ and $2i$, respectively, for $i = 1, 2, \ldots,$ npt.

Description

Function imsls_f_survival_estimates computes estimates of survival probabilities and hazard rates for the parametric survival/reliability models fit by function imsls_f_survival_glm.

Let $\eta = x^T\beta$ be the linear parameterization, where x is the design vector corresponding to a row of xpt (imsls_f_survival_estimates generates the design vector using function imsls_f_regressors_for_glm), and β is a vector of parameters associated with the linear model. Let T denote the random response variable and $S(t)$ denote the probability that $T > t$. All models considered also allow a fixed parameter w (input in column ifix of xpt). Use of the parameter is discussed in function imsls_f_survival_glm. There also may be nuisance parameters $\theta > 0$ or $\sigma > 0$. Let Φ denote the cumulative normal distribution. The survival models available in `imsls_f_survival_estimates` are:

model	**Name**	***S* (*t*)**
0	Exponential	$\exp\left[-t \exp\left(w_i + \eta\right)\right]$
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	$\{1 + \exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}^{-1}$
5	Logistic	$\{1 + \exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}^{-1}$
6	Log least extreme value	$\exp\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}$
7	Least extreme value	$\exp\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}$
8	Log extreme value	$1 - \exp\{-\exp\left(\frac{\ln(t) - \eta - w_i}{\sigma}\right)\}$
9	Extreme value	$1 - \exp\{-\exp\left(\frac{t - \eta - w_i}{\sigma}\right)\}$
10	Weibull	$\exp\{-\left[\frac{t}{\exp\left(w_i + \eta\right)}\right]^{\theta}\}$

Let $\lambda(t)$ denote the hazard rate at time *t*. Then $\lambda(t)$ and $S(t)$ are related at

$$S(t) = \exp(\int_{-\infty}^{t} \lambda(s)\,ds)$$

Models 0, 1, 2, 4, 6, 8, and 10 require that $T > 0$ (in which case assume $\lambda(s) = 0$ for $s < 0$), while the remaining models allow arbitrary values for T, $-\infty < T < \infty$. The computations proceed in function `imsls_f_survival_estimates` as follows:

1. The input arguments are checked for consistency and validity.

2. For each row of `xpt`, the explanatory variables are generated from the classification and variables and the covariates using function `imsls_f_regressors_for_glm` with `dummy_method` = `IMSLS_LEAVE_OUT_LAST`. Given the explanatory variables x, η is computed as $\eta = x^T\beta$, where β is input in `survival_info`.

3. For each point requested in the time grid, the survival probabilities and hazard rates are computed.

Example

This example is a continuation of the first example given for function `imsls_f_survival_glm`. Prior to calling `survival_estimates`, `imsls_f_survival_glm` is invoked to compute the parameter estimates (contained in the structure `survival_info`). The example is taken from Lawless (1982, p. 287) and involves the mortality of patients suffering from lung cancer.

```
#include <imsls.h>
#include <stdlib.h>
main() {
    static float x[40][7] = {
        1.0,    0.0,    7.0,   64.0,    5.0,  411.0,    0.0,
        1.0,    0.0,    6.0,   63.0,    9.0,  126.0,    0.0,
        1.0,    0.0,    7.0,   65.0,   11.0,  118.0,    0.0,
        1.0,    0.0,    4.0,   69.0,   10.0,   92.0,    0.0,
        1.0,    0.0,    4.0,   63.0,   58.0,    8.0,    0.0,
        1.0,    0.0,    7.0,   48.0,    9.0,   25.0,    1.0,
        1.0,    0.0,    7.0,   48.0,   11.0,   11.0,    0.0,
        2.0,    0.0,    8.0,   63.0,    4.0,   54.0,    0.0,
        2.0,    0.0,    6.0,   63.0,   14.0,  153.0,    0.0,
        2.0,    0.0,    3.0,   53.0,    4.0,   16.0,    0.0,
        2.0,    0.0,    8.0,   43.0,   12.0,   56.0,    0.0,
        2.0,    0.0,    4.0,   55.0,    2.0,   21.0,    0.0,
        2.0,    0.0,    6.0,   66.0,   25.0,  287.0,    0.0,
        2.0,    0.0,    4.0,   67.0,   23.0,   10.0,    0.0,
        3.0,    0.0,    2.0,   61.0,   19.0,    8.0,    0.0,
        3.0,    0.0,    5.0,   63.0,    4.0,   12.0,    0.0,
        4.0,    0.0,    5.0,   66.0,   16.0,  177.0,    0.0,
        4.0,    0.0,    4.0,   68.0,   12.0,   12.0,    0.0,
        4.0,    0.0,    8.0,   41.0,   12.0,  200.0,    0.0,
        4.0,    0.0,    7.0,   53.0,    8.0,  250.0,    0.0,
        4.0,    0.0,    6.0,   37.0,   13.0,  100.0,    0.0,
        1.0,    1.0,    9.0,   54.0,   12.0,  999.0,    0.0,
        1.0,    1.0,    5.0,   52.0,    8.0,  231.0,    1.0,
        1.0,    1.0,    7.0,   50.0,    7.0,  991.0,    0.0,
        1.0,    1.0,    2.0,   65.0,   21.0,    1.0,    0.0,
        1.0,    1.0,    8.0,   52.0,   28.0,  201.0,    0.0,
        1.0,    1.0,    6.0,   70.0,   13.0,   44.0,    0.0,
        1.0,    1.0,    5.0,   40.0,   13.0,   15.0,    0.0,
        2.0,    1.0,    7.0,   36.0,   22.0,  103.0,    1.0,
        2.0,    1.0,    4.0,   44.0,   36.0,    2.0,    0.0,
        2.0,    1.0,    3.0,   54.0,    9.0,   20.0,    0.0,
        2.0,    1.0,    3.0,   59.0,   87.0,   51.0,    0.0,
        3.0,    1.0,    4.0,   69.0,    5.0,   18.0,    0.0,
        3.0,    1.0,    6.0,   50.0,   22.0,   90.0,    0.0,
        3.0,    1.0,    8.0,   62.0,    4.0,   84.0,    0.0,
        4.0,    1.0,    7.0,   68.0,   15.0,  164.0,    0.0,
        4.0,    1.0,    3.0,   39.0,    4.0,   19.0,    0.0,
        4.0,    1.0,    6.0,   49.0,   11.0,   43.0,    0.0,
        4.0,    1.0,    8.0,   64.0,   10.0,  340.0,    0.0,
        4.0,    1.0,    7.0,   67.0,   18.0,  231.0,    0.0};

    int   n_observations = 40;
    int   n_estimates = 2;
    int   n_class = 2;
    int   n_continuous = 3;
    int   model = 0;
    int   icen = 6, ilt = -1, irt = 5;
    int   lp_max = 40;
    float time = 10.0;
    int   npt = 10;
    float delta = 20.0;

    int   n_coef;
    float *sprob;
    Imsls_f_survival *survival_info;
    char *fmt = "%12.2f%10.4f%10.6f%10.4f%10.6f";
    char *clabels[] = {"", "Time", "S1", "H1", "S2", "H2"};

    n_coef = imsls_f_survival_glm(n_observations, n_class,
        n_continuous,
        model, &x[0][0],
        IMSLS_X_COL_CENSORING, icen, ilt, irt,
        IMSLS_INFINITY_CHECK, lp_max,
        IMSLS_SURVIVAL_INFO, &survival_info,
        0);

    sprob = imsls_f_survival_estimates(survival_info, n_estimates,
        &x[0][0], time, npt, delta, 0);

    imsls_f_write_matrix("Survival and Hazard Estimates",
        npt, 2*n_estimates+1, sprob,
        IMSLS_WRITE_FORMAT, fmt, IMSLS_NO_ROW_LABELS,
        IMSLS_COL_LABELS, clabels, 0);

    free (survival_info);
    free (sprob);
}
```

Output

```
                Survival and Hazard Estimates

        Time        S1        H1        S2        H2
       10.00    0.9626  0.003807    0.9370  0.006503
       30.00    0.8921  0.003807    0.8228  0.006503
       50.00    0.8267  0.003807    0.7224  0.006503
       70.00    0.7661  0.003807    0.6343  0.006503
       90.00    0.7099  0.003807    0.5570  0.006503
      110.00    0.6579  0.003807    0.4890  0.006503
      130.00    0.6096  0.003807    0.4294  0.006503
      150.00    0.5649  0.003807    0.3770  0.006503
```

```
170.00      0.5235    0.003807      0.3310    0.006503
190.00      0.4852    0.003807      0.2907    0.006503
```

Note that the hazard rate is constant over time for the exponential model.

Warning Errors

`IMSLS_CONVERGENCE_ASSUMED_1`	Too many step halvings. Convergence is assumed.
`IMSLS_CONVERGENCE_ASSUMED_2`	Too many step iterations. Convergence is assumed.
`IMSLS_NO_PREDICTED_1`	"estimates[0]" > 1.0. The expected value for the log logistic distribution ("model" = 4) does not exist. Predicted values will not be calculated.
`IMSLS_NO_PREDICTED_2`	"estimates[0]" > 1.0. The expected value for the log extreme value distribution ("model" = 8) does not exist. Predicted values will not be calculated.
`IMSLS_NEG_EIGENVALUE`	The Hessian has at least one negative eigenvalue. An upper bound on the absolute value of the minimum eigenvalue is # corresponding to variable index #.
`IMSLS_INVALID_FAILURE_TIME_4`	"x[#]["ilt"= #]" = # and "x[#]["irt"= #]" = #. The censoring interval has length 0.0. The censoring code for this observation is being set to 0.0.

Fatal Error

`IMSLS_MAX_CLASS_TOO_SMALL`	The number of distinct values of the classification variables exceeds "max_class" = #.
`IMSLS_TOO_FEW_COEF`	`IMSLS_INITIAL_EST_INPUT` is specified, and "n_coef_input" = #. The model specified requires # coefficients.
`IMSLS_TOO_FEW_VALID_OBS`	"n_observations" = %(i1) and "n_rows_missing" = #. "n_observations"–"n_rows_missing" must be greater

	than or equal to 2 in order to estimate the coefficients.
`IMSLS_SVGLM_1`	For the exponential model ("model" = 0) with "n_effects" = # and no intercept, "n_coef" has been determined to equal 0. With no coefficients in the model, processing cannot continue.
`IMSLS_INCREASE_LP_MAX`	Too many observations are to be deleted from the model. Either use a different model or increase the workspace.
`IMSLS_INVALID_DATA_8`	"n_class_values[#]" = #. The number of distinct values for each classification variable must be greater than one.

nonparam_hazard_rate

Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.

Synopsis

#include `<imsls.h>`

float `*imsls_f_nonparam_hazard_rate` (*int* `n_observations`,
float `t[]`, *int* `n_hazard`, *float* `hazard_min`,
float `hazard_increment`, ..., 0)

The type *double* function is `imsls_d_nonparam_hazard_rate`.

Required Arguments

int `n_observations` (Input)
Number of observations.

float `t[]` (Input)
An array of `n_observations` containing the failure times. If optional argument `IMSLS_CENSOR_CODES` is used, the values of `t` may be treated as exact failure times, as right-censored times, or a combination of exact and right censored times. By default, all times in `t` are assumed to be exact failure times.

int `n_hazard` (Input)
Number of grid points at which to compute the hazard. The function computes the hazard rates over the range given by:
`hazard_min` + j * `hazard_increment`, for $j = 0, \ldots,$ `n_hazard` – 1.

float `hazard_min` (Input)

First grid value.

float `hazard_increment` (Input)

Increment between grid values.

Return Value

Pointer to an array of length `n_hazard` containing the estimated hazard rates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_nonparam_hazard_rate (`*int* `n_observations`,
 float `t[],` *int* `n_hazard`, *float* `hazard_min`,
 float `hazard_increment`
 `IMSLS_RETURN_USER`, *float* `haz[]`,
 `IMSLS_PRINT_LEVEL`, *int* `iprint`,
 `IMSLS_CENSOR_CODES`, *int* `censor_codes[]`,
 `IMSLS_WEIGHT`, *int* `iwto`,
 `IMSLS_SORT_OPTION`, *int* `isort`,
 `IMSLS_K_GRID`, *int* `n_k`, *float* `k_min`, *float* `k_increment`,
 `IMSLS_BETA_GRID`, *int* `n_beta_grid`, *float* `beta_start`,
 float `beta_increment`,
 `IMSLS_N_MISSING`, *int* `*nmiss`,
 `IMSLS_ALPHA,` *float* `*alpha`,
 `IMSLS_BETA,` *float* `*beta`,
 `IMSLS_CRITERION`, *float* `*vml`,
 `IMSLS_K`, *int* `*k`,
 `IMSLS_SORTED_EVENT_TIMES`, *float* `**event_times`,
 `IMSLS_SORTED_EVENT_TIMES_USER`, *float* `event_times[]`,
 `IMSLS_SORTED_CENSOR_CODES`, *int* `**isorted_censor`,
 `IMSLS_SORTED_CENSOR_CODES_USER`, *int* `isorted_censor[]`,
 0)

Optional Arguments

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

If specified, `haz` is a user supplied array of length `n_hazard` containing the estimated hazard rates.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)

Printing option. Default: `iprint` = 0.

iprint	**Action**
0	No printing is performed.
1	The grid estimates and the optimized estimates are printed for each value of *k*.

`IMSLS_CENSOR_CODES`, *int* `censor_codes[]` (Input)
`censor_codes` is an array of length `n_observations` containing the censoring codes for each time in `t`. If `censor_codes[i]=0` the failure time `t[i]` is treated as an exact time of failure. Otherwise it is treated as a right-censored time; that is, the exact time of failure is greater than `t[i]`.
Default: All failure times are treated as exact times of failure with no censoring.

`IMSLS_WEIGHT_OPTION`, *int* `iwto` (Input)
Weight option . If `iwto` = 1, then
$\texttt{weight} = \ln\left(1+1/\left(\texttt{n\_observations}-i\right)\right)$ is used for the *i*-th smallest observation. Otherwise, $\texttt{weight} = 1/\left(\texttt{n\_observations}-i\right)$ is used.
Default: `iwto` = 0.

`IMSLS_SORT_OPTION`, *int* `isort` (Input)
Sorting option . If `isort` = 1, then the event times are not automatically sorted by the function. Otherwise, sorting is performed with exact failure times following tied right-censored times.
Default: `isort` = 0.

`IMSLS_K_GRID`, *int* `n_k`, *float* `k_min`, *float* `k_increment` (Input)
Finds the optimal value of *k* over the range given by: `kmin` + (*j* – 1) * `k_increment`, for *j* = 1, …, `n_k`. Where `n_k` is the number of values of *k* to be considered. `k_min` is the minimum value for parameter *k*. `k_increment` is the increment between successive values of parameter *k*. Parameter *k* is the number of nearest neighbors to be used in computing the *k*-th nearest neighbor distance.
Default: `k_min` is the smallest possible value of *k,* `k_increment` =2, and `n_k` will be at most 10 points.

`IMSLS_BETA_GRID`, *int* `n_beta_grid`, *float* `beta_start`, *float* `beta_increment` (Input)
For `n_beta_grid` > 0, a user-defined grid is used. This grid is defined as `beta_start` + (*j* – 1)*`beta_increment`, for *j* = 1, …, `n_beta_grid`. `beta_start` is the first value to be used in the user-defined grid and `beta_increment` is the increment between successive grid values of `beta`.
Default: The values in the initial beta search are given as follows: Let $\beta^* = -8, -4, -2, -1, -0.5, 0.5, 1$, and 2, and

$$\beta = e^{-\beta^*}$$

For each value of β, `vml` is computed at the optimizing β. The maximizing β is used to initiate the iterations. If the initial β^* is determined from the search to be less than −6, then it is presumed that β is infinite, and an analytic estimate of α based upon infinite β is used. Infinite β corresponds to a flat hazard rate.

`IMSLS_N_MISSING`, *int* `*nmiss` (Output)
Number of missing (NaN, not a number) failure times in `t`.

`IMSLS_ALPHA`, *float* `*alpha` (Output)
Optimal estimate for the parameter α.

`IMSLS_BETA`, *float* `*beta` (Output)
Optimal estimate for the parameter β.

`IMSLS_CRITERION`, *float* `*vml` (Output)
Optimum value of the criterion function.

`IMSLS_K`, *int* `*k` (Output)
Optimal estimate for the parameter *k*.

`IMSLS_SORTED_EVENT_TIMES`, *float* `**event_times` (Output)
Address of a pointer to an array of length `n_observations` containing the times of occurrence of the events, sorted from smallest to largest.

`IMSLS_SORTED_EVENT_TIMES_USER`, *float* `event_times[]` (Output)
Storage for `event_times` is provided by the user. See `IMSLS_SORTED_EVENT_TIMES`.

`IMSLS_SORTED_CENSOR_CODES`, *int* `**isorted_censor` (Output)
Address of a pointer to an array of length `n_observations` containing the sorted censor codes. Censor codes are sorted corresponding to the events `event_times[i]`, with censored observations preceding tied failures.

`IMSLS_SORTED_CENSOR_CODES_USER`, *int* `isorted_censor[]` (Output)
Storage for `isorted_censor` is provided by the user. See `IMSLS_SORTED_CENSOR_CODE`.

Description

Function `imsls_f_nonparam_hazard_rate` is an implementation of the methods discussed by Tanner and Wong (1984) for estimating the hazard rate in survival or reliability data with right censoring. It uses the biweight kernel,

$$K(x) = \begin{cases} \frac{15}{16}(1-x^2)^2 & \text{for} \left|x\right| < 1 \\ 0 & \text{elsewhere} \end{cases}$$

and a modified likelihood to obtain data-based estimates of the smoothing parameters α, β, and *k* needed in the estimation of the hazard rate. For kernel *K*(*x*), define the “smoothed” kernel
$K_s(x - x(j)$ as follows:

$$K_S(x - x_{(j)}) = \frac{1}{\alpha d_{jk}} K\left(\frac{x - x(j)}{\beta d_{jk}}\right)$$

where d_{jk} is the distance to the k-th nearest failure from $x(j)$, and $x(j)$ is the j-th ordered observation (from smallest to largest). For given α and β, the hazard at point x is then

$$h(x) = \sum_{i=1}^{N} \{(1-\delta_i) w_i K_s(x - x_{(i)})\}$$

where N = `n_observations`, δ_i is the i-th observation's censor code (1 = censored, 0 = failed), and w_i is the i-th ordered observation's weight, which may be chosen as either $1/(N - i + 1)$, or
$\ln(1 + 1/(N - i + 1))$. Let

$$H(x) = \int_0^x h(s)\ ds$$

The likelihood is given by

$$L = \prod\nolimits_{i=1}^{N} \{h(x_i)^{(1-\delta_i)} \exp(-H(x_{(i)}))\},$$

where Π denotes product. Since the likelihood leads to degenerate estimates, Tanner and Wong (1984) suggest the use of a modified likelihood. The modification consists of deleting observation x_i in the calculation of $h(x_i)$ and $H(x_i)$ when the likelihood term for x_i is computed using the usual optimization techniques. α and β for given k can then be estimated.

Estimates for α and β are computed as follows: for given β, a closed form solution is available for α. The problem is thus reduced to the estimation of β. A grid search for β is first performed. Experience indicates that if the initial estimate of β from this grid search is greater than, say, e^6, then the modified likelihood is degenerate because the hazard rate does not change with time. In this situation, β should be taken to be infinite, and an estimate of α corresponding to infinite β should be directly computed. When the estimate of β from the grid search is less than e^6, a secant algorithm is used to optimize the modified likelihood. The secant algorithm iteration stops when the change in β from one iteration to the next is less than 10^{-5}. Alternatively, the iterations may cease when the value of β becomes greater than e^6, at which point an infinite β with a degenerate likelihood is assumed.

To find the optimum value of the likelihood with respect to k, a user-specified grid of k-values is used. For each grid value, the modified likelihood is optimized with respect to α and β. That grid point, which leads to the smallest likelihood, is taken to be the optimal k.

Programming Notes

1. If sorting of the data is performed by `imsls_f_nonparam_hazard_rate`, then the sorted array will be such that all censored observations at a given

time precede all failures at that time. To specify an arbitrary pattern of censored/failed observations at a given time point, the `isort` = 1 option must be used. In this case, it is assumed that the times have already been sorted from smallest to largest.

2. The smallest value of *k* must be greater than the largest number of tied failures since d_{jk} must be positive for all *j*. (Censored observations are not counted.) Similarly, the largest value of *k* must be less than the total number of failures. If the grid specified for *k* includes values outside the allowable range, then a warning error is issued; but *k* is still optimized over the allowable grid values.

3. The secant algorithm iterates on the transformed parameter $\beta^* = \exp(-\beta)$. This assures a positive β, and it also seems to lead to a more desirable grid search. All results returned to the user are in the original parameterization, however.

4. Since local minimums have been observed in the modified likelihood, it is recommended that more than one grid of initial values for α and β be used.

5. Function `imsls_f_nonparam_hazard_rate` assumes that the hazard grid points are new data points.

Example

The following example is taken from Tanner and Wong (1984). The data are from Stablein, Carter, and Novak (1981) and involve the survival times of individuals with nonresectable gastric carcinoma. Only individuals treated with both radiation and chemotherapy are used. For each value of *k* from 18 to 22 with increment of 2, the default grid search for β is performed. Using the optimal value of β in the grid, the optimal parameter estimates of α and β are computed for each value of *k*. The final solution is the parameter estimates for the value of *k* which optimizes the modified likelihood (`vml`). Because the `iprint` = 1 is in effect, `imsls_f_nonparam_hazard_rate` prints all of the results in the output.

```
#include "imsls.h"

void main ()
{
  int n_observations = 45, iprint = 1, kmin = 18;
  int increment_k = 2, n_k = 3, isort = 1, nmiss, *isorted_censor;
  float *event_times, *haz;
  int n_hazard=100;
  float hazard_min = 0.0, hazard_inc = 10;

  float t[] = { 17.0, 42.0, 44.0, 48.0, 60.0, 72.0, 74.0, 95.0,
                         103.0, 108.0, 122.0, 144.0, 167.0, 170.0, 183.0,
                         185.0, 193.0, 195.0, 197.0, 208.0, 234.0, 235.0,
                         254.0, 307.0, 315.0, 401.0, 445.0, 464.0, 484.0,
                         528.0, 542.0, 567.0, 577.0, 580.0, 795.0, 855.0,
                         882.0, 892.0,1031.0,1033.0,1306.0,1335.0,1366.0,
                         1452.0, 1472.0};
  float censor_codes[] = { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};

  haz = imsls_f_nonparam_hazard_rate I  (n_observations, t,
                                         n_hazard, hazard_min, hazard_inc,
                                         IMSLS_K_GRID, n_k, kmin,
                                              increment_k,
                                         IMSLS_PRINT_LEVEL, iprint,
                                         IMSLS_N_MISSING, &nmiss,
                                         IMSLS_SORT_OPTION, isort,
                                         IMSLS_CENSOR_CODES, censor_codes,
                                         IMSLS_SORTED_EVENT_TIMES,
                                              &event_times,
                                         IMSLS_SORTED_CENSOR_CODES,
                                              &isorted_censor,
                                         0);

  printf ("\nnmiss = %d\n", nmiss);
  imsls_f_write_matrix ("Sorted Event Times", 1, n_observations,
                        event_times, IMSLS_WRITE_FORMAT, "%7.1f", 0);
  imsls_i_write_matrix ("Sorted Censors", 1, n_observations,
                        isorted_censor, 0);
  imsls_f_write_matrix ("Hazard Rates", 1, n_hazard, haz, 0);

}
```

Output

```
              *** Grid search for k =    18 ***
  alpha                  beta                vml
 4.57832              2980.96           -266.805
 4.54312              54.5982            -266.62
 4.33646              20.0855           -265.541
 4.01933              12.1825           -264.001
 3.54274              7.38906            -262.54
 2.99058              4.48169           -262.512
 2.35154              2.71828           -262.634
 1.58417              1.64872           -262.158
0.966332                    1           -262.868

            *** Optimal parameter estimates ***
  alpha                  beta                vml
 1.69515              1.76926           -262.119

              *** Grid search for k =    20 ***
  alpha                  beta                vml
 4.05393              2980.96           -266.526
 4.03284              54.5982           -266.401
 3.90505              20.0855           -265.648
 3.68782              12.1825           -264.402
 3.30434              7.38906           -262.666
 2.82272              4.48169            -262.08
 2.25276              2.71828           -262.445
 1.55578              1.64872           -261.772
0.955586                    1           -262.618
```

```
                    *** Optimal parameter estimates ***
              alpha                     beta                     vml
            1.54053                  1.63155                -261.771

    *** Grid search for k =    22 ***
          alpha                     beta                    vml
          3.65641                 2980.96               -267.595
          3.64159                 54.5982               -267.499
          3.55056                 20.0855               -266.904
          3.38875                 12.1825               -265.859
          3.07147                 7.38906               -264.066
          2.64504                 4.48169               -263.039
           2.1374                 2.71828               -263.335
          1.51261                 1.64872                -262.64
         0.936368                       1               -262.683

                 *** Optimal parameter estimates ***
          alpha                     beta                    vml
          1.34217                 1.45001               -262.561

               *** The final solution     (k =    20) ***
          alpha                     beta                    vml
          1.54053                 1.63155               -261.771

nmiss = 0

                          Sorted Event Times
        1         2         3         4         5         6         7         8
     17.0      42.0      44.0      48.0      60.0      72.0      74.0      95.0

        9        10        11        12        13        14        15        16
    103.0     108.0     122.0     144.0     167.0     170.0     183.0     185.0

       17        18        19        20        21        22        23        24
    193.0     195.0     197.0     208.0     234.0     235.0     254.0     307.0

       25        26        27        28        29        30        31        32
    315.0     401.0     445.0     464.0     484.0     528.0     542.0     567.0

       33        34        35        36        37        38        39        40
    577.0     580.0     795.0     855.0     882.0     892.0    1031.0    1033.0

       41        42        43        44        45
   1306.0    1335.0    1366.0    1452.0    1472.0

                                Sorted Censors
 1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19
 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

20 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1

39  40  41  42  43  44  45
1   1   1   1   1   1   1

                           Hazard Rates
        1            2            3            4            5            6
  0.000962     0.001111     0.001276     0.001451     0.001634     0.001819
```

```
         7          8          9         10         11         12
  0.002004   0.002185   0.002359   0.002523   0.002675   0.002813

        13         14         15         16         17         18
  0.002935   0.003040   0.003126   0.003193   0.003240   0.003266

        19         20         21         22         23         24
  0.003273   0.003260   0.003229   0.003179   0.003114   0.003034

        25         26         27         28         29         30
  0.002941   0.002838   0.002727   0.002612   0.002495   0.002381

        31         32         33         34         35         36
  0.002273   0.002175   0.002084   0.001998   0.001917   0.001841

        37         38         39         40         41         42
  0.001771   0.001709   0.001655   0.001608   0.001569   0.001537

        43         44         45         46         47         48
  0.001510   0.001484   0.001459   0.001435   0.001411   0.001388

        49         50         51         52         53         54
  0.001365   0.001343   0.001323   0.001304   0.001285   0.001266

        55         56         57         58         59         60
  0.001247   0.001228   0.001208   0.001188   0.001167   0.001146

        61         62         63         64         65         66
  0.001125   0.001103   0.001081   0.001060   0.001040   0.001020

        67         68         69         70         71         72
  0.000999   0.000979   0.000958   0.000936   0.000913   0.000891

        73         74         75         76         77         78
  0.000868   0.000845   0.000821   0.000798   0.000775   0.000752

        79         80         81         82         83         84
  0.000730   0.000708   0.000685   0.000662   0.000640   0.000617

        85         86         87         88         89         90
  0.000595   0.000573   0.000552   0.000530   0.000510   0.000490

        91         92         93         94         95         96
  0.000471   0.000452   0.000434   0.000416   0.000399   0.000383

        97         98         99        100
  0.000366   0.000351   0.000336   0.000321
```

Fatal Errors

`IMSLS_ALL_OBSERVATIONS_MISSING`	
	All observations are missing (NaN, not a number) values.

life_tables

Produces population and cohort life tables.

Synopsis

#include `<imsls.h>`

float `*imsls_f_life_tables` (*int* `n_classes`, *float* `age[]`, *float* `a[]`,
int `n_cohort[]`, ..., 0)

The type *double* function is `imsls_d_life_tables`.

Required Arguments

int `n_classes` (Input)
Number of age classes.

float `age[]` (Input)
Array of length `n_classes` + 1 containing the lowest age in each age interval, and in `age[n_classes]`, the endpoint of the last age interval. Negative `age[0]` indicates that the age intervals are all of length `|age[0]|` and that the initial age interval is from 0.0 to `|age[0]|`. In this case, all other elements of `age` need not be specified. `age[n_classes]` need not be specified when getting a cohort table.

float `a[]` (Input)
Array of length `n_classes` containing the fraction of those dying within each interval who die before the interval midpoint. A common choice for all `a[i]` is 0.5. This choice may also be specified by setting `a[0]` to any negative value. In this case, the remaining values of `a` need not be specified.

int `n_cohort[]` (Input)
Array of length `n_classes` containing the cohort sizes during each interval. If the `IMSL_POPULATION_LIFE_TABLE` option is used, then `n_cohort[i]` contains the size of the population at the midpoint of interval `i`. Otherwise, `n_cohort[i]` contains the size of the cohort at the beginning of interval `i`. When requesting a population table, the population sizes in `n_cohort` may need to be adjusted to correspond to the number of deaths in `n_deaths`. See the Description section for more information.

Return Value

Pointer to an array of length `n_classes` by 12 containing the life table. The function returns a cohort table by default. If the `IMSL_POPULATION_LIFE_TABLE` option is used, a population table is returned. Entries in the *i*th row are for the age interval defined by `age[i]`. Column definitions are described in the following table.

Column	Description
0	Lowest age in the age interval.
1	Fraction of those dying within the interval who die before the interval midpoint.
2	Number surviving to the beginning of the interval.
3	Number of deaths in the interval.
4	Death rate in the interval. For cohort table, this column is set to NaN (not a number).
5	Proportion dying in the interval.
6	Standard error of the proportion dying in the interval.
7	Proportion of survivors at the beginning of the interval.
8	Standard error of the proportion of survivors at the beginning of the interval.
9	Expected lifetime at the beginning of the interval.
10	Standard error of the expected life at the beginning of the interval.
11	Total number of time units lived by all of the population in the interval.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_life_tables` (*int* `n_classes`, *float* `age[]`,
float `a[]`, *int* `n_cohort[]`,
`IMSLS_RETURN_USER`, *float* `table[]`,
`IMSLS_PRINT_LEVEL`, *int* `iprint`,
`IMSLS_POPULATION_SIZE`, *int* `initial_pop`,
`IMSLS_POPULATION_LIFE_TABLE`, *int* `*n_deaths`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `table[]` (Output)
If specified, `table` is an user-specified array of length `n_classes*12` containing the life table.

`IMSLS_PRINT_LEVEL`, *int* `iprint` (Input)
Printing option.
Default: `iprint` = 0.

Iprint	**Action**
0	No printing is performed.
1	The life table is printed.

IMSLS_POPULATION_SIZE, *int* initial_pop (Input)

The population size at the beginning of the first age interval in requesting population table. A default value of 10,000 is used to allow easy entry of n_cohorts and n_deaths when numbers are available as percentages.
Default: initial_pop = 10000.

IMSLS_POPULATION_LIFE_TABLE, *int* *n_deaths (Input)

Compute a population table. n_deaths is an array of length n_classes containing the number of deaths in each age interval.

Description

Function imsls_f_life_tables computes population (current) or cohort life tables based upon the observed population sizes at the middle (for population table) or the beginning (for cohort table) of some userspecified age intervals. The number of deaths in each of these intervals must also be observed.

The probability of dying prior to the middle of the interval, given that death occurs somewhere in the interval, may also be specified. Often, however, this probability is taken to be 0.5. For a discussion of the probability models underlying the life table here, see the references.

Let t_i, for $i = 0, 1, \ldots, t_n$ denote the time grid defining the n age intervals, and note that the length of the age intervals may vary. Following Gross and Clark (1975, page 24), let d_i denote the number of individuals dying in age interval i, where age interval i ends at time t_i. For population table, the death rate at the middle of the interval is given by $r_i = d_i/(M_ih_i)$, where M_i is the number of individuals alive at the middle of the interval, and $h_i = t_i - t_{i-1}$, $t_0 = 0$. The number of individuals alive at the beginning of the interval may be estimated by $P_i = M_i + (1 - a_i)d_i$ where a_i is the probability that an individual dying in the interval dies prior to the interval midpoint. For cohort table, P_i is input directly while the death rate in the interval, r_i, is not needed.

The probability that an individual dies during the age interval from t_{i-1} to t_i is given by $q_i = d_i/P_i$. It is assumed that all individuals alive at the beginning of the last interval die during the last interval. Thus, $q_n = 1.0$. The asymptotic variance of q_i can be estimated by

$$\sigma_i^2 = q_i(1-q_i)/P_i$$

For population table, the number of individuals alive in the middle of the time interval (input in n_cohort[i]) must be adjusted to correspond to the number of deaths observed in the interval. Function imsls_f_life_tables assumes that the number of deaths observed in interval h_i occur over a time period equal to h_i. If d_i is measured over a period u_i, where $u_i \neq d_i$, then n_cohort[i] must be adjusted to correspond to d_i by multiplication by u_i/h_i, i.e., the value M_i input into imsls_f_life_tables as n_cohort[i] is computed as

$$M_i^* = M_i u_i / h_i$$

Let S_i denote the number of survivors at time t_i from a hypothetical (for population table) or observed (for cohort table) population. Then, S_0 = `initial_pop` for population table, and S_0 = `n_cohort[0]` for cohort table, and S_i is given by $S_i = S_{i-1} - \delta_{i-1}$ where $\delta_i = S_i q_i$ is the number of individuals who die in the i-th interval. The proportion of survivors in the interval is given by $V_i = S_i/S_0$ while the asymptotic variance of V_i can be estimated as follows.

$$\text{var}(V_i) = V_i^2 \sum_{j=1}^{i-1} \frac{\sigma_j^2}{(1-q_j)^2}$$

The expected lifetime at the beginning of the interval is calculated as the total lifetime remaining for all survivors alive at the beginning of the interval divided by the number of survivors at the beginning of the interval. If e_i denotes this average expected lifetime, then the variance of e_i can be estimated as (see Chiang 1968)

$$\text{var}(e_i) = \frac{\sum_{j=i}^{n-1} P_j^2 \sigma_j^2 [e_{j+1} + h_{j+1}(1-a_j)]^2}{P_j^2}$$

where $\text{var}(e_n) = 0.0$.

Finally, the total number of time units lived by all survivors in the time interval can be estimated as:

$$U_i = h_i[S_i - \delta_i(1-a_i)]$$

Example

The following example is taken from Chiang (1968). The cohort life table has thirteen equally spaced intervals, so `age[0]` is set to −5.0. Similarly, the probabilities of death prior to the middle of the interval are all taken to be 0.5, so `a[0]` is set to −1.0. Since `IMSLS_PRINT_LEVEL` option is used, `imsls_f_life_tables` prints the life table.

```
#include "imsls.h"

#define N_CLASSES 13

void main ()
{
  int iprint = 1;
  int n_cohort[] =
    { 270, 268, 264, 261, 254, 251, 248, 232, 166, 130, 76, 34, 13 };
  float age[N_CLASSES + 1], a[N_CLASSES];
  float *result;

  age[0] = -5.0;
  a[0] = -1.0;
  result = imsls_f_life_tables (N_CLASSES, age, a, n_cohort,
                                IMSLS_PRINT_LEVEL, iprint, 0);
}
```

Output

```
                              Life Table
Age Class          Age       PDHALF        Alive       Deaths  Death Rate
        1            0          0.5          270            2  ..........
        2            5          0.5          268            4  ..........
        3           10          0.5          264            3  ..........
        4           15          0.5          261            7  ..........
        5           20          0.5          254            3  ..........
        6           25          0.5          251            3  ..........
        7           30          0.5          248           16  ..........
        8           35          0.5          232           66  ..........
        9           40          0.5          166           36  ..........
       10           45          0.5          130           54  ..........
       11           50          0.5           76           42  ..........
       12           55          0.5           34           21  ..........
       13           60          0.5           13           13  ..........

Age Class         P(D)   Std(P(D))         P(S)   Std(P(S))     Lifetime
        1     0.007407    0.005218            1           0        43.19
        2      0.01493    0.007407       0.9926    0.005218        38.49
        3      0.01136    0.006523       0.9778    0.008971        34.03
        4      0.02682        0.01       0.9667     0.01092         29.4
        5      0.01181    0.006779       0.9407     0.01437        25.14
        6      0.01195    0.006859       0.9296     0.01557        20.41
        7      0.06452      0.0156       0.9185     0.01665        15.63
        8       0.2845     0.02962       0.8593     0.02116        11.53
        9       0.2169     0.03199       0.6148     0.02962        10.12
       10       0.4154     0.04322       0.4815     0.03041        7.231
       11       0.5526     0.05704       0.2815     0.02737        5.592
       12       0.6176     0.08334       0.1259     0.02019        4.412
       13            1           0      0.04815     0.01303          2.5

Age Class    Std(Life)  Time Units
        1       0.6993        1345
        2       0.6707        1330
        3        0.623        1313
        4        0.594        1288
        5       0.5403        1263
        6       0.5237        1248
        7       0.5149        1200
        8       0.4982         995
        9       0.4602         740
       10       0.4328         515
       11       0.4361         275
       12       0.4167       117.5
       13            0        32.5
```

Chapter 11: Probability Distribution Functions and Inverses

Routines

Usage Notes

Definitions and discussions of the terms basic to this chapter can be found in Johnson and Kotz (1969, 1970a, 1970b). These are also good references for the specific distributions.

In order to keep the calling sequences simple, whenever possible, the subprograms described in this chapter are written for standard forms of statistical distributions. Hence, the number of parameters for any given distribution may be fewer than the number often associated with the distribution. For example, while a gamma distribution is often characterized by two parameters (or even a third, "location"), there is only one parameter that is necessary, the "shape". The "scale" parameter can be used to scale the variable to the standard gamma distribution. Also, the functions relating to the normal distribution, `imsls_f_normal_cdf` (page 748) and `imsls_f_normal_inverse_cdf` (page 750), are for a normal distribution with mean equal to zero and variance equal to one. For other means and variances, it is very easy for the user to standardize the variables by subtracting the mean and dividing by the square root of the variance.

The *distribution function* for the (real, single-valued) random variable X is the function F defined for all real x by

$$F(x) = \text{Prob}(X \le x)$$

where Prob(·) denotes the probability of an event. The distribution function is often called the *cumulative distribution function* (CDF).

For distributions with finite ranges, such as the beta distribution, the CDF is 0 for values less than the left endpoint and 1 for values greater than the right endpoint. The subprograms described in this chapter return the correct values for the distribution functions when values outside of the range of the random variable are input, but warning error conditions are set in these cases.

Discrete Random Variables

For discrete distributions, the function giving the probability that the random variable takes on specific values is called the *probability function*, defined by

$$p(x) = \text{Prob}(X = x)$$

The "`PR`" routines described in this chapter evaluate probability functions.

The CDF for a discrete random variable is

$$F(x) = \sum_A p(k)$$

where A is the set such that $k \le x$. The "DF" routines in this chapter evaluate cumulative distribution functions. Since the distribution function is a step function, its inverse does not exist uniquely.

Continuous Distributions

For continuous distributions, a probability function, as defined above, would not be useful because the probability of any given point is 0. For such distributions, the useful analog is the *probability density function* (PDF). The integral of the PDF is the probability over the interval, if the continuous random variable X has PDF f, then

$$\text{Prob}(a < X \le b) = \int_a^b f(x)dx$$

The relationship between the CDF and the PDF is

$$F(x) = \int_{-\infty}^{x} f(t)dt\ .$$

The "`_cdf`" functions described in this chapter evaluate cumulative distribution functions.

For (absolutely) continuous distributions, the value of F(x) uniquely determines x within the support of the distribution. The "`_inverse_cdf`" functions described in this chapter compute the inverses of the distribution functions, that is, given F(x) (called "P" for "probability"), a routine such as `imsls_f_beta_inverse_cdf` (page 731) computes `x`. The inverses are defined only over the open interval (0,1).

Additional Comments

Whenever a probability close to 1.0 results from a call to a distribution function or is to be input to an inverse function, it is often impossible to achieve good accuracy because of the nature of the representation of numeric values. In this case, it may be better to work with the complementary distribution function (one minus the distribution function). If the distribution is symmetric about some point (as the normal distribution, for example) or is reflective about some point (as the beta distribution, for example), the complementary distribution function has a simple relationship with the distribution function. For example, to evaluate the standard normal distribution at 4.0, using `imsls_f_normal_inverse_cdf` (page 750) 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` (page 730) at 0.7, 0.999953 is obtained. A more precise result is obtained by evaluating `imsls_f_beta_cdf` with parameters 10 and 2 at 0.3. This yields 4.72392E-5. (In both of these examples, it is wise not to trust the last digit.)

Many of the algorithms used by routines in this chapter are discussed by Abramowitz and Stegun (1964). The algorithms make use of various expansions and recursive relationships and often use different methods in different regions.

Cumulative distribution functions are defined for all real arguments, however, if the input to one of the distribution functions in this chapter is outside the range of the random variable, an error of Type 1 is issued, and the output is set to zero or one, as appropriate. A Type 1 error is of lowest severity, a “note”, and, by default, no printing or stopping of the program occurs. The other common errors that occur in the routines of this chapter are Type 2, “alert”, for a function value being set to zero due to underflow, Type 3, “warning”, for considerable loss of accuracy in the result returned, and Type 5, “terminal”, for incorrect and/or inconsistent input, complete loss of accuracy in the result returned, or inability to represent the result (because of overflow). When a Type 5 error occurs, the result is set to NaN (not a number, also used as a missing value code).

binomial_cdf

Evaluates the binomial distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_binomial_cdf` (*int* `k`, *int* `n`, *float* `p`)

The type *double* function is `imsls_d_binomial_cdf`.

Required Arguments

int `k` (Input)
Argument for which the binomial distribution function is to be evaluated.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial.

Return Value

The probability that *k* or fewer successes occur in *n* independent Bernoulli trials, each of which has a probability *p* of success.

Description

The `imsls_f_binomial_cdf` function evaluates the distribution function of a binomial random variable with parameters *n* and *p*. It does this by summing probabilities of the random variable taking on the specific values in its range. These probabilities are computed by the recursive relationship:

$$Pr\left(X = j\right) = \frac{\left(n + 1 - j\right) p}{j\left(1 - p\right)} Pr\left(X = j - 1\right)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0 if *k* is not greater than $n \times p$; otherwise, they are computed backward from *n*. The smallest positive machine number, ε, is used as the starting value for summing the probabilities, which are rescaled by $(1 - p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is used.

For the special case of $p = 0$, `imsls_f_binomial_cdf` is set to 1; for the case $p = 1$, `imsls_f_binomial_cdf` is set to 1 if $k = n$ and is set to 0 otherwise.

Example

Suppose *X* is a binomial random variable with $n = 5$ and $p = 0.95$. In this example, the function finds the probability that *X* is less than or equal to 3.

```
#include <imsls.h>

void main()
{
    int         k = 3;
    int         n = 5;
    float       p = 0.95;
    float       pr;

    pr = imsls_f_binomial_cdf(k,n,p);
    printf("Pr(x <= 3) = %6.4f\n", pr);
}
```

Output

```
Pr(x <= 3) = 0.0226
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since “k” = # is less than zero, the distribution function is set to zero.
`IMSLS_GREATER_THAN_N`	The input argument, *k*, is greater than the number of Bernoulli trials, *n*.

binomial_pdf

Evaluates the binomial probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_binomial_pdf` (*int* `k`, *int* `n`, *float* `p`,..., 0)

The type *double* function is `imsls_d_binomial_pdf`.

Required Arguments

int `k` (Input)
Argument for which the binomial probability function is to be evaluated.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial.

Return Value

The probability that a binomial random variable takes on a value equal to `k`.

Description

The function `imsls_f_binomial_pdf` evaluates the probability that a binomial random variable with parameters n and p takes on the value k. It does this by computing probabilities of the random variable taking on the values in its range less than (or the values greater than) k. These probabilities are computed by the recursive relationship

$$\Pr(X = j) = \frac{(n+1-j)p}{j(1-p)} \Pr(X = j-1)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0, if k is not greater than n times p, and are computed backward from n, otherwise. The smallest positive machine number, ε, is used as the starting value for computing the probabilities, which are rescaled by $(1 - p)^n ε$ if forward computation is performed and by $p^n ε$ if backward computation is done.

For the special case of $p = 0$, `imsls_f_binomial_pdf` is set to 0 if k is greater than 0 and to 1 otherwise; and for the case $p = 1$, `imsls_f_binomial_pdf` is set to 0 if k is less than n and to 1 otherwise.

Example 1

Suppose X is a binomial random variable with $n = 5$ and $p = 0.95$. In this example, we find the probability that X is equal to 3.

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

void main()
{
  int k, n;
  float p, prob;

  k = 3;
  n = 5;
  p = 0.95;
  prob = imsls_f_binomial_pdf(k, n, p);

  printf("The probability that X is equal to 3 is %f\n", prob);
 }
```

Output

```
The probability that X is equal to 3 is 0.021434
```

hypergeometric_cdf

Evaluates the hypergeometric distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_hypergeometric_cdf` (*int* `k`, *int* `n`, *int* `m`, *int* `l`)

The type *double* function is `imsls_d_hypergeometric_cdf`.

Required Arguments

int `k` (Input)
Argument for which the hypergeometric distribution function is to be evaluated.

int `n` (Input)
Sample size. Argument `n` must be greater than or equal to `k`.

int `m` (Input)
Number of defectives in the lot.

int `l` (Input)
Lot size. Argument `l` must be greater than or equal to `n` and `m`.

Return Value

The probability that *k* or fewer defectives occur in a sample of size *n* drawn from a lot of size *l* that contains *m* defectives.

Description

Function `imsls_f_hypergeometric_cdf` evaluates the distribution function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *x* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type. The probability function is

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

where $i = max\ (0, n - l + m)$.

If *k* is greater than or equal to *i* and less than or equal to min (*n*, *m*), `imsls_f_hypergeometric_cdf` sums the terms in this expression for *j* going from *i* up to *k*; otherwise, 0 or 1 is returned, as appropriate. To avoid rounding in the accumulation, `imsls_f_hypergeometric_cdf` performs the summation differently, depending on whether or not *k* is greater than the mode of the distribution, which is the greatest integer less than or equal to $(m + 1)\ (n + 1)/(l + 2)$.

Example

Suppose *X* is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. In this example, evaluate the distribution function at 7.

```
#include <imsls.h>

void main()
{
    int         k = 7;
    int         l = 1000;
    int         m = 70;
    int         n = 100;
    float       p;

    p = imsls_f_hypergeometric_cdf(k,n,m,l);
    printf("\nPr (x <= 7) = %6.4f", p);
}
```

Output

```
Pr (x <= 7) = 0.599
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since “k” = # is less than zero, the distribution function is set to zero.
`IMSLS_K_GREATER_THAN_N`	The input argument, *k*, is greater than the sample size.

Fatal Errors

`IMSLS_LOT_SIZE_TOO_SMALL`	Lot size must be greater than or equal to *n* and *m*.

hypergeometric_pdf

Evaluates the hypergeometric probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_hypergeometric_pdf` (*int* `k`, *int* `n`, *int* `m`, *int* `l`)

The type *double* function is `imsls_d_hypergeometric_pdf`.

Required Arguments

int `k` (Input)
Argument for which the hypergeometric probability function is to be evaluated.

int `n` (Input)
Sample size. `n` must be greater than zero and greater than or equal to `k`.

int `m` (Input)
Number of defectives in the lot.

int `l` (Input)
Lot size. `l` must be greater than or equal to `n` and `m`.

Return Value

The probability that a hypergeometric random variable takes a value equal to `k`. This value is the probability that exactly `k` defectives occur in a sample of size `n` drawn from a lot of size `l` that contains `m` defectives.

Description

The function `imsls_f_hypergeometic_pdf` evaluates the probability function of a hypergeometric random variable with parameters *n*, *l*, and *m*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *n* that is drawn without replacement from a population of size *l* containing *m* items of this type. The probability function is

$$\Pr(X = k) = \frac{\binom{m}{k}\binom{l-m}{n-k}}{\binom{l}{n}} \quad \text{for } k = i, i+1, i+2, \ldots \min(n, m)$$

where $i = \max(0, n - l + m)$. `imsls_f_hypergeometic_pdf` evaluates the expression using log gamma functions.

Example

Suppose *X* is a hypergeometric random variable with $n = 100$, $l = 1000$, and $m = 70$. In this example, we evaluate the probability function at 7.

```
include "imsls.h"

void main()

{

  int k=7, n=100, l=1000, m=70;

  float pr;

  pr = imsls_f_hypergeometic_pdf(k, n, m, l);

  printf("  The probability that X is equal to 7 is %6.4f\n", pr);

}
```

Output

```
  The probability that X is equal to 7 is 0.1628
```

poisson_cdf

Evaluates the Poisson distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_poisson_cdf` (*int* `k`, *float* `theta`)

The type *double* function is `imsls_d_poisson_cdf`.

Required Arguments

int `k` (Input)
Argument for which the Poisson distribution function is to be evaluated.

float `theta` (Input)
Mean of the Poisson distribution. Argument `theta` must be positive.

Return Value

The probability that a Poisson random variable takes a value less than or equal to *k*.

Description

Function `imsls_f_poisson_cdf` evaluates the distribution function of a Poisson random variable with parameter `theta`. The mean of the Poisson random variable, `theta`, must be positive. The probability function (with θ = `theta`) is as follows:

$$f(x) = e^{-\theta}\theta^{x} / x!, \qquad \text{for } x = 0, 1, 2, \ldots$$

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. Function `imsls_f_poisson_cdf` uses the recursive relationship

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

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

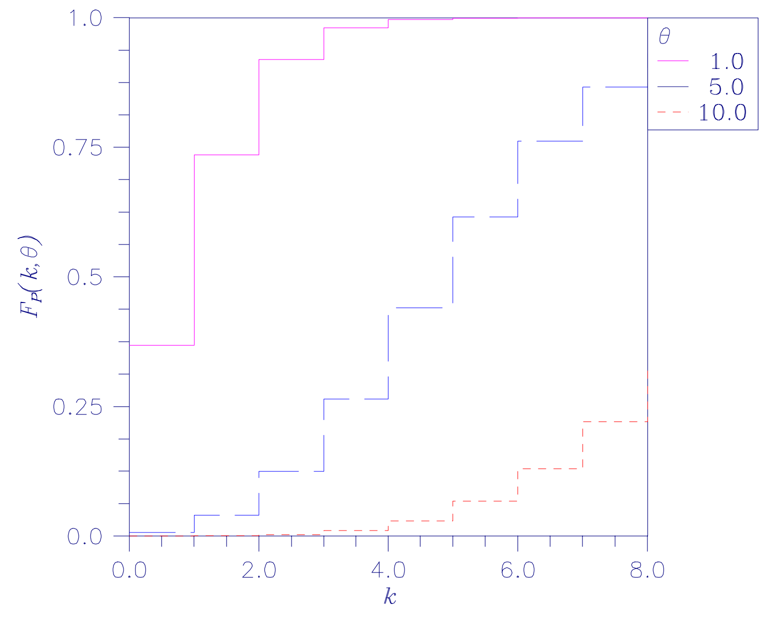


Figure 11-1 Plot of F_p (k, θ)

Example

Suppose X is a Poisson random variable with θ = 10. In this example, we evaluate the probability that X is less than or equal to 7.

```
#include <imsls.h>

void main()
{
    int         k = 7;
    float       theta = 10.0;
    float       p;
    p = imsls_f_poisson_cdf(k, theta);
    printf("Pr(x <= 7) = %6.4f\n", p);
}
```

Output

```
Pr(x <= 7) = 0.2202
```

Informational Errors

`IMSLS_LESS_THAN_ZERO`	Since “k” = # is less than zero, the distribution function is set to zero.

poisson_pdf

Evaluates the Poisson probability function.

Synopsis

#include `<imsls.h>`

float `imsls_f_poisson_pdf` (*int* `k`, *float* `theta`)

The type *double* function is `imsls_d_poisson_pdf`.

Required Arguments

int `k` (Input)
Argument for which the Poisson distribution function is to be evaluated.

float `theta` (Input)
Mean of the Poisson distribution. `theta` must be positive.

Return Value

Function value, the probability that a Poisson random variable takes a value equal to `k`.

Description

Function `imsls_f_poisson_pdf` evaluates the probability function of a Poisson random variable with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with θ = `theta`) is

$$f(x) = e^{-\theta}\,\theta^{k}/k!, \quad \text{for } k = 0, 1, 2, \ldots$$

`imsls_f_poisson_pdf` evaluates this function directly, taking logarithms and using the log gamma function.

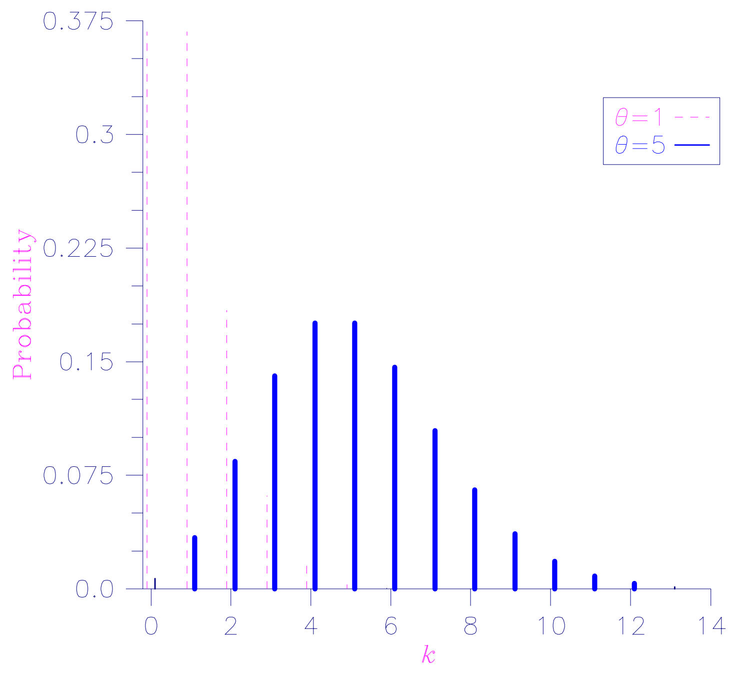


Figure 11-2   Poisson Probability Function

Example

Suppose X is a Poisson random variable with θ = 10. In this example, we evaluate the probability function at 7.

```
#include "imsls.h"

void main () {
  int k = 7;
  float theta = 10.0;

  printf ("The probability that X is equal to 7 is %g.\n",
         imsls_f_poisson_pdf (k, theta));
}
```

Output

```
The probability that X is equal to 7 is 0.0900792.
```

beta_cdf

Evaluates the beta probability distribution function.

Synopsis

#include <imsls.h>

float imsls_f_beta_cdf (*float* x, *float* pin, *float* qin)

The type *double* function is imsls_d_beta_cdf.

Required Arguments

float x (Input)
Argument for which the beta probability distribution function is to be evaluated.

float pin (Input)
First beta distribution parameter. Argument pin must be positive.

float qin (Input)
Second beta distribution parameter. Argument qin must be positive.

Return Value

The probability that a beta random variable takes on a value less than or equal to *x*.

Description

Function imsls_f_beta_cdf evaluates the distribution function of a beta random variable with parameters pin and qin. This function is sometimes called the incomplete beta ratio and, with p = pin and q = qin, is denoted by $I_x(p, q)$. It is given by

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

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function by $I_x(p, q)$ is the probability that the random variable takes a value less than or equal to x.

The integral in the expression above is called the incomplete beta function and is denoted by $\beta_x(p, q)$. The constant in the expression is the reciprocal of the beta function (the incomplete function evaluated at 1) and is denoted by $\beta(p, q)$.

Function imsls_f_beta_cdf uses the method of Bosten and Battiste (1974).

Example

Suppose *X* is a beta random variable with parameters 12 and 12 (*X* has a symmetric distribution). This example finds the probability that *X* is less than 0.6 and the probability that *X* is between 0.5 and 0.6. (Since *X* is a symmetric beta random variable, the probability that it is less than 0.5 is 0.5.)

```
#include <imsls.h>

main()
{
        float           p, pin, qin, x;

        pin = 12.0;
        qin = 12.0;
        x = 0.6;
        p = imsls_f_beta_cdf(x, pin, qin);
        printf("The probability that X is less than 0.6 is %6.4f\n",
                p);
        x = 0.5;
        p -= imsls_f_beta_cdf(x, pin, qin);
        printf("The probability that X is between 0.5 and");
        printf(" 0.6 is %6.4f\n", p);
}
```

Output

```
The probability that X is less than 0.6 is 0.8364
The probability that X is between 0.5 and 0.6 is 0.3364
```

beta_inverse_cdf

Evaluates the inverse of the beta distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta_inverse_cdf` (*float* `p`, *float* `pin`, *float* `qin`)

The type *double* function is `imsls_d_beta_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the beta distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

Function `imsls_f_beta_inverse_cdf` returns the inverse distribution function of a beta random variable with parameters `pin` and `qin`.

Description

With P = `p`, p = `pin`, and q = `qin`, the beta_inverse_cdf returns x such that

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

where Γ (·) is the gamma function. The probability that the random variable takes a value less than or equal to x is P.

Example

Suppose X is a beta random variable with parameters 12 and 12 (X has a symmetric distribution). In this example, we find the value x such that the probability that X is less than or equal to x is 0.9.

```
#include <imsls.h>

main()
{
        float           p, pin, qin, x;


        pin = 12.0;
        qin = 12.0;
        p = 0.9;
        x = imsls_f_beta_inverse_cdf(p, pin, qin);
        printf(" X is less than %6.4f with probability 0.9.\n",
                x);
}
```

Output

```
X is less than 0.6299 with probability 0.9.
```

bivariate_normal_cdf

Evaluates the bivariate normal distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_bivariate_normal_cdf` (*float* `x`, *float* `y`, *float* `rho`)

The type *double* function is `imsls_d_bivariate_normal_cdf`.

Required Arguments

float `x` (Input)
The x-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `y` (Input)
The y-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

float `rho` (Input)
Correlation coefficient.

Return Value

The probability that a bivariate normal random variable with correlation `rho` takes a value less than or equal to x and less than or equal to y.

Description

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

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

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

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

transform $(U, V)^T$ to a vector with zero means and unit variances. The input to `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 |ρ| = 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` (page 748).

Example

Suppose (X, Y) is a bivariate normal random variable with mean (0, 0) and variance-covariance matrix as follows:

$$\begin{bmatrix} 1.0 & 0.9 \\ 0.9 & 1.0 \end{bmatrix}$$

In this example, we find the probability that *X* is less than –2.0 and *Y* is less than 0.0.

```
#include <imsls.h>

main()
{
        float           p, rho, x, y;

        x = -2.0;
        y = 0.0;
        rho = 0.9;
        p = imsls_f_bivariate_normal_cdf(x, y, rho);
        printf(" The probability that X is less than -2.0\n"
               " and Y is less than 0.0 is %6.4f\n", p);

}
```

Output

```
The probability that X is less than -2.0
and Y is less than 0.0 is 0.0228
```

chi_squared_cdf

Evaluates the chi-squared distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_chi_squared_cdf` (*float* `chi_squared`, *float* `df`)

The type *double* function is `imsls_d_chi_squared_cdf`.

Required Arguments

float `chi_squared` (Input)
Argument for which the chi-squared distribution function is to be evaluated.

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. Argument `df` must be greater than or equal to 0.5.

Return Value

The probability that a chi-squared random variable takes a value less than or equal to `chi_squared`.

Description

Function `imsls_f_chi_squared_cdf` evaluates the distribution function, *F*, of a chi-squared random variable x = `chi_squared` with ν = `df`. Then,

$$F(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

For $\nu > 65$, `imsls_f_chi_squared_cdf` uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) to the normal distribution, and function `imsls_f_normal_cdf` is used to evaluate the normal distribution function.

For $\nu \le 65$, `imsls_f_chi_squared_cdf` uses series expansions to evaluate the distribution function. If $x <$ `max` $(\nu / 2, 26)$, `imsls_f_chi_squared_cdf` uses the series 6.5.29 in Abramowitz and Stegun (1964); otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.

Example

Suppose *X* is a chi-squared random variable with two degrees of freedom. In this example, we find the probability that *X* is less than 0.15 and the probability that *X* is greater than 3.0.

```
#include <imsls.h>

void main()
{
    float       chi_squared = 0.15;
    float       df = 2.0;
    float       p;

    p    = imsls_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared\n",
        "with 2 df is less than 0.15 is", p);

    chi_squared = 3.0;
    p    = 1.0 - imsls_f_chi_squared_cdf(chi_squared, df);
    printf("%s %s %6.4f\n", "The probability that chi-squared\n",
        "with 2 df is greater than 3.0 is", p);
}
```

Output

```
The probability that chi-squared
 with 2 df is less than 0.15 is 0.0723
The probability that chi-squared
 with 2 df is greater than 3.0 is 0.2231
```

Informational Errors

`IMSLS_ARG_LESS_THAN_ZERO`	Since "chi_squared" = # is less than zero, the distribution function is zero at "chi_squared."

Alert Errors

`IMSLS_NORMAL_UNDERFLOW`	Using the normal distribution for large degrees of freedom, underflow would have occurred.

chi_squared_inverse_cdf

Evaluates the inverse of the chi-squared distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_chi_squared_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsls_d_chi_squared_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the chi-squared distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Number of degrees of freedom of the chi-squared distribution. Argument `df` must be greater than or equal to 0.5.

Return Value

The inverse at the chi-squared distribution function evaluated at `p`. The probability that a chi-squared random variable takes a value less than or equal to `imsls_f_chi_squared_inverse_cdf` is `p`.

Description

Function `imsls_f_chi_squared_inverse_cdf` evaluates the inverse distribution function of a chi-squared random variable with ν = `df` and with probability *p*. That is, it determines x = `imsls_f_chi_squared_inverse_cdf` (`p`, `df`), such that

$$p = \frac{1}{2^{\nu/2}\Gamma\left(\nu/2\right)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where Γ (·) is the gamma function. The probability that the random variable takes a value less than or equal to *x* is *p*.

For $\nu < 40$, `imsls_f_chi_squared_inverse_cdf` uses bisection (if $\nu \le 2$ or $p > 0.98$) or regula falsi to find the point at which the chi-squared distribution function is equal to *p*. The distribution function is evaluated using IMSL function `imsls_f_chi_squared_cdf`.

For $40 \le \nu < 100$, a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.18) to the normal distribution is used. IMSL function `imsls_f_normal_cdf` is used to evaluate the inverse of the normal distribution function. For $\nu \ge 100$, the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, Equation 26.4.17) is used.

Example

In this example, we find the 99-th percentage point of a chi-squared random variable with 2 degrees of freedom and of one with 64 degrees of freedom.

```
#include <imsls.h>

void main ()
{
    float       df, x;
    float       p = 0.99;

    df = 2.0;
    x  = imsls_f_chi_squared_inverse_cdf(p, df);
    printf("For p = .99 with  2 df, x = %7.3f.\n", x);

    df = 64.0;
    x  = imsls_f_chi_squared_inverse_cdf(p,df);
    printf("For p = .99 with 64 df, x = %7.3f.\n", x);
}
```

Output

```
For p = .99 with  2 df, x =   9.210.
For p = .99 with 64 df, x =  93.217.
```

Warning Errors

`IMSLS_UNABLE_TO_BRACKET_VALUE`	The bounds that enclose "p" could not be found. An approximation for `imsls_f_chi_squared_inverse_cdf` is returned.
`IMSLS_CHI_2_INV_CDF_CONVERGENCE`	The value of the inverse chi-squared could not be found within a specified number of iterations. An approximation for `imsls_f_chi_squared_inverse_cdf` is returned.

non_central_chi_sq

Evaluates the noncentral chi-squared distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_chi_sq` *(float* `chi_squared`*, float* `df` *, float* `delta`*)*

The type *double* function is `imsls_d_non_central_chi_sq`.

Required Arguments

float `chi_squared` (Input)
Argument for which the noncentral chi-squared distribution function is to be evaluated.

float `df` (Input)
Number of degrees of freedom of the noncentral chi-squared distribution. Argument `df` must be greater than or equal to 0.5

float `delta` (Input)
The noncentrality parameter. `delta` must be nonnegative, and `delta + df` must be less than or equal to 200,000.

Return Value

The probability that a noncentral chi-squared random variable takes a value less than or equal to `chi_squared`.

Description

Function `imsls_f_non_central_chi_sq` evaluates the distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, that is, with v = `df`, λ = `alam`, and x = `chi_squared`,

$$non\_central\_chi\_sq(x) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^x \frac{t^{(v+2i)/2-1}e^{-t/2}}{2^{(v+2i)/2}\Gamma\left(\frac{v+2i}{2}\right)} dt$$

where Γ(·) is the gamma function. This is a series of central chi-squared distribution functions with Poisson weights. The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

The noncentral chi-squared random variable can be defined by the distribution function above, or alternatively and equivalently, as the sum of squares of independent normal random variables. If Y_i have independent normal distributions with means μ_i and variances equal to one and

$$X = \sum_{i=1}^{n} Y_i^2$$

then X has a noncentral chi-squared distribution with n degrees of freedom and noncentrality parameter equal to

$$\sum_{i=1}^{n} \mu_i^2$$

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the chi-squared distribution.

Function `imsls_f_non_central_chi_sq` determines the point at which the Poisson weight is greatest, and then sums forward and backward from that point, terminating when the additional terms are sufficiently small or when a maximum of 1000 terms have been accumulated. The recurrence relation 26.4.8 of Abramowitz and Stegun (1964) is used to speed the evaluation of the central chi-squared distribution functions.

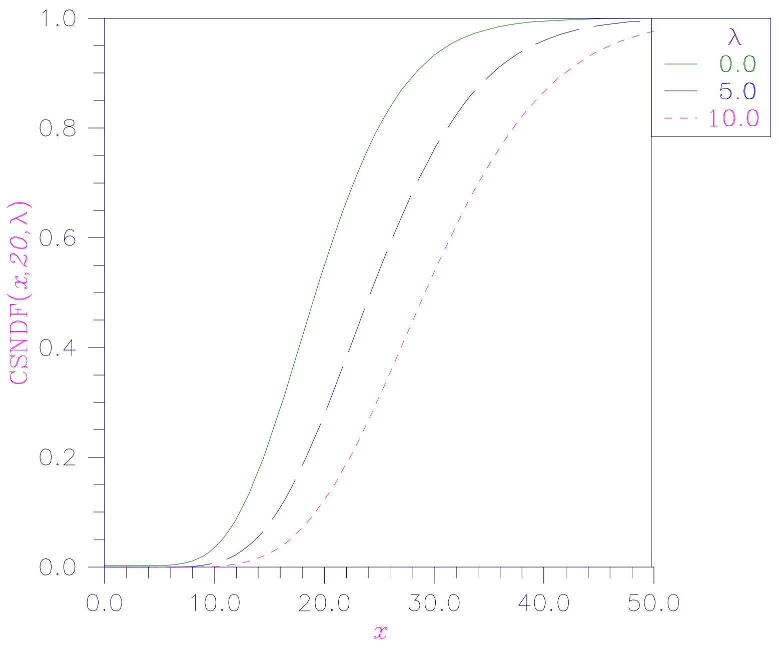


Figure 11-3 Noncentral Chi-squared Distribution Function

Example

In this example, `imsls_f_non_central_chi_sq` is used to compute the probability that a random variable that follows the noncentral chi-squared distribution with noncentrality parameter of 1 and with 2 degrees of freedom is less than or equal to 8.642.

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

```
#include <stdio.h>
void main()
{
        float chsq = 8.642;
        float df = 2.0;
        float alam = 1.0;
        float p;
        p = imsls_f_non_central_chi_sq(chsq, df, alam);
        printf("The probability that a noncentral chi-squared random\n"
       "variable with %2.0f df and noncentrality parameter %3.1f is less\n"
       "than %5.3f is %5.3f.\n", df, alam, chsq, p);
}
```

Output

```
The probability that a noncentral chi-squared random
variable with 2 df and noncentrality parameter 1.0 is less
than 8.642 is 0.950
```

non_central_chi_sq_inv

Evaluates the inverse of the noncentral chi-squared function.

Synopsis

#include <`imsls.h`>

float `imsls_f_non_central_chi_sq_inv` *(float* `p`*, float* `df`*, float* `delta`*)*

The type *double* function is `imsls_d_non_central_chi_sq_inv`.

Required Arguments

float p (Input)
: Probability for which the inverse of the noncentral chi-squared distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
: Number of degrees of freedom of the noncentral chi-squared distribution. Argument df must be greater than or equal to 0.5

float `delta` (Input)
: The noncentrality parameter. `delta` must be nonnegative, and `delta + df` must be less than or equal to 200,000.

Return Value

The probability that a noncentral chi-squared random variable takes a value less than or equal to `imsls_f_non_central_chi_sq_inv` is *p*.

Description

Function `imsls_f_non_central_chi_sq_inv` evaluates the inverse distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, v = `df`, and λ = `delta`, it determines c_0 (= `imsls_f_non_central_chi_sq_inv (p, df, delta)`), such that

$$P = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^{c_0} \frac{x^{(v+2i)/2-1} e^{-x/2}}{2^{(v+2i)/2}\,\Gamma\left(\frac{v+2i}{2}\right)} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to c_0 is P.

Function `imsls_f_non_central_chi_sq_inv` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using routine `imsls_f_non_central_chi_sq` (page 738). See `imsls_f_non_central_chi_sq` for an alternative definition of the noncentral chi-squared random variable in terms of normal random variables.

Example

In this example, we find the 95-th percentage point for a noncentral chi-squared random variable with 2 degrees of freedom and noncentrality parameter 1.

```
#include <imsls.h>

#include <stdio.h>

void main()

{

        float p = .95;

        int df = 2;

        float delta = 1.0;

        float chi_squared;

        chi_squared = imsls_f_non_central_chi_sq_inv(p, df, delta);

       printf("The 0.05 noncentral chi-squared critical value is %6.4f.\n",

               chi_squared);

}
```

Output

```
The 0.05 noncentral chi-squared critical value is  8.6422.
```

F_cdf

Evaluates the *F* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_F_cdf` (*float* `f`, *float* `df_numerator`,
    float `df_denominator`)

The type *double* function is `imsls_d_F_cdf`.

Required Arguments

float `f` (Input)
Point at which the *F* distribution function is to be evaluated.

float `df_numerator` (Input)
The numerator degrees of freedom. Argument `df_numerator` must be positive.

float `df_denominator` (Input)
The denominator degrees of freedom. Argument `df_denominator` must be positive.

Return Value

The probability that an *F* random variable takes a value less than or equal to the input point, `f`.

Description

Function `imsls_f_F_cdf` evaluates the distribution function of a Snedecor's *F* random variable with `df_numerator` and `df_denominator`. The function is evaluated by making a transformation to a beta random variable, then evaluating the incomplete beta function. If X is an F variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1 X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. Function `imsls_f_F_cdf` also uses a relationship between *F* random variables that can be expressed as

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

where F_F is the distribution function for an *F* random variable.

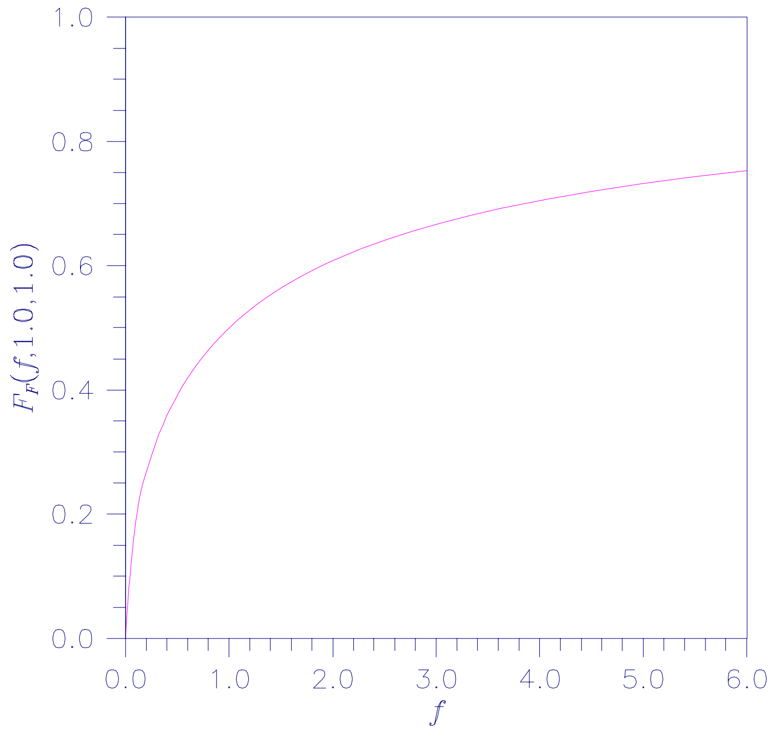


Figure 11-4   Plot of $F_F(f, 1.0, 1.0)$

Example

This example finds the probability that an *F* random variable with one numerator and one denominator degree of freedom is greater than 648.

```
#include <imsls.h>

main()
{
    float       p;
    float       F = 648.0;
    float       df_numerator = 1.0;
    float       df_denominator = 1.0;

    p = 1.0 - imsls_f_F_cdf(F,df_numerator, df_denominator);
    printf("%s %s %6.4f.\n", "The probability that an F(1,1) variate",
        "is greater than 648 is", p);
}
```

Output

```
The probability that an F(1,1) variate is greater than 648 is 0.0250.
```

F_inverse_cdf

Evaluates the inverse of the *F* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_F_inverse_cdf` (*float* `p`, *float* `df_numerator`,
    float `df_denominator`)

The type *double* function is `imsls_d_F_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the *F* distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df_numerator` (Input)
Numerator degrees of freedom. Argument `df_numerator` must be positive.

float `df_denominator` (Input)
Denominator degrees of freedom. Argument `df_denominator` must be positive.

Return Value

The value of the inverse of the *F* distribution function evaluated at `p`. The probability that an *F* random variable takes a value less than or equal to `imsls_f_F_inverse_cdf` is `p`.

Description

Function `imsls_f_F_inverse_cdf` evaluates the inverse distribution function of a Snedecor's *F* random variable with ν_1 = `df_numerator` numerator degrees of freedom and ν_2 = `df_denominator` denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable, then evaluating the inverse of an incomplete beta function. If X is an *F* variate with ν_1 and ν_2 degrees of freedom and $Y = (\nu_1 X)/(\nu_2 + \nu_1 X)$, then Y is a beta variate with parameters $p = \nu_1/2$ and $q = \nu_2/2$. If $p \le 0.5$, `imsls_f_F_ inverse_cdf` uses this relationship directly; otherwise, it also uses a relationship between *F* random variables that can be expressed as follows:

$$F_F(f, \nu_1, \nu_2) = 1 - F_F(1/f, \nu_2, \nu_1)$$

Example

This example finds the 99-th percentage point for an *F* random variable with 7 and 1 degrees of freedom.

```
#include <imsls.h>

main()
{
    float        df_denominator = 1.0;
    float        df_numerator = 7.0;
    float        f;
    float        p = 0.99;

    f = imsls_f_F_inverse_cdf(p, df_numerator, df_denominator);

    printf("The F(7,1) 0.01 critical value is %6.3f\n", f);
}
```

Output

```
The F(7,1) 0.01 critical value is 5928.370
```

Fatal Errors

`IMSLS_F_INVERSE_OVERFLOW`	Function `imsls_f_F_inverse_cdf` overflows. This is because `df_numerator` or `df_denominator` and *p* are too large. The return value is set to machine infinity.

gamma_cdf

Evaluates the gamma distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_cdf` (*float* `x`, *float* `a`)

The type *double* function is `imsls_d_gamma_cdf`.

Required Arguments

float `x` (Input)
Argument for which the gamma distribution function is to be evaluated.

float `a` (Input)
Shape parameter of the gamma distribution. This parameter must be positive.

Return Value

The probability that a gamma random variable takes a value less than or equal to `x`.

Description

Function `imsls_f_gamma_cdf` evaluates the distribution function, *F*, of a gamma random variable with shape parameter *a*,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

where Γ(·) is the gamma function. (The gamma function is the integral from 0 to ∞ of the same integrand as above.) The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The gamma distribution is often defined as a two-parameter distribution with a scale parameter *b* (which must be positive) or as a three-parameter distribution in which the third parameter *c* is a location parameter. In the most general case, the probability density function over (*c*, ∞) is as follows:

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}$$

If *T* is a random variable with parameters *a*, *b*, and *c*, the probability that $T \le t_0$ can be obtained from `imsls_f_gamma_cdf` by setting $x = (t_0 - c)/b$.

If *x* is less than *a* or less than or equal to 1.0, `imsls_f_gamma_cdf` uses a series expansion; otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun 1964.)

Example

Let *X* be a gamma random variable with a shape parameter of four. (In this case, it has an *Erlang distribution* since the shape parameter is an integer.) This example finds the probability that *X* is less than 0.5 and the probability that *X* is between 0.5 and 1.0.

```
#include <imsls.h>

main()
{
    float       p, x;
    float       a = 4.0;

    x = 0.5;
    p = imsls_f_gamma_cdf(x,a);
    printf("The probability that X is less than 0.5 is %6.4f\n", p);

    x = 1.0;
    p = imsls_f_gamma_cdf(x,a) - p;
    printf("The probability that X is between 0.5 and 1.0 is %6.4f\n",
        p);
}
```

Output

```
The probability that X is less than 0.5 is 0.0018
The probability that X is between 0.5 and 1.0 is 0.0172
```

Informational Errors

`IMSLS_ARG_LESS_THAN_ZERO`	Since “x” = # is less than zero, the distribution function is zero at “x.”

Fatal Errors

`IMSLS_X_AND_A_TOO_LARGE`	Since “x” = # and “a” = # are so large, the algorithm would overflow.

gamma_inverse_cdf

Evaluates the inverse of the gamma distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_inverse_cdf` (*float* `p`, *float* `a`)

The type *double* function is `imsls_d_gamma_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the gamma distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

float `a` (Input)
The shape parameter of the gamma distribution. This parameter must be positive.

Return Value

The probability that a gamma random variable takes a value less than or equal to the returned value is `p`.

Description

Function `imsls_f_gamma_inverse_cdf` evaluates the inverse distribution function of a gamma random variable with shape parameter *a*, that is, it determines *x* (=`imsls_f_gamma_inverse_cdf` (`p`, `a`)), such that

$$P = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to x is P. See the documentation for function `imsls_f_gamma_cdf` (page 745) for further discussion of the gamma distribution.

Function `imsls_f_gamma_inverse_cdf` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using function `imsls_f_gamma_cdf`.

Example

In this example, we find the 95-th percentage point for a gamma random variable with shape parameter of 4.

```
include "imsls.h"

void main()

{

  float p = .95, a = 4.0, x;

  x = imsls_f_gamma_inverse_cdf(p,a);

  printf("The 0.05 gamma(4) critical value is %6.4f\n", x);

}
```

Output

```
The 0.05 gamma(4) critical value is 7.7537
```

normal_cdf

Evaluates the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_cdf` (*float* `x`)

The type *double* function is `imsls_d_normal_cdf`.

Required Arguments

float `x` (Input)
Point at which the normal distribution function is to be evaluated.

Return Value

The probability that a normal random variable takes a value less than or equal to x.

Description

Function `imsls_f_normal_cdf` evaluates the distribution function, Φ, of a standard normal (Gaussian) random variable as follows:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point *x* is the probability that the random variable takes a value less than or equal to *x*.

The standard normal distribution (for which `imsls_f_normal_cdf` is the distribution function) has mean of 0 and variance of 1. The probability that a normal random variable with mean μ and variance σ^2 is less than *y* is given by `imsls_f_normal_cdf` evaluated at $(y - \mu)/\sigma$.

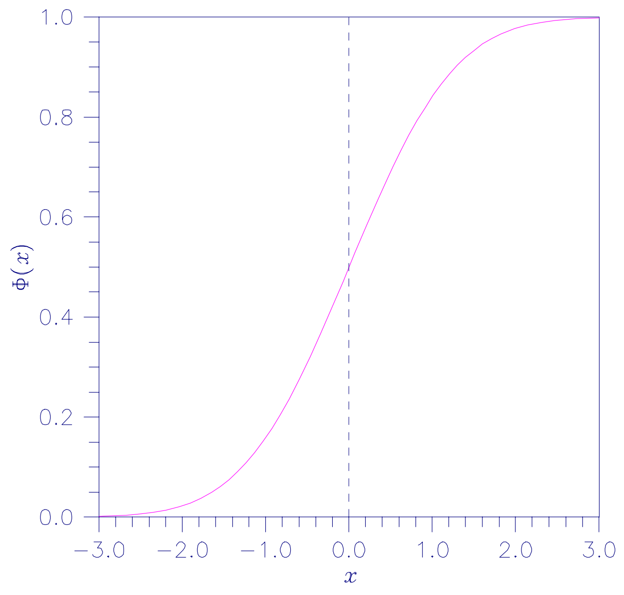


Figure 11-5   Plot of Φ(x)

Example

Suppose *X* is a normal random variable with mean 100 and variance 225. This example finds the probability that *X* is less than 90 and the probability that *X* is between 105 and 110.

```
#include <imsls.h>

main()
{
    float      p, x1, x2;

    x1   = (90.0-100.0)/15.0;
    p    = imsls_f_normal_cdf(x1);
    printf("The probability that X is less than 90 is %6.4f\n", p);

    x1 = (105.0-100.0)/15.0;
    x2 = (110.0-100.0)/15.0;
    p  = imsls_f_normal_cdf(x2) - imsls_f_normal_cdf(x1);
    printf("The probability that X is between 105 and 110 is %6.4f\n",
        p);
}
```

Output

```
The probability that X is less than 90 is 0.2525
The probability that X is between 105 and 110 is 0.1169
```

normal_inverse_cdf

Evaluates the inverse of the standard normal (Gaussian) distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_normal_inverse_cdf` (*float* `p`)

The type *double* function is `imsls_d_normal_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the normal distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

Return Value

The inverse of the normal distribution function evaluated at `p`. The probability that a standard normal random variable takes a value less than or equal to `imsls_f_normal_inverse_cdf` is `p`.

Description

Function `imsls_f_normal_inverse_cdf` evaluates the inverse of the distribution function, Φ, of a standard normal (Gaussian) random variable, `imsls_f_normal_inverse_cdf`(p) = $\Phi^{-1}(x)$, where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x. The standard normal distribution has a mean of 0 and a variance of 1.

Function `imsls_f_normal_inverse_cdf` (*p*) is evaluated by use of minimax rational-function approximations for the inverse of the error function. General descriptions of these approximations are given in Hart et al. (1968) and Strecok (1968). The rational functions used in `imsls_f_normal_inverse_cdf` are described by Kinnucan and Kuki (1968).

Example

This example computes the point such that the probability is 0.9 that a standard normal random variable is less than or equal to this point.

```
#include <imsls.h>

main()
{
    float       x;
    float       p = 0.9;

    x = imsls_f_normal_inverse_cdf(p);
    printf("The 90th percentile of a standard normal is %6.4f.\n", x);
}
```

Output

```
The 90th percentile of a standard normal is 1.2816.
```

t_cdf

Evaluates the Student's *t* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_t_cdf` (*float* `t`, *float* `df`)

The type *double* function is `imsls_d_t_cdf`.

Required Arguments

float `t` (Input)
Argument for which the Student's *t* distribution function is to be evaluated.

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The probability that a Student's *t* random variable takes a value less than or equal to the input *t*.

Description

Function `imsls_f_t_cdf` evaluates the distribution function of a Student's *t* random variable with ν = `df` degrees of freedom. If the square of *t* is greater than or equal to ν, the relationship of a *t* to an *F* random variable (and subsequently, to a beta random variable) is exploited, and percentage points from a beta distribution are used. Otherwise, the method described by Hill (1970) is used. If ν is not an integer, is greater than 19, or is greater than 200, a Cornish-Fisher expansion is used to evaluate the distribution function. If ν is less than 20 and $|t|$ is less than 2.0, a trigonometric series is used (see Abramowitz and Stegun 1964, Equations 26.7.3 and 26.7.4 with some rearrangement). For the remaining cases, a series given by Hill (1970) that converges well for large values of *t* is used.

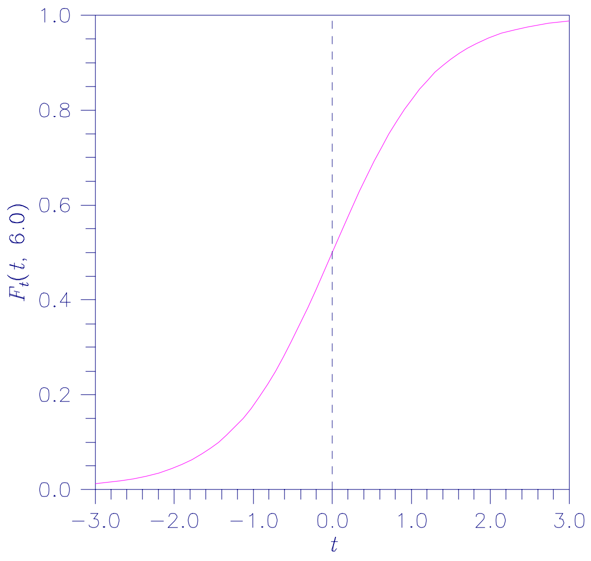


Figure 11-6 Plot of F_t (t, 6.0)

Example

This example finds the probability that a *t* random variable with 6 degrees of freedom is greater in absolute value than 2.447. The fact that *t* is symmetric about 0 is used.

```
#include <imsls.h>

main ()
{
    float       p;
    float       t = 2.447;
    float       df = 6.0;

    p  = 2.0*imsls_f_t_cdf(-t,df);
    printf("Pr(|t(6)| > 2.447) = %6.4f\n", p);
}
```

Output

```
Pr(|t(6)| > 2.447) = 0.0500
```

t_inverse_cdf

Evaluates the inverse of the Student's *t* distribution function.

Synopsis

#include `<imsls.h>`

float `imsls_f_t_inverse_cdf` (*float* `p`, *float* `df`)

The type *double* function is `imsls_d_t_inverse_cdf`.

Required Arguments

float `p` (Input)
Probability for which the inverse of the Student's *t* distribution function is to be evaluated. Argument `p` must be in the open interval (0.0, 1.0).

float `df` (Input)
Degrees of freedom. Argument `df` must be greater than or equal to 1.0.

Return Value

The inverse of the Student's *t* distribution function evaluated at `p`. The probability that a Student's *t* random variable takes a value less than or equal to `imsls_f_t_inverse_cdf` is `p`.

Description

Function `imsls_f_t_inverse_cdf` evaluates the inverse distribution function of a Student's *t* random variable with ν = `df` degrees of freedom. If ν equals 1 or 2, the inverse can be obtained in closed form. If ν is between 1 and 2, the relationship of a *t* to a beta random variable is exploited and the inverse of the beta distribution is used to evaluate the inverse; otherwise, the algorithm of Hill (1970) is used. For small values of ν greater than 2, Hill's algorithm inverts an integrated expansion in $1/(1 + t^2/\nu)$ of the *t* density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

Example

This example finds the 0.05 critical value for a two-sided *t* test with 6 degrees of freedom.

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

```
void main()
{
    float       df = 6.0;
    float       p = 0.975;
    float       t;

    t  = imsls_f_t_inverse_cdf(p,df);

    printf("The two-sided t(6) 0.05 critical value is %6.3f\n", t);
}
```

Output

```
The two-sided t(6) 0.05 critical value is  2.447
```

Informational Errors

`IMSLS_OVERFLOW`	Function `imsls_f_t_inverse_cdf` is set to machine infinity since overflow would occur upon modifying the inverse value for the *F* distribution with the result obtained from the inverse beta distribution.

non_central_t_cdf

Evaluates the noncentral Student's *t* distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_t_cdf` *(float* `t`*, int* `df` *, float* `delta`*)*

The type *double* function is `imsls_d_non_central_t_cdf`.

Required Arguments

float `t` *(Input)*

Argument for which the noncentral Student's *t* distribution function is to be evaluated.

int `df` *(Input)*

Number of degrees of freedom of the noncentral Student's *t* distribution. Argument df must be greater than or equal to 0.0

float `delta` *(Input)*

The noncentrality parameter.

Return Value

The probability that a noncentral Student's *t* random variable takes a value less than or equal to `t`.

Description

Function `imsls_f_non_central_t_cdf` evaluates the distribution function F of a noncentral t random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with v = `df`, δ = `delta` , and t_0 = `t`,

$$F(t_0) = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(v/2)(v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2)(\tfrac{\delta^i}{i!})(\tfrac{2x^2}{v+x^2})^{i/2} dx$$

where Γ(·) is the gamma function. The value of the distribution function at the point t_0 is the probability that the random variable takes a value less than or equal to t_0.

The noncentral t random variable can be defined by the distribution function above, or alternatively and equivalently, as the ratio of a normal random variable and an independent chi-squared random variable. If w has a normal distribution with mean δ and variance equal to one, u has an independent chi-squared distribution with v degrees of freedom, and

$$x = w / \sqrt{u / v}$$

then x has a noncentral t distribution with degrees of freedom and noncentrality parameter δ.

The distribution function of the noncentral t can also be expressed as a double integral involving a normal density function (see, for example, Owen 1962, page 108). The function `TNDF` uses the method of Owen (1962, 1965), which uses repeated integration by parts on that alternate expression for the distribution function.

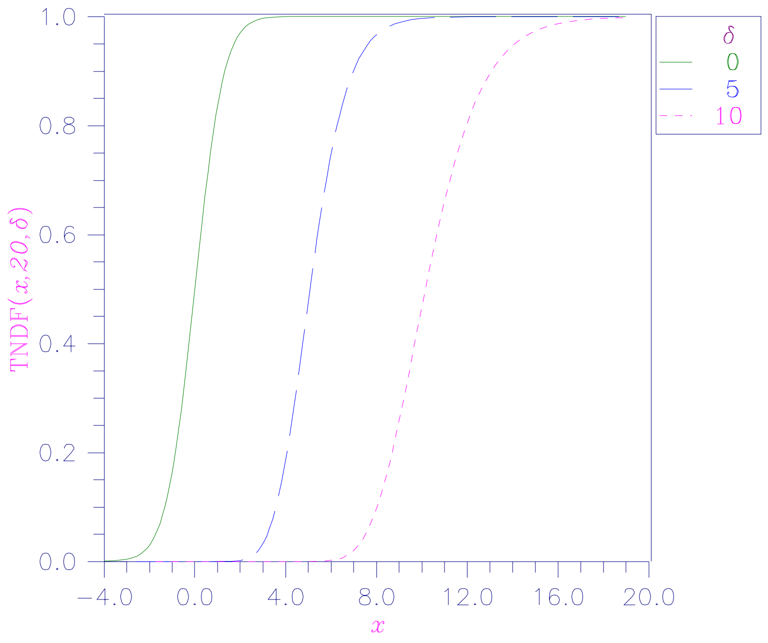


Figure 11-7 Noncentral Student's *t* Distribution Function

Example

Suppose `t` is a noncentral *t* random variable with 6 degrees of freedom and noncentrality parameter 6. In this example, we find the probability that `t` is less than 12.0. (This can be checked using the table on page 111 of Owen 1962, with $\eta = 0.866$, which yields $\lambda = 1.664$.)

```
#include <imsls.h>

#include <stdio.h>

void main()

{

        float t = 12.0;

        int df = 6;

        float delta = 6.0;

        float p;

        p = imsls_f_non_central_t_cdf(t, df, delta);

        printf("The probability that t is less than 12 is %6.4f.\n", p);

}
```

Output

```
The probability that T is less than 12.0 is 0.9501
```

non_central_t_inv_cdf

Evaluates the inverse of the noncentral Student’s *t* distribution function.

Synopsis

#include <imsls.h>

float `imsls_f_non_central_t_inv_cdf` *(float* `p`, *int* `df` , *float* `delta`*)*

The type *double* function is `imsls_d_non_central_t_inv_cdf`.

Required Arguments

float `p` (Input)
A Probability for which the inverse of the noncentral Student's *t* distribution function is to be evaluated. `p` must be in the open interval (0.0, 1.0).

int `df` (Input)
Number of degrees of freedom of the noncentral Student’s *t* distribution. Argument `df` must be greater than or equal to 0.0

float `delta` (Input)
The noncentrality parameter.

Return Value

The probability that a noncentral Student's *t* random variable takes a value less than or equal to `t` is `p`.

Description

Function `imsls_f_non_central_t_inv_cdf` evaluates the inverse distribution function of a noncentral *t* random variable with `df` degrees of freedom and noncentrality parameter `delta`; that is, with P = `p`, v = `df`, and δ = `delta`, it determines t_0 (= `imsls_f_non_central_t_inv_cdf (p, df, delta`)), such that

$$P = \int_{-\infty}^{t_0} \frac{v^{v/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(v/2)(v+x^2)^{(v+1)/2}} \sum_{i=0}^{\infty} \Gamma((v+i+1)/2)(\tfrac{\delta^i}{i!})(\tfrac{2x^2}{v+x^2})^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to t_0 is P. See `imsls_f_non_central_t_cdf` (page 754) for an alternative definition in terms of normal and chi-squared random variables. The function `imsls_f_non_central_t_inv_cdf` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using routine `imsls_f_non_central_t_cdf`.

Example

In this example, we find the 95-th percentage point for a noncentral *t* random variable with 6 degrees of freedom and noncentrality parameter 6.

```
#include <imsls.h>

#include <stdio.h>

void main()

{

        float p = .95;

        int df = 6;

        float delta = 6.0;

        float t;

        t = imsls_f_non_central_t_inv_cdf(p, df, delta);

        printf("The 0.05 noncentral t critical value is %6.4f.\n", t);
}
```

Output

```
The 0.05 noncentral t critical value is 11.995.
```

Chapter 12: Random Number Generation

Routines

Usage Notes

Overview of Random Number Generation

Sections 12.1 through 12.7 describe functions for the generation of random numbers that are useful for applications in Monte Carlo or simulation studies. Before using any of the random number generators, the generator must be initialized by selecting a *seed* or starting value. The user can do this by calling the function `imsls_random_seed_set`. If the user does not select a seed, one is generated using the system clock. A seed needs to be selected only once in a program, unless two or more separate streams of random numbers are maintained. Other utility functions in this chapter can be used to select the form of the basic generator to restart simulations and to maintain separate simulation streams.

In the following discussions, the phrases "random numbers," "random deviates," "deviates," and "variates" are used interchangeably. The phrase "pseudorandom" is sometimes used to emphasize that the numbers generated are really not "random" since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they simulate the realizations of independent and identically distributed random variables.

Basic Uniform Generators

The random number generators in this chapter use either a multiplicative congruential method or a generalized feedback shift register. The selection of the type of generator is made by calling the routine `imsls_random_option` (page 845). If no selection is made explicitly, a multiplicative generator (with multiplier 16807) is used. Whatever distribution is being simulated, uniform (0, 1) numbers are first generated and then transformed if necessary. These routines are *portable* in the sense that, given the same seed and for a given type of generator, they produce the same sequence in all computer/compiler environments. There are many other issues that must be considered in developing programs for the methods described below (see Gentle 1981 and 1990).

The Multiplicative Congruential Generators

The form of the multiplicative congruential generators is

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

Each x_i is then scaled into the unit interval (0,1). If the multiplier, c, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator will have a maximal period of $2^{31} - 2$. There are several other considerations, however. See Knuth (1981) for a good general discussion. The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by the function `imsls_random_option`. The choice of 16807 will result in the fastest execution time, but other evidence suggests that the performance of 950706376 is best among these three choices (Fishman and Moore 1982). If no selection is made explicitly, the functions use the multiplier 16807, which has been in use for some time (Lewis et al. 1969).

The generation of uniform (0,1) numbers is done by the function `imsls_f_random_uniform`. This function is portable in the sense that, given the same seed, it produces the same sequence in all computer/compiler environments.

Shuffled Generators

The user also can select a shuffled version of these generators using `imsls_random_option`. The shuffled generators use a scheme due to Learmonth and Lewis (1973). In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruential generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number; and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table. This scheme is similar to that of Bays and Durham (1976), and their analysis is applicable to this scheme as well.

The Generalized Feedback Shift Register Generator

The GFSR generator uses the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$. This generator, which is different from earlier GFSR generators, was proposed by Fushimi (1990), who discusses the theory behind the generator and reports on several empirical tests of it. Background discussions on this type of generator can be found in Kennedy and Gentle (1980), pages 150–162.

Setting the Seed

The seed of the generator can be set in `imsls_random_seed_set` and can be retrieved by `imsls_random_seed_get`. Prior to invoking any generator in this section, the user can call `imsls_random_seed_set` to initialize the seed, which is an integer variable with a value between 1 and 2147483647. If it is not initialized by `imsls_random_seed_set`, a random seed is obtained from the system clock. Once it is initialized, the seed need not be set again.

If the user wants to restart a simulation, `imsls_random_seed_get` can be used to obtain the final seed value of one run to be used as the starting value in a subsequent run. Also, if two simultaneous random number streams are desired in one run, `imsls_random_seed_set` and `imsls_random_seed_get` can be used before and after the invocations of the generators in each stream.

If a shuffled generator or the GFSR generator is used, in addition to resetting the seed, the user must also reset some values in a table. For the shuffled generators, this is done using the routines `imsls_f_random_table_get` (page 851) and `imsls_f_random_table_set` (page 851); and for the GFSR generator; the table is retrieved and set by the routines `imsls_random_GFSR_table_get` (page 852) and `imsls_random_GFSR_table_set` (page 853). The tables for the shuffled generators are separate for single and double precision; so, if precisions are mixed in a program, it is necessary to manage each precision separately for the shuffled generators.

Timing Considerations

The generation of the uniform (0,1) numbers is done by the routine `imsls_f_random_uniform` (page 804). The particular generator selected in `imsls_random_option` (page 845), that is, the value of the multiplier and whether shuffling is done or whether the GFSR generator is used, affects the speed of `imsls_f_random_uniform`. The smaller multiplier (16807, selected by `iopt` = 1) is faster than the other multipliers. The multiplicative congruential generators that do not shuffle are faster than the ones that do. The GFSR generator is roughly as fast as the fastest multiplicative congruential generator, but the initialization for it (required only on the first invocation) takes longer than the generation of thousands of uniform random numbers. Precise statements of relative speeds depend on the computing system.

Distributions Other than the Uniform

The nonuniform generators use a variety of transformation procedures. All of the transformations used are exact (mathematically). The most straightforward transformation is the *inverse CDF technique*, but it is often less efficient than others involving *acceptance/rejection* and *mixtures*. See Kennedy and Gentle (1980) for discussion of these and other techniques.

Many of the nonuniform generators in this chapter use different algorithms depending on the values of the parameters of the distributions. This is particularly true of the generators for discrete distributions. Schmeiser (1983) gives an overview of techniques for generating deviates from discrete distributions.

Although, as noted above, the uniform generators yield the same sequences on different computers, because of rounding, the nonuniform generators that use acceptance/rejection may occasionally produce different sequences on different computer/compiler environments.

Although the generators for nonuniform distributions use fast algorithms, if a very large number of deviates from a fixed distribution are to be generated, it might be worthwhile to consider a table-sampling method, as implemented in the routines `imsls_f_random_general_discrete` (page 777), `imsls_f_discrete_table_setup` (page 781), `imsls_f_random_general_continuous` (page 810), and `imsls_f_continuous_table_setup` (page 812). 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` (page 804) are reported by Fishman and Moore (1982 and 1986). Results of tests on the generator using the multiplier 16807 with and without shuffling are reported by Learmonth and Lewis (1973b). If the user wishes to perform additional tests, the routines in Chapter 7, “Tests of Goodness of Fit and Randomness,” may be of use. Often in Monte Carlo applications, it is appropriate to construct an ad hoc test that is sensitive to departures that are important in the given application. For example, in using Monte Carlo methods to evaluate a one-dimensional integral, autocorrelations of order one may not be harmful, but they may be disastrous in evaluating a two-dimensional integral. Although generally the routines in this chapter for generating random deviates from nonuniform distributions use exact methods, and, hence, their quality depends almost solely on the quality of the underlying uniform generator, it is often advisable to employ an ad hoc test of goodness of fit for the transformations that are to be applied to the deviates from the nonuniform generator.

Other Notes on Usage

The generators for continuous distributions are available in both single and double-precision versions. This is merely for the convenience of the user; the double-precision versions should not be considered more “accurate,” except possibly for the multivariate distributions.

random_binomial

Generates pseudorandom numbers from a binomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_binomial` (*int* `n_random`, *int* `n`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_binomial`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `n` (Input)
Number of Bernoulli trials.

float `p` (Input)
Probability of success on each trial. Parameter `p` must be greater than 0.0 and less than 1.0.

Return Value

An integer array of length `n_random` containing the random binomial deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_binomial` (*int* `n_random`, *int* `n`, *float* `p`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random binomial deviates.

Description

Function `imsls_f_random_binomial` generates pseudorandom numbers from a binomial distribution with parameters *n* and *p*. Parameters *n* and *p* must be positive, and *p* must less than 1. The probability function (with *n* = `n` and *p* = `p`) is

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

for $x = 0, 1, 2, \ldots, n$.

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 14, “Utilities”), the inverse CDF technique is used; otherwise, the BTPE algorithm of Kachitvichyanukul and Schmeiser (see Kachitvichyanukul 1982) is used. This is an acceptance/rejection method using a composition of four regions. (TPE=Triangle, Parallelogram, Exponential, left and right.)

Example

In this example, `imsls_f_random_binomial` generates five pseudorandom binomial deviates from a binomial distribution with parameters 20 and 0.5.

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

void main()
{
    int   n_random = 5;
    int   n = 20;
    float p = 0.5;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_binomial(n_random, n, p, 0);
    imsls_i_write_matrix("Binomial (20, 0.5) random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Binomial (20, 0.5) random deviates:
       14    9   12   10   12
```

random_geometric

Generates pseudorandom numbers from a geometric distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_geometric` (*int* `n_random`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_geometric`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `p` (Input)
Probability of succes on each trial. Parameter `p` must be positive and less than 1.0.

Return Value

An integer array of length `n_random` containing the random geometric deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_geometric` (*int* `n_random`, *float* `p`,
  `IMSLS_RETURN_USER`, *int* `ir[]`,
  0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
  User-supplied integer array of length `n_random` containing the random geometric deviates.

Description

Function `imsls_f_random_geometric` generates pseudorandom numbers from a geometric distribution with parameter P, where P is the probability of getting a success on any trial. A geometric deviate can be interpreted as the number of trials until the first success (including the trial in which the first success is obtained). The probability function is

$$f(x) = P(1 - P)^{x-1}$$

for $x = 1, 2, \ldots$ and $0 < P < 1$.

The geometric distribution as defined above has mean $1/P$.

The i-th geometric deviate is generated as the smallest integer not less than

$(\log (U_i))/(\log (1 - P))$, where the U_i are independent uniform(0, 1) random numbers (see Knuth 1981).

The geometric distribution is often defined on 0, 1, 2, ..., with mean $(1 - P)/P$. Such deviates can be obtained by subtracting 1 from each element of `ir` (the returned vector of random deviates).

Example

In this example, `imsls_f_random_geometric` generates five pseudorandom geometric deviates from a geometric distribution with parameter an equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float p = 0.3;
    int *ir;
    imsls_random_seed_set(123457);
    ir = imsls_f_random_geometric(n_random, p, 0);
    imsls_i_write_matrix("Geometric(0.3) random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Geometric(0.3) random deviates:
       1   4   1   2   1
```

random_hypergeometric

Generates pseudorandom numbers from a hypergeometric distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_hypergeometric` (*int* `n_random`, *int* `n`, *int* `m`,
int `l`, ..., 0)

The type *double* function is `imsls_d_random_hypergeometric`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `n` (Input)
Number of items in the sample. Parameter `n` must be positive.

int `m` (Input)
Number of special items in the population, or lot. Parameter `m` must be positive.

int `l` (Input)
Number of items in the lot. Parameter `l` must be greater than both `n` and `m`.

Return Value

An integer array of length `n_random` containing the random hypergeometric deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_hypergeometric` (*int* `n_random`, *int* `n`, *int* `m`,
int `l`,

```
IMSLS_RETURN_USER, int ir[],
0)
```

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random hypergeometric deviates.

Description

Function `imsls_f_random_hypergeometric` generates pseudorandom numbers from a hypergeometric distribution with parameters *N*, *M*, and *L*. The hypergeometric random variable *X* can be thought of as the number of items of a given type in a random sample of size *N* that is drawn without replacement from a population of size *L* containing *M* items of this type. The probability function is

$$f(x) = \frac{\binom{M}{x}\binom{L-M}{N-x}}{\binom{L}{N}}$$

for $x = \max(0, N - L + M), 1, 2, \ldots, \min(N, M)$

If the hypergeometric probability function with parameters *N*, *M*, and *L* evaluated at $N - L + M$ (or at 0 if this is negative) is greater than the machine epsilon (see `imsls_f_machine`, Chapter 14, "Utilities"), and less than 1.0 minus the machine epsilon, then `imsls_f_random_hypergeometric` uses the inverse CDF technique. The routine recursively computes the hypergeometric probabilities, starting at $x = \max(0, N - L + M)$ and using the ratio

$$\frac{f(X = x+1)}{f(X = x)}$$

(see Fishman 1978, p. 475).

If the hypergeometric probability function is too small or too close to 1.0, the `imsls_f_random_hypergeometric` generates integer deviates uniformly in the interval $[1, L - i]$ for $i = 0, 1, \ldots$, and at the *i*-th step, if the generated deviate is less than or equal to the number of special items remaining in the lot, the occurence of one special item is tallied and the number of remaining special items is decreased by one. This process continues until the sample size of the number of special items in the lot is reached, whichever comes first. This method can be much slower than the inverse CDF technique. The timing depends on *N*. If *N* is more than half of *L* (which in practical examples is rarely the case), the user may wish to modify the problem, replacing *N* by $L - N$, and to consider the generated deviates to be the number of special items *not* included in the sample.

Example

In this example, `imsls_f_random_hypergeometric` generates five pseudorandom hypergeometric deviates from a hypergeometric distribution to simulate taking random samples of size 4 from a lot containing 20 items, of which 12 are defective. The resulting hypergeometric deviates represent the numbers of defectives in each of the five samples of size 4.

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

void main()
{
    int n_random = 5;
    int n = 4;
    int m = 12;
    int l = 20;
    int *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_hypergeometric(n_random, n, m, l, 0);
    imsls_i_write_matrix("Hypergeometric random deviates: ",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Hypergeometric random deviates:
        4   2   3   3   3
```

Fatal Errors

`IMSLS_LOT_SIZE_TOO_SMALL`	The lot size must be greater than the sample size and the number of defectives in the lot. Lot size = #. Sample size = #. Number of defectives in the lot = #.

random_logarithmic

Generates pseudorandom numbers from a logarithmic distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_logarithmic` (*int* `n_random`, *float* `a`, ..., 0)

The type *double* function is `imsls_d_random_logarithmic`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Parameter of the logarithmic distribution. Parameter `a` must be positive and less than 1.0.

Return Value

An integer array of length `n_random` containing the random logarithmic deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_logarithmic` (*int* `n_random`, *float* `a`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random logarithmic deviates.

Description

Function `imsls_f_random_logarithmic` generates pseudorandom numbers from a logarithmic distribution with parameter `a`. The probability function is

$$f(x) = -\frac{a^x}{x\ln(1-a)}$$

for $x = 1, 2, 3, \ldots$, and $0 < a < 1$

The methods used are described by Kemp (1981) and depend on the value of *a*. If *a* is less than 0.95, Kemp's algorithm LS, which is a "chop-down" variant of an inverse CDF technique, is used. Otherwise, Kemp's algorithm LK, which gives special treatment to the highly probable values of 1 and 2 is used.

Example

In this example, `imsls_f_random_logarithmic` generates five pseudorandom logarithmic deviates from a logarithmic distribution with parameter a equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float a = 0.3;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_logarithmic(n_random, a, 0);
    imsls_i_write_matrix("logarithmic random deviates:",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
logarithmic random deviates:
      2   1   1   1   2
```

random_neg_binomial

Generates pseudorandom numbers from a negative binomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_neg_binomial` (*int* `n_random`, *float* `rk`, *float* `p`, ..., 0)

The type double function is `imsls_d_random_neg_binomial`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `rk` (Input)
Negative binomial parameter. Parameter `rk` must be positive. If `rk` is an integer, the generated deviates can be thought of as the number of failures in a sequence of Bernoulli trials before `rk` successes occur.

float `p` (Input)
Probability of failure on each trial. Parameter `p` must be greater than machine epsilon (see `imsls_f_machine`, Chapter 14, “Utilities”) and less than 1.0.

Return Value

An integer array of length `n_random` containing the random negative binomial deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_neg_binomial` (*int* `n_random`, *float* `rk`, *float* `p`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random negative binomial deviates.

Description

Function `imsls_f_random_neg_binomial` generates pseudorandom numbers from a negative binomial distribution with parameters `rk` and `p`. Parameters `rk` and `p` must be positive and `p` must be less than 1. The probability function (with r = `rk` and p = `p`) is

$$f(x) = \binom{r+x-1}{x}(1-p)^r p^x$$

for $x = 0, 1, 2, ...$

If r is an integer, the distribution is often called the Pascal distribution and can be thought of as modeling the length of a sequence of Bernoulli trials until r successes are obtained, where p is the probability of getting a failure on any trial. In this form, the random variable takes values r, $r + 1$, $r + 2$, … and can be obtained from the negative binomial random variable defined above by adding r to the negative binomial variable. This latter form is also equivalent to the sum of r geometric random variables defined as taking values 1, 2, 3, ...

If $rp/(1 - p)$ is less than 100 and $(1 - p)^r$ is greater than the machine epsilon, `imsls_f_random_neg_binomial` uses the inverse CDF technique; otherwise, for each negative binomial deviate, `imsls_f_random_neg_binomial` generates a gamma $(r, p/(1 - p))$ deviate Y and then generates a Poisson deviate with parameter Y.

Example

In this example, `imsls_f_random_neg_binomial` generates five pseudorandom negative binomial deviates from a negative binomial (Pascal) distribution with parameters r equal to 4 and p equal to 0.3.

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

void main()
{
    int   n_random = 5;
    float rk = 4.0;
    float p = 0.3;
    int   *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_neg_binomial(n_random, rk, p, 0);
    imsls_i_write_matrix(
        "Negative Binomial (4.0, 0.3) random deviates: ",
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Negative Binomial (4.0, 0.3) random deviates:
                 5   1   3   2   3
```

random_poisson

Generates pseudorandom numbers from a Poisson distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_random_poisson` (*int* `n_random`, *float* `theta`, ..., 0)

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `theta` (Input)
Mean of the Poisson distribution. Argument `theta` must be positive.

Return Value

An array of length `n_random` containing the random Poisson deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_poisson` (*int* `n_random`, *float* `theta`,
`IMSLS_RETURN_USER`, *int* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `r[]` (Output)
User-supplied array of length `n_random` containing the random Poisson deviates.

Description

Function `imsls_random_poisson` generates pseudorandom numbers from a Poisson distribution with positive mean `theta`. The probability function (with θ = `theta`) is

$$f(x) = \left(e^{-\theta}\theta^{x}\right)/x! \qquad \text{for } x = 0, 1, 2, ...$$

If `theta` is less than 15, `imsls_random_poisson` uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) (see also Schmeiser 1983) is used. The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

Example

In this example, `imsls_random_poisson` is used to generate five pseudorandom deviates from a Poisson distribution with mean equal to 0.5.

```
#include <imsls.h>

#define N_RANDOM  5

void main()
{
    int         *r;
    int         seed = 123457;
    float       theta = 0.5;

    imsls_random_seed_set (seed);
    r = imsls_random_poisson (N_RANDOM, theta, 0);
    imsls_i_write_matrix ("Poisson(0.5) random deviates", 1, N_RANDOM, r,
0);
}
```

Output

```
Poisson(0.5) random deviates
      1   2   3   4   5
      2   0   1   0   1
```

random_uniform_discrete

Generates pseudorandom numbers from a discrete uniform distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_uniform_discrete` (*int* `n_random`, *int* `k`, ..., 0)

The type *double* function is `imsls_d_random_uniform_discrete`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `k` (Input)
Parameter of the discrete uniform distribution. The integers 1, 2, ..., `k` occur with equal probability. Parameter `k` must be positive.

Return Value

An integer array of length `n_random` containing the random discrete uniform deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_uniform_discrete` (*int* `n_random`, *int* `k`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied integer array of length `n_random` containing the random discrete uniform deviates.

Description

Function `imsls_f_random_uniform_discrete` generates pseudorandom numbers from a uniform discrete distribution over the integers 1, 2, ...`k`. A random integer is generated by multiplying `k` by a uniform (0, 1) random number, adding 1.0, and truncating the result to an integer. This, of course, is equivalent to sampling with replacement from a finite population of size `k`

Example

In this example, `imsls_f_random_uniform_discrete` generates five pseudorandom discrete uniform deviates from a discrete uniform distribution over the integers 1 to 6.

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

void main()
{
    int n_random = 5;
    int k = 6;
    int *ir;

    imsls_random_seed_set(123457);
    ir = imsls_f_random_uniform_discrete(n_random, k, 0);
    imsls_i_write_matrix("Discrete uniform (1, 6) random deviates:" ,
        1, n_random, ir, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
Discrete uniform (1, 6) random deviates:
            6   2   5   4   6
```

random_general_discrete

Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.

Synopsis

#include `<imsls.h>`

int `*imsls_f_random_general_discrete` (*int* `n_random`, *int* `imin`, *int* `nmass`, *float* `probs[]`,..., 0)

The type *double* function is `imsls_d_random_general_discrete`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `imin` (Input)
Smallest value the random deviate can assume.
This is the value corresponding to the probability in `probs`[0].

int `nmass` (Input)
Number of mass points in the discrete distribution.

float `probs[]` (Input)
Array of length `nmass` containing probabilities associated with the individual mass points. The elements of `probs` must be nonnegative and must sum to 1.0.

If the optional argument `IMSLS_TABLE` is used, then `probs` is a vector of length at least `nmass` + 1 containing in the first `nmass` positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities.
IMSL routine `imsls_f_discrete_table_setup` (page 781) can be used to initialize `probs` properly. If no elements of `probs` are used as indexes, `probs` [`nmass`] is 0.0 on input. The value in `probs`[0] is the probability of `imin`. The value in `probs` [`nmass-1`] must be exactly 1.0 (since this is the `CDF` at the upper range of the distribution.)

Return Value

An integer array of length `n_random` containing the random discrete deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_general_discrete` (*int* `n_random`, *int* `imin`, *int* `nmass`, *float* `probs[]`,
    `IMSLS_GET_INDEX_VECTORS`, *int* `**iwk`, *float* `**wk`,
    `IMSLS_GET_INDEX_VECTORS_USER`, *int* `iwk[]`, *float* `wk[]`,
    `IMSLS_SET_INDEX_VECTORS`, *int* `iwk[]`, *float* `wk[]`,
    `IMSLS_RETURN_USER`, *int* `ir[]`,
    `IMSLS_TABLE`,
    0)

Optional Arguments

`IMSLS_GET_INDEX_VECTORS`, *int* `**iwk`, *float* `**wk` (Output)
Retrieve indexing vectors that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` with the same values in `probs`.

`IMSLS_GET_INDEX_VECTORS_USER`, *int* `iwk[]`, *float* `wk[]` (Output)
User-supplied arrays of length `nmass` used for retrieve indexing vectors that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` with the same values in `probs`.

`IMSLS_SET_INDEX_VECTORS`, *int* `*iwk`, *float* `*wk` (Input)
Arrays of length `nmass` that can be used to increase efficiency when multiple calls will be made to `imsls_f_random_general_discrete` the same values in `probs`. These arrays are obtained by using one of the options `IMSLS_GET_INDEX_VECTORS` or `IMSLS_GET_INDEX_VECTORS_USER` in the first call to `imsls_f_random_general_discrete`.

`IMSLS_TABLE` (Input)
Generate pseudorandom numbers from a general discrete distribution using a table lookup method. If this option is used, then `probs` is a vector of length at least `nmass` + 1 containing in the first `nmass` positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities.

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of length `n_random` containing the random discrete deviates.

Description

Routine `imsls_f_random_general_discrete` generates pseudorandom numbers from a discrete distribution with probability function given in the vector `probs`; that is

$$\Pr(X = i) = p_j$$

for $i = i_{\min}, i_{\min} + 1, \ldots, i_{\min} + n_m - 1$ where $j = i - i_{\min} + 1$, p_j = `probs`[*j-1*], $i_{\min}$ = `imin`, and n_m = `nmass`.

The algorithm is the *alias* method, due to Walker (1974), with modifications suggested by Kronmal and Peterson (1979). The method involves a setup phase, in which the vectors `iwk` and `wk` are filled. After the vectors are filled, the generation phase is very fast. To increase efficiency, the first call to `imsls_f_random_general_discrete` can retrieve the arrays `iwk` and `wk` using the optional arguments `IMSLS_GET_INDEX_VECTORS` or `IMSLS_GET_INDEX_VECTORS_USER` , then subsequent calls can be made using the optional argument `IMSLS_SET_INDEX_VECTORS`.

If the optional argument `IMSLS_TABLE` is used, `imsls_f_random_general_discrete` generates pseudorandom deviates from a discrete distribution, using the table `probs`, which contains the cumulative probabilities of the distribution and, possibly, indexes to speed the search of the table. The routine `imsls_f_discrete_table_setup` (page 781) can be used to set up the table `probs`. `imsls_f_random_general_discrete` uses the inverse CDF method to generate the variates.

Example 1

In this example, `imsls_f_random_general_discrete` is used to generate five pseudorandom variates from the discrete distribution:

$$\Pr(X = 1) = .05$$

$$\Pr(X = 2) = .45$$

$$\Pr(X = 3) = .31$$

$$\Pr(X = 4) = .04$$

$$\Pr(X = 5) = .15$$

When `imsls_f_random_general_discrete` is called the first time, `IMSLS_GET_INDEX_VECTORS` is used to initialize the index vectors `iwk` and `wk`. In the next call, `IMSLS_GET_INDEX_VECTORS` is used, so the setup phase is bypassed.

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

void main()
{
  int nr = 5, nmass = 5, iopt = 0, imin = 1, *iwk, *ir;

  float probs[] = {.05, .45, .31, .04, .15};
  float *wk;

  imsls_random_seed_set(123457);


  ir = imsls_f_random_general_discrete(nr, imin, nmass, probs,
                                  IMSLS_GET_INDEX_VECTORS, &iwk, &wk,
                                  0);

  imsls_i_write_matrix("Random deviates", 1, 5, ir,
                    IMSLS_NO_COL_LABELS,
                    0);
  free(ir);

  ir = imsls_f_random_general_discrete(nr, imin, nmass, probs,
                                  IMSLS_SET_INDEX_VECTORS, iwk, wk,
                                  0);

  imsls_i_write_matrix("Random deviates", 1, 5, ir,
                    IMSLS_NO_COL_LABELS,
                    0);

}
```

Output

```
  Random deviates
 3   2   2   3   5

  Random deviates
 1   3   4   5   3
```

Example 2

In this example, `imsls_f_discrete_table_setup` (page 781) is used to set up a table and then `imsls_f_random_general_discrete` is used to generate five pseudorandom variates from the binomial distribution with parameters 20 and 0.5.

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

float prf(int ix);
void main()
{
  int nndx = 12, imin = 0, nmass = 21, nr = 5;
  float del = 0.00001, *cumpr;
  int *ir = NULL;


  cumpr = imsls_f_discrete_table_setup (prf,  del, nndx,  &imin, &nmass, 0);

  imsls_random_seed_set(123457);

  ir = imsls_f_random_general_discrete(nr, imin, nmass, cumpr,
                                  IMSLS_TABLE, 0);

  imsls_i_write_matrix("Binomial (20, 0.5) random deviates", 1, 5, ir,
                    IMSLS_NO_COL_LABELS,
                    0);

}

float prf(int ix)
```

```
{
  int n = 20;
  float  p = .5;
  return imsls_f_binomial_probability (ix, n, p);
}
```

Output

```
Binomial (20, 0.5) random deviates
       14    9   12   10   12
```

discrete_table_setup

Sets up table to generate pseudorandom numbers from a general discrete distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_discrete_table_setup` (*float* `prf()`, *float* `del`,
int `nndx`, *int* `*imin`, *int* `*nmass`, ..., 0)

The type *double* function is `imsls_d_discrete_table_setup`.

Required Arguments

float `prf(`*int* `ix)` (Input)
User-supplied function to compute the probability associated with each mass point of the distribution The argument to the function is the point at which the probability function is to be evaluated. `ix` can range from `imin` to the value at which the cumulative probability is greater than or equal to 1.0 – `del`.

float `del` (Input)
Maximum absolute error allowed in computing the cumulative probability.
Probabilities smaller than `del` are ignored; hence, `del` should be a small positive number. If `del` is too small, however, the return value, `cumpr` [`nmass-1`] must be exactly 1.0 since that value is compared to 1.0 – `del`.

int `nndx` (Input)
The number of elements of `cumpr` available to be used as indexes. `nndx` must be greater than or equal to 1. In general, the larger `nndx` is, to within sixty or seventy percent of `nmass`, the more efficient the generation of random numbers using `imsls_f_random_general_discrete` will be.

int `*imin` (Input/Output)
Pointer to a scalar containing the smallest value the random deviate can

assume. (Input/Output)

imin is not used if optional argument IMSLS_INDEX_ONLY is used. By default, prf is evaluated at imin. If this value is less than del, imin is incremented by 1 and again prf is evaluated at imin. This process is continued until prf(imin) ≥ del. imin is output as this value and the return value cumpr [0] is output as prf(imin).

int *nmass (Input/Output)

Pointer to a scalar containing the number of mass points in the distribution. Input, if IMSLS_INDEX_ONLY is used; otherwise, output. By default, nmass is the smallest integer such that prf(imin + nmass − 1) > 1.0 − del. nmass does include the points $\text{imin}_{\text{in}} + j$ for which $\text{prf}(\text{imin}_{\text{in}} + j) < \text{del}$, for $j = 0, 1, \ldots, \text{imin}_{\text{out}} - \text{imin}_{\text{in}}$, where imin_{in} denotes the input value of imin and imin_{out} denotes its output value.

Return Value

Array, cumpr, of length nmass + nndx containing in the first nmass positions, the cumulative probabilities and in some of the remaining positions, indexes to speed access to the probabilities. To release this space, use free.

Synopsis with Optional Arguments

#include <imsls.h>

void imsls_f_discrete_table_setup (*float* prf(), *float* del, *int* nndx,
    int *imin, *int* *nmass,
    IMSLS_INDEX_ONLY,
    IMSLS_RETURN_USER, *float* cumpr[], *int* lcumpr,
    IMSLS_FCN_W_DATA, *float* prf(), *void* *data,
    0)

Optional Arguments

IMSLS_INDEX_ONLY (Intput)

Fill only the index portion of the result, cumpr, using the values in the first nmass positions. prf is not used and may be a dummy function; also, imin is not used. The optional argument IMSLS_RETURN_USER is required if IMSLS_INDEX_ONLY is used.

IMSLS_RETURN_USER, *float* cumpr[], *int* lcumpr (Input/Output)

cumpr is a user-allocated array of length nmass + nndx containing in the first nmass positions, the cumulative probabilities and in some of the remaining positions, indexes to speed access to the probabilities. lcumpr is the actual length of cumpr as specified in the calling function. Since, by default, the logical length of cumpr is determined in imsls_f_discrete_table_setup, lcumpr is used for error checking. If the option IMSLS_INDEX_ONLY is used, then only the index portion of cumpr are filled.

IMSLS_FCN_W_DATA, *float* prf(*int* ix), *void* *data, (Input)

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

Description

Routine imsls_f_discrete_table_setup sets up a table that routine imsls_f_random_general_discrete (page 777) uses to generate pseudorandom deviates from a discrete distribution. The distribution can be specified either by its probability function prf or by a vector of values of the cumulative probability function. Note that prf is *not* the cumulative probability distribution function. If the cumulative probabilities are already available in cumpr, the only reason to call imsls_f_discrete_table_setup is to form an index vector in the upper portion of cumpr so as to speed up the generation of random deviates by the routine imsls_f_random_general_discrete.

Example 1

In this example, imsls_f_discrete_table_setup is used to set up a table to generate pseudorandom variates from the discrete distribution:

$$\Pr(X = 1) = .05$$

$$\Pr(X = 2) = .45$$

$$\Pr(X = 3) = .31$$

$$\Pr(X = 4) = .04$$

$$\Pr(X = 5) = .15$$

In this simple example, we input the cumulative probabilities directly in cumpr and request 3 indexes to be computed (nndx = 4). Since the number of mass points is so small, the indexes would not have much effect on the speed of the generation of the random variates.

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

float prf(int ix);
void main()
{
  int i, lcumpr = 9, ir[5];
  int nndx = 4, imin = 1, nmass = 5, nr = 5;

  float cumpr[9], del = 0.00001, *p_cumpr = NULL;
  i = 0;
  cumpr[i++] = .05;
  cumpr[i++] = .5;
  cumpr[i++] = .81;
  cumpr[i++] = .85;
  cumpr[i++] = 1.0;

 imsls_f_discrete_table_setup (prf,  del,
                          nndx,  &imin, &nmass,
                          IMSLS_INDEX_ONLY,
                          IMSLS_RETURN_USER, cumpr, lcumpr,
                          0);
 imsls_f_write_matrix("Cumulative probabilities and indexes",
                  1, lcumpr, cumpr, 0);

}

float prf(int ix)
{
  return 0.;

}
```

Output

```
                Cumulative probabilities and indexes
          1            2            3            4            5            6
       0.05         0.50         0.81         0.85         1.00         3.00

          7            8            9
       1.00         2.00         5.00
```

Example 2

This example, `imsls_f_random_general_discrete` is used to set up a table to generate binomial variates with parameters 20 and 0.5. The routine `imsls_f_binomial_probabililty` (Chapter 11, Probability Distribution Functions and Inverses) is used to compute the probabilities.

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

float prf(int ix);
void main()
{
  int lcumpr = 33;
  int nndx = 12, imin = 0, nmass = 21, nr = 5;
  float del = 0.00001, *cumpr;
  int *ir = NULL;


  cumpr = imsls_f_discrete_table_setup (prf,  del, nndx,  &imin, &nmass, 0);

  printf("The smallest point with positive probability using \n");
  printf("the given del is %d and all points after \n", imin);
  printf("point number %d (counting from the input value\n", nmass);
  printf("of IMIN) have zero probability.\n");
  imsls_f_write_matrix("Cumulative probabilities and indexes",
                    nmass+nndx, 1, cumpr,
                        IMSLS_WRITE_FORMAT, "%11.7f", 0);
}

float prf(int ix)
{
  int n = 20;
  float  p = .5;
  return imsls_f_binomial_probability(ix, n, p);
}
```

Output

```
The smallest point with positive probability using
the given del is 1 and all points after
point number 19 (counting from the input value
of IMIN) have zero probability.

Cumulative probabilities and indexes
            1    0.0000191
            2    0.0002003
            3    0.0012875
            4    0.0059080
            5    0.0206938
            6    0.0576583
            7    0.1315873
            8    0.2517219
            9    0.4119013
           10    0.5880987
           11    0.7482781
           12    0.8684127
           13    0.9423417
           14    0.9793062
           15    0.9940920
           16    0.9987125
           17    0.9997997
           18    0.9999809
           19    1.0000000
           20   11.0000000
           21    1.0000000
           22    7.0000000
           23    8.0000000
           24    9.0000000
           25    9.0000000
           26   10.0000000
           27   11.0000000
           28   11.0000000
           29   12.0000000
           30   13.0000000
           31   19.0000000
```

random_beta

Generates pseudorandom numbers from a beta distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_beta` (*int* `n_random`, *float* `pin`, *float* `qin`, ..., 0)

The type *double* function is `imsls_d_random_beta`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `pin` (Input)
First beta distribution parameter. Argument `pin` must be positive.

float `qin` (Input)
Second beta distribution parameter. Argument `qin` must be positive.

Return Value

If no optional arguments are used, `imsls_f_random_beta` returns an array of length `n_random` containing the random standard beta deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_beta` (*int* `n_random`, *float* `pin`, *float* `qin`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
Array of length `n_random` containing the random standard beta deviates.

Description

Function `imsls_f_random_beta` generates pseudorandom numbers from a beta distribution with parameters `pin` and `qin`, both of which must be positive. With p = `pin` and q = `qin`, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1}(1-x)^{q-1} \qquad \text{for } 0 \le x \le 1$$

where $\Gamma(\cdot)$ is the gamma function.

The algorithm used depends on the values of *p* and *q*. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all of the methods use acceptance/rejection. If *p* and *q* are both less than 1, the method of Jöhnk (1964) is used. If either *p* or *q* is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both *p* and *q* are greater than 1, algorithm BB (Cheng 1978), which requires very little setup time, is used if `n_random` is less than 4; and algorithm B4PE of Schmeiser and Babu (1980) is used if `n_random` is greater than or equal to 4. Note that for *p* and *q* both greater than 1, calling `imsls_f_random_beta` in a loop getting less than four variates on each call will not yield the same set of deviates as calling `imsls_f_random_beta` once and getting all the deviates at once because two different algorithms are used.

The values returned in `r` are less than 1.0 and greater than ε, where ε is the smallest positive number such that 1.0 – ε is less than 1.0.

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

Example

In this example, `imsls_f_random_beta` generates five pseudorandom beta (3, 2) variates.

```
#include <imsls.h>

main()
{

    int         n_random = 5;
    int         seed = 123457;
    float       pin = 3.0;
    float       qin = 2.0;
    float       *r;

    imsls_random_seed_set (seed);
    r = imsls_f_random_beta (n_random, pin, qin, 0);
    imsls_f_write_matrix("Beta (3,2) random deviates", 1, n_random,
                         r, 0);
}
```

Output

```
                Beta (3,2) random deviates
         1           2           3           4           5
    0.2814      0.9483      0.3984      0.3103      0.8296
```

random_cauchy

Generates pseudorandom numbers from a Cauchy distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_cauchy` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_cauchy`.

Required Arguments

int `n_random` (Input)
: Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random Cauchy deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_cauchy` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
: User-supplied array of length `n_random` containing the random Cauchy deviates.

Description

Function `imsls_f_random_cauchy` generates pseudorandom numbers from a Cauchy distribution. The probability density function is

$$f(x) = \frac{S}{\pi[S^2 + (x - T)^2]}$$

where T is the median and $T - S$ is the first quartile. This function first generates standard Cauchy random numbers ($T = 0$ and $S = 1$) using the technique described below, and then scales the values using T and S.

Use of the inverse CDF technique would yield a Cauchy deviate from a uniform (0, 1) deviate, u, as tan [π ($u - 0.5$)]. Rather than evaluating a tangent directly, however, `random_cauchy` generates two uniform (−1, 1) deviates, x_1 and x_2. These values can be thought of as sine and cosine values. If

$$x_1^2 + x_2^2$$

is less than or equal to 1, then x_1/x_2 is delivered as the unscaled Cauchy deviate; otherwise, x_1 and x_2 are rejected and two new uniform (−1, 1) deviates are generated. This method is also equivalent to taking the ration of two independent normal deviates.

Example

In this example, `imsls_f_random_cauchy` generates five pseudorandom Cauchy numbers. The generator used is a simple multiplicative congruential with a multiplier of 16807.

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

void main()
{
    int n_random = 5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_cauchy(n_random, 0);
    printf("Cauchy random deviates: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
        r[0], r[1], r[2], r[3], r[4]);

}
```

Output

```
Cauchy random deviates:   3.5765  0.9353 15.5797  2.0815 -0.1333
```

random_chi_squared

Generates pseudorandom numbers from a chi-squared distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_chi_squared` (*int* `n_random`, *float* `df`, ..., `0`)

The type *double* function is `imsls_d_random_chi_squared`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `df` (Input)
Degrees of freedom. Parameter `df` must be positive.

Return Value

An array of length `n_random` containing the random chi-squared deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_chi_squared` (*int* `n_random`, *float* `df`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random chi-squared deviates.

Description

Function `imsls_f_random_chi_squared` generates pseudorandom numbers from a chi-squared distribution with `df` degrees of freedom. If `df` is an even integer less than 17, the chi-squared deviate *r* is generated as

$$r = -2\ln\left(\prod_{i=1}^{n} u_i\right)$$

where n = `df`/2 and the u_i are independent random deviates from a uniform (0, 1) distribution. If `df` is an odd integer less than 17, the chi-squared deviate is generated in the same way, except the square of a normal deviate is added to the expression above. If `df` is is greater than 16 or is not an integer, and if it is not too large to cause overflow in the gamma random number generator, the chi-squared deviate is generated as a special case of a gamma deviate, using function `imsls_f_random_gamma` (page 794). If overflow would occur in `imsls_f_random_gamma`, the chi-squared deviate is generated in the manner described above, using the logarithm of the product of uniforms, but scaling the quantities to prevent underflow and overflow.

Example

In this example, `imsls_f_random_chi_squared` generates five pseudorandom chi-squared deviates with five degrees of freedom.

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

void main()
{
    int   n_random = 5;
    float df = 5.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_chi_squared(n_random, df, 0);
    imsls_f_write_matrix("Chi-Squared random deviates: ",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
              Chi-Squared random deviates:
     12.09        0.48        1.80       14.87        1.75
```

random_exponential

Generates pseudorandom numbers from a standard exponential distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_exponential` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_exponential`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random standard exponential deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_exponential` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random standard exponential deviates.

Description

Function `imsls_f_random_exponential` generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. Function `imsls_f_random_exponential` uses an antithetic inverse CDF technique; that is, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at 1.0 – *U* to yield the exponential deviate.

Deviates from the exponential distribution with mean θ can be generated by using `imsls_f_random_exponential` and then multiplying each entry in `r` by θ.

Example

In this example, `imsls_f_random_exponential` generates five pseudorandom deviates from a standard exponential distribution.

```
#include <imsls.h>

#define N_RANDOM    5

main()

{
        int             seed = 123457;
        int             n_random = N_RANDOM;
        float           *r;

        imsls_random_seed_set(seed);
        r = imsls_f_random_exponential(n_random, 0);
        printf("%s: %8.4f%8.4f%8.4f%8.4f\n",
               "Exponential random deviates",
               r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Exponential random deviates:   0.0344  1.3443  0.2662  0.5633
```

random_exponential_mix

Generates pseudorandom numbers from a mixture of two exponential distributions.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_exponential_mix` (*int* `n_random`, *float* `theta1`, *float* `theta2`, *float* `p`, ..., 0)

The type *double* function is `imsls_d_random_exponential_mix`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `theta1` (Input)
Mean of the exponential distribution which has the larger mean.

float `theta2` (Input)

Mean of the exponential distribution which has the smaller mean. Parameter `theta2` must be positive and less than or equal to `theta1`.

float `p` (Input)

Mixing parameter. Parameter `p` must be non-negative and less than or equal to `theta1`/(`theta1` – `theta2`).

Return Value

An array of length `n_random` containing the random deviates of a mixture of two exponential distributions.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_exponential_mix` (*int* `n_random`, *float* `theta1`,
float `theta2`, *float* `p`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

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

User-supplied array of length `n_random` containing the random deviates.

Description

Function `imsls_f_random_exponential_mix` generates pseudorandom numbers from a mixture of two exponential distributions. The probability density function is

$$f(x) = \frac{p}{\theta_1} e^{-x/\theta_1} + \frac{1-p}{\theta_2} e^{-x/\theta_2}$$

for $x > 0$, where p = `p`, θ_1 = `theta1`, and θ_2 = `theta2`.

In the case of a convex mixture, that is, the case $0 < p < 1$, the mixing parameter p is interpretable as a probability; and `imsls_f_random_exponential_mixed` with probability p generates an exponential deviate with mean θ_1, and with probability $1 - p$ generates an exponential with mean θ_2. When p is greater than 1, but less than $\theta_1/(\theta_1 - \theta_2)$, then either an exponential deviate with mean θ_1 or the sum of two exponentials with means θ_1 and θ_2 is generated. The probabilities are $q = p - (p - 1)(\theta_1/\theta_2)$ and $1 - q$, respectively, for the single exponential and the sum of the two exponentials.

Example

In this example, `imsls_f_random_exponential_mix` is used to generate five pseudorandom deviates from a mixture of exponentials with means 2 and 1, respecctively, and with mixing parameter 0.5.

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

void main()
{
    int   n_random = 5;
    float theta1 = 2.0;
    float theta2 = 1.0;
    float p = 0.5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_exponential_mix(n_random, theta1, theta2, p, 0);
    imsls_f_write_matrix("Mixed exponential random deviates: ",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);

}
```

Output

```
            Mixed exponential random deviates:
     0.070       1.302       0.630       1.976       0.372
```

random_gamma

Generates pseudorandom numbers from a standard gamma distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_gamma` (*int* `n_random`, *float* `a`, ..., 0)

The type *double* function is `imsls_d_random_gamma`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Shape parameter of the gamma distribution. This parameter must be positive.

Return Value

An array of length `n_random` containing the random standard gamma deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_gamma` (*int* `n_random`, *float* `a`,
  `IMSLS_RETURN_USER`, *float* `r[]`,
  0)

Optional Arguments

`IMSLS_USER_RETURN`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random standard gamma deviates.

Description

Function `imsls_f_random_gamma` generates pseudorandom numbers from a gamma distribution with shape parameter a and unit scale parameter. The probability density function is

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \qquad \text{for } x \geq 0$$

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>

void main()
{
    int         seed = 123457;
    int         n_random = 5;
    float       a = 3.0;
    float       *r;

    imsls_random_seed_set(seed);
    r = imsls_f_random_gamma(n_random, a, 0);
    imsls_f_write_matrix("Gamma(3) random deviates", 1, n_random, r, 0);
}
```

Output

```
                  Gamma(3) random deviates
         1            2            3            4            5
     6.843        3.445        1.853        3.999        0.779
```

random_lognormal

Generates pseudorandom numbers from a lognormal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_lognormal` (*int* `n_random`, *float* `mean`,
float `std`, ..., 0)

The type *double* function is `imsls_d_random_lognormal`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `mean` (Input)
Mean of the underlying normal distribution.

float `std` (Input)
Standard deviation of the underlying normal distribution.

Return Value

An array of length `n_random` containing the random deviates of a lognormal distribution. The log of each element of the vector has a normal distribution with mean `mean` and standard deviation `std`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_lognormal` (*int* `n_random`, *float* `mean`,
    float `std`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
  User-supplied array of length `n_random` containing the random lognormal deviates.

Description

Function `imsls_f_random_lognormal` generates pseudorandom numbers from a lognormal distribution with parameters `mean` and `std`. The scale parameter in the underlying normal distribution, `std`, must be positive. The method is to generate normal deviates with mean `mean` and standard deviation `std` and then to exponentiate the normal deviates.

With μ = `mean` and σ = `std`, the probability density function for the lognormal distribution is

$$f(x) = \frac{1}{\sigma x \sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma^2} (\ln x - \mu)^2 \right]$$

for $x > 0$. The mean and variance of the lognormal distribution are exp $(\mu + \sigma^2/2)$ and exp $(2\mu + 2\sigma^2)$ – exp $(2\mu + \sigma^2)$, respectively.

Example

In this example, `imsls_f_random_lognormal` is used to generate five pseudorandom lognormal deviates with a mean of 0 and standard deviation of 1.

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

void main()
{
    int   n_random = 5;
    float mean = 0.0;
    float std = 1.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_lognormal(n_random, mean, std, 0);
    imsls_f_write_matrix("lognormal random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                  lognormal random deviates:
     7.780       2.954       1.086       3.588       0.293
```

random_normal

Generates pseudorandom numbers from a normal, N (μ, σ^2), distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_normal` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_normal`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random normal deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_normal` (*int* `n_random`,
`IMSLS_MEAN`, *float* `mean`,
`IMSLS_VARIANCE`, *float* `variance`,
`IMSLS_ACCEPT_REJECT_METHOD`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_MEAN`, *float* `mean` (Input)
Parameter `mean` contains the mean, μ, of the N(μ, σ^2) from which random normal deviates are to be generated.
Default: `mean` = 0.0

`IMSLS_VARIANCE`, *float* `variance` (Input)
Parameter variance contains the variance of the N (μ, σ^2) from which random normal deviates are to be generated.
Default: `variance` = 1.0

`IMSLS_ACCEPT_REJECT_METHOD`
By default, random numbers are generated using an inverse CDF technique. When optional argument `IMSLS_ACCEPT_REJECT_METHOD` is specified, an acceptance/ rejection method is used instead. See the “Description” section for details about each method.

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the generated random standard normal deviates.

Description

By default, function `imsls_f_random_normal` generates pseudorandom numbers from a normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0, 1) random deviate is generated. The inverse of the normal distribution function is then evaluated at that point, using the function `imsls_f_normal_inverse_cdf` (Chapter 11, Probablility Distribution Functions and Inverses).

If optional argument `IMSLS_ACCEPT_REJECT_METHOD` is specified, function `imsls_f_random_normal` generates pseudorandom numbers using an acceptance/rejection technique due to Kinderman and Ramage (1976). In this method, the normal density is represented as a mixture of densities over which a variety of acceptance/rejection method due to Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia et al. (1964) are applied. This method is faster than the inverse CDF technique.

Remarks

Function `imsls_random_seed_set` can be used to initialize the seed of the random number generator; function `imsls_random_option` can be used to select the form of the generator.

Example

In this example, `imsls_f_random_normal` generates five pseudorandom deviates from a standard normal distribution.

```
#include <imsls.h>
#define N_RANDOM  5

void main()
{
    int         seed = 123457;
    int         n_random = N_RANDOM;
    float       *r;

    imsls_random_seed_set (seed);
    r = imsls_f_random_normal(n_random, 0);
    printf("%s:\n%8.4f%8.4f%8.4f%8.4f%8.4f\n",
           "Standard normal random deviates",
           r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Standard normal random deviates:
  1.8279 -0.6412  0.7266  0.1747  1.0145
  1.8280
```

random_stable

Generates pseudorandom numbers from a stable distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_stable` (*int* `n_random`, *float* `alpha`,
 float `bprime`, ..., 0)

The type *double* function is `imsls_d_random_stable`.

Required Arguments

int `n_random` (Input)
 Number of random numbers to generate.

float `alpha` (Input)
 Characteristic exponent of the stable distribution. This parameter must be positive and less than or equal to 2.

float `bprime` (Input)
 Skewness parameter of the stable distribution. When `bprime` = 0, the distribution is symmetric. Unless `alpha` = 1, `bprime` is not the usual skewness parameter of the stable distribution. `bprime` must be greater than or equal to − 1 and less than or equal to 1.

Return Value

An integer array of length `n_random` containing the random deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_binomial` (*int* `n_random`, *float* `alpha`,
 float `bprime`,
 `IMSLS_RETURN_USER`, *float* `r[]`,
 0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
 User-supplied array of length `n_random` containing the random deviates.

Description

Function `imsls_f_random_stable` generates pseudorandom numbers from a stable distribution with parameters `alpha` and `bprime`. `alpha` is the usual characteristic exponent parameter α and `bprime` is related to the usual skewness parameter β of the stable distribution. With the restrictions $0 < \alpha \le 2$ and $-1 \le \beta \le 1$, the characteristic function of the distribution is

$$\varphi(t) = \exp[-|t|^{\alpha} \exp(-\pi i\beta(1 - |1 - \alpha|)\text{sign}(t)/2)] \quad \text{for } \alpha \ne 1$$

and

$$\varphi(t) = \exp[-|t|(1 + 2i\beta \ln|t|)\text{sign}(t)/\pi)] \quad \text{for } \alpha = 1$$

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` $= \beta'$ and the standard β is

$$\beta' = -\tan(\pi(1 - \alpha)/2) \tan(-\pi\beta(1 - |1 - \alpha|)/2) \quad \text{for } \alpha \ne 1$$

and

$$\beta' = \beta \quad \text{for } \alpha = 1$$

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

void main()
{
  int  nr = 5;
  float alpha = 1.5, bprime = 0.0, *r;

  imsls_random_seed_set(123457);

  r = imsls_f_random_stable(nr, alpha, bprime, 0);
  imsls_f_write_matrix("Stable random deviates", 5, 1, r,
                       IMSLS_NO_ROW_LABELS, 0);

}
```

Output

```
Stable random deviates
           4.409
           1.056
           2.546
           5.672
           2.166
```

random_student_t

Generates pseudorandom numbers from a Student's *t* distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_student_t` (*int* `n_random`, *float* `df`, ..., 0)

The type *double* function is `imsls_d_random_student_t`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `df` (Input)
Degrees of freedom. Parameter `df` must be positive.

Return Value

An array of length `n_random` containing the random deviates of a Student's *t* distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_student_t` (*int* `n_random`, *float* `df`,
`IMSLS_RETURN_USER`, *float* `r[]`,
`IMSLS_MEAN`, *float* `mean`,
`IMSLS_VARIANCE`, *float* `variance`,
0)

Optional Arguments

`IMSLS_MEAN`, *float* `mean` (Input)
Mean of the Student's *t* distribution.
Default: `mean` = 0.0

`IMSLS_VARIANCE`, *float* `variance` (Input)
Variance of the Student's *t* distribution.
Default: `variance` = 1.0

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random Student's *t* deviates.

Description

Function `imsls_f_random_student_t` generates pseudorandom numbers from a Student's *t* distribution with `df` degrees of freedom, using a method suggested by Kinderman et al. (1977). The method ("TMX" in the reference) involves a representation of the *t* density as the sum of a triangular density over (−2, 2) and the difference of this and the *t* density. The mixing probabilities depend on the degrees of freedom of the *t* distribution. If the triangular density is chosen, the variate is generated as the sum of two uniforms; otherwise, an acceptance/rejection method is used to generate the difference density.

random_triangular

Generates pseudorandom numbers from a triangular distribution on the interval (0, 1).

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_triangular` (*int* `n_random`, ..., 0)

The type *double* function is `imsls_d_random_triangular`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random deviates of a triangular distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_triangular` (*int* `n_random`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random triangular deviates.

Description

Function `imsls_f_random_triangular` generates pseudorandom numbers from a triangular distribution over the unit interval. The probability density function is $f(x) = 4x$, for $0 \le x \le 0.5$, and $f(x) = 4(1 - x)$, for $0.5 < x \le 1$. An inverse CDF technique is used.

Example

In this example, `imsls_f_random_triangular` is used to generate five pseudorandom deviates from a triangular distribution.

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

void main()
{
    int   n_random = 5;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_triangular(n_random, 0);
    imsls_f_write_matrix("Triangular random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                Triangular random deviates:
    0.8700      0.3610      0.6581      0.5360      0.7215
```

random_uniform

Generates pseudorandom numbers from a uniform (0, 1) distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_uniform` (*int* `n_random`, …, 0)

The type *double* function is `imsls_d_random_uniform`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

Return Value

An array of length `n_random` containing the random uniform (0, 1) deviates.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_uniform` (*int* `n_random`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
    User-supplied array of length `n_random` containing the random uniform (0, 1) deviates.

Description

Function `imsls_f_random_uniform` generates pseudorandom numbers from a uniform (0, 1) distribution using a multiplicative congruential method. The form of the generator is as follows:

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

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

void main()
{
    float     *r;

    imsls_random_seed_set(123457);

    r = imsls_f_random_uniform(N_RANDOM, 0);

    printf("Uniform random deviates: %8.4f%8.4f%8.4f%8.4f%8.4f\n",
            r[0], r[1], r[2], r[3], r[4]);
}
```

Output

```
Uniform random deviates:   0.9662  0.2607  0.7663  0.5693  0.8448
```

random_von_mises

Generates pseudorandom numbers from a von mises distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_von_mises` (*int* `n_random`, *float* `c`, …, 0)

The type *double* function is `imsls_d_random_von_mises`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `c` (Input)
Parameter of the von Mises distribution. This parameter must be greater than one-half of machine epsilon (On many machines, the lower bound for `c` is 10^{-3}).

Return Value

An array of length `n_random` containing the random deviates of a von Mises distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_von_mises` (*int* `n_random`, *float* `c`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
    User-supplied array of length `n_random` containing the random von mises deviates.

Description

Function `imsls_f_random_von_mises` generates pseudorandom numbers from a von Mises distribution with parameter `c`, which must be positive. With c = `c`, the probability density function is

$$f(x) = \frac{1}{2\pi I_0(c)} \exp\left[c \cos(x)\right]$$

for $-\pi < x < \pi$, where I_0 (c) is the modified Bessel function of the first kind of order 0. The probability density is equal to 0 outside the interval $(-\pi, \pi)$.

The algorithm is an acceptance/rejection method using a wrapped Cauchy distribution as the majorizing distribution. It is due to Nest and Fisher (1979).

Example

In this example, `imsls_f_random_von_mises` is used to generate five pseudorandom von Mises variates with $c = 1$.

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

void main()
{
    int   n_random = 5;
    float c = 1.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_von_mises(n_random, c, 0);
    imsls_f_write_matrix("Von Mises random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
          Von Mises random deviates:
0.247     -2.433     -1.022     -2.172     -0.503
```

random_weibull

Generates pseudorandom numbers from a Weibull distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_weibull` (*int* `n_random`, *float* `a`, …, 0)

The type *double* function is `imsls_d_random_weibull`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

float `a` (Input)
Shape parameter of the Weibull distribution. This parameter must be positive.

Return Value

An array of length `n_random` containing the random deviates of a Weibull distribution.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_weibull` (*int* `n_random`, *float* `a`,
`IMSLS_B`, *float* `b`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_B`, *float* `b` (Input)
Scale parameter of the two parameter Weibull distribution.
Default: `b` = 1.0

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random Weibull deviates.

Description

Function `imsls_f_random_weibull` generates pseudorandom numbers from a Weibull distribution with shape parameter *a* and scale parameter *b*. The probability density function is

$$f\left(x\right)=abx^{a-1}\exp\left(-bx^{a}\right)$$

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

$$r\left(x\right)=\frac{1}{\alpha^{2}}xe^{-\left(x^{2}/\left(2\alpha^{2}\right)\right)}$$

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 two-parameter Weibull distribution with shape parameter equal to 2.0 and scale parameter equal to 6.0—a Rayleigh distribution with the following parameter:

$$\alpha=3\sqrt{2}$$

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

void main()
{
    int   n_random = 5;
    float a = 3.0;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_weibull(n_random, a, 0);
    imsls_f_write_matrix("Weibull random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                 Weibull random deviates:
     0.325       1.104       0.643       0.826       0.552
```

Warning Errors

`IMSLS_SMALL_A`	The shape parameter is so small that a relatively large proportion of the values of deviates from the Weibull cannot be represented.

random_general_continuous

Generates pseudorandom numbers from a general continuous distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_general_continuous` (*int* `n_random`, *int* `ndata`,
    float `table[]`,..., 0)

The type *double* function is `imsls_d_random_general_continuous`.

Required Arguments

int `n_random` (Input)
    Number of random numbers to generate.

int `ndata` (Input)
    Number of points at which the CDF is evaluated for interpolation.
    `ndata` must be greater than or equal to 4.

float `*table` (Input/Ouput)
    `ndata` by 5 table to be used for interpolation of the cumulative distribution function.
    The first column of `table` contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing beginning with 0.0 and ending at 1.0) and the remaining columns contain values used in interpolation. This table is set up using routine `imsls_f_continous_table_setup` (page 812).

Return Value

An array of length `n_random` containing the random discrete deviates. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_f_random_general_continuous` (*int* `n_random`, *int* `ndata`,
    float `table[]`,
    `IMSLS_TABLE_COL_DIM`, *int* `table_col_dim`,

`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_TABLE_COL_DIM`, *int* `table_col_dim` (Intput)
Column dimension of the matrix `table`.
Default: `table_col_dim` = 5

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` containing the random continuous deviates.

Description

Routine `imsls_f_random_general_continuous` generates pseudorandom numbers from a continuous distribution using the inverse CDF technique, by interpolation of points of the distribution function given in `table`, which is set up by routine `imsls_f_continuous_table_setup` (page 812). A strictly monotone increasing distribution function is assumed. The interpolation is by an algorithm attributable to Akima (1970), using piecewise cubics. The use of this technique for generation of random numbers is due to Guerra, Tapia, and Thompson (1976), who give a description of the algorithm and accuracy comparisons between this method and linear interpolation. The relative errors using the Akima interpolation are generally considered very good.

Example 1

In this example, `imsls_f_continuous_table_setup` (page 812) is used to set up a table for generation of beta pseudorandom deviates. The CDF for this distribution is computed by the routine `imsls_f_beta_cdf` (Chapter 11, Probability Distribution Functions and Inverses). The table contains 100 points at which the CDF is evaluated and that are used for interpolation.

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

float cdf(float);
void main()
{
  int i, iopt=0, ndata= 100;
  float table[100][5], x = 0.0, *r;

  for (i=0;i<ndata;i++) {
    table[i][0] = x;
    x += .01;
  }

  imsls_f_continuous_table_setup(cdf, iopt, ndata, (float*)table);

  imsls_random_seed_set(123457);
  r = imsls_f_random_general_continuous (5,  ndata, table, 0);
  imsls_f_write_matrix("Beta (3, 2) random deviates", 5, 1, r, 0);
}

float cdf(float x)
{
  return imsls_f_beta_cdf(x, 3., 2.);
}
```

Output

```
*** WARNING  Error  from imsls_f_continuous_table_setup.  The values of the
***          CDF in the second column of table did not begin at 0.0 and end
***          at 1.0, but they have been adjusted. Prior to adjustment,
***          table[0][1] = 0.000000e+00 and table[ndata-1][1]= 9.994079e-01.

Beta (3, 2) random deviates
       1      0.9208
       2      0.4641
       3      0.7668
       4      0.6536
       5      0.8171
```

continuous_table_setup

Sets up table to generate pseudorandom numbers from a general continuous distribution.

Synopsis

#include `<imsls.h>`

void `imsls_f_continuous_table_setup` (*float* `cdf()`, *int* `iopt`, *int* `ndata`, *float* `*table`, ..., 0)

The type *double* function is `imsls_d_continuous_table_setup`.

Required Arguments

float `cdf(`*float* `x)` (Input)
User-supplied function to compute the cumulative distribution function. The argument to the function is the point at which the distribution function is to be evaluated

int `iopt` (Input)
Indicator of the extent to which `table` is initialized prior to calling `imsls_f_continuous_table_setup`.

iopt	**Action**
0	`imsls_f_continuous_table_setup` fills the last four columns of `table`. The user inputs the points at which the CDF is to be evaluated in the first column of `table`. These must be in ascending order.

1	`imsls_f_continuous_table_setup` fills the last three columns of `table`. The user supplied function `cdf` is not used and may be a dummy function; instead, the cumulative distribution function is specified in the first two columns of `table`. The abscissas (in the first column) must be in ascending order and the function must be strictly monotonically increasing.

int `ndata` (Input)
Number of points at which the CDF is evaluated for interpolation. `ndata` must be greater than or equal to 4.

float `*table` (Input/Ouput)
`ndata` by 5 table to be used for interpolation of the cumulative distribution function.
The first column of `table` contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing), and the remaining columns contain values used in interpolation. The first row of `table` corresponds to the left limit of the support of the distribution and the last row corresponds to the right limit of the support; that is, `table`[0][1] = 0.0 and `table`[`ndata`-1][1] = 1.0.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_continuous_table_setup` (*float* `cdf()`, *int* `iopt`,
int `ndata`, *float* `table[]`,
`IMSLS_TABLE_COL_DIM`,
`IMSLS_FCN_W_DATA`, *float* `cdf()`, *void* `*data`,
0)

Optional Arguments

`IMSLS_TABLE_COL_DIM`, *int* `table_col_dim` (Intput)
Column dimension of the array `table`.
Default: `table_col_dim` = 5

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

Description

Routine `imsls_f_continuous_table_setup` sets up a table that routine `imsls_f_random_general_continuous` (page 810) can use to generate pseudorandom deviates from a continuous distribution. The distribution is specified by its cumulative distribution function, which can be supplied either in tabular form in `table` or by a function `cdf`. See the documentation for the routine `imsls_f_random_general_continuous` for a description of the method.

Example 1

In this example, `imsls_f_continuous_table_setup` is used to set up a table to generate pseudorandom variates from a beta distribution. This example is continued in the documentation for routine `imsls_f_random_general_continuous` (page 810) to generate the random variates.

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

float cdf(float);
void main()
{
  int i, iopt=0, ndata= 100;
  float table[100][5], x = 0.0;

  for (i=0;i<ndata;i++) {
    table[i][0] = x;
    x += .01;
  }

  imsls_f_continuous_table_setup(cdf, iopt, ndata, table);
  printf("The first few values from the table:\n");
  for (i=0;i<10;i++) printf("%4.2f\t%8.4f\n", table[i][0], table[i][1]);

}

float cdf(float x)
{
  return imsls_f_beta_cdf(x, 3., 2.);
}
```

Output

```
*** WARNING  Error  from imsls_f_continuous_table_setup.  The values of the
***          CDF in the second column of table did not begin at 0.0 and end
***          at 1.0, but they have been adjusted. Prior to adjustment,
***          table[0][1] = 0.000000e+00 and table[ndata-1][1]= 9.994079e-01.

The first few values from the table:
0.00     0.0000
0.01     0.0000
0.02     0.0000
0.03     0.0001
0.04     0.0002
0.05     0.0005
0.06     0.0008
```

```
0.07     0.0013
0.08     0.0019
0.09     0.0027
```

random_normal_multivariate

Generates pseudorandom numbers from a multivariate normal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_normal_multivariate` (*int* `n_vectors`,
    int `length`, *float* `*covariances`, …, 0)

The type *double* function is `imsls_d_random_normal_multivariate`.

Required Arguments

int `n_vectors` (Input)
  Number of random multivariate normal vectors to generate.

int `length` (Input)
  Length of the multivariate normal vectors.

float `*covariances` (Input)
  Array of size `length` × `length` containing the variance-covariance matrix.

Return Value

An array of length `n_vectors` × `length` containing the random multivariate normal vectors stored consecutively.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_normal_multivariate` (*int* `n_vectors`,
    int `length`, *float* `*covariances`,
    `IMSLS_RETURN_USER`, *float* `r[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
  User-supplied array of length `n_vectors` × `length` containing the random multivariate normal vectors stored consecutively.

Description

Function `imsls_f_random_normal_multivariate` generates pseudorandom numbers from a multivariate normal distribution with mean vector consisting of all zeros and variance-covariance matrix `imsls_f_covariances`. First, the Cholesky factor of the variance-covariance matrix is computed. Then, independent random normal deviates with mean 0 and variance 1 are generated, and the matrix containing these deviates is postmultiplied by the Cholesky factor. Because the Cholesky factorization is performed in each invocation, it is best to generate as many random vectors as needed at once.

Deviates from a multivariate normal distribution with means other than 0 can be generated by using `imsls_f_random_normal_multivariate` and then by adding the vectors of means to each row of the result.

Example

In this example, `imsls_f_random_normal_multivariate` generates five pseudorandom normal vectors of length 2 with variance-covariance matrix equal to the following:

$$\begin{bmatrix} 0.500 & 0.375 \\ 0.375 & 0.500 \end{bmatrix}$$

```
#include <imsls.h>

void main()
{
    int n_vectors = 5;
    int length = 2;
    float covariances[] = {.5, .375, .375, .5};
    float *random;

    imsls_random_seed_set (123457);
    random = imsls_f_random_normal_multivariate (n_vectors, length,
        covariances, 0);

    imsls_f_write_matrix ("multivariate normal random deviates",
        n_vectors, length, random, 0);
}
```

Output

```
multivariate normal random deviates
                 1            2
     1       1.451        1.246
     2       0.766       -0.043
     3       0.058       -0.669
     4       0.903        0.463
     5      -0.867       -0.933
```

random_orthogonal_matrix

Generates a pseudorandom orthogonal matrix or a correlation matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_orthogonal_matrix` (*int* `n`, ..., 0)

The type *double* function is `imsls_d_random_orthogonal_matrix`.

Required Arguments

int `n` (Input)
The order of the matrix to be generated.

Return Value

`n` by `n` random orthogonal matrix. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_orthogonal_matrix` (*int* `n`,
`IMSLS_EIGENVALUES`, *float* `*eignevalues[]`,
`IMSLS_A_MATRIX`, *float* `*a`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_EIGENVALUES`, *float* `*eigenvalues` (Input)
A vector of length `n` containing the eigenvalues of the correlation matrix to be generated. The elements of `eigenvalues` must be positive, they must sum to `n`, and they cannot all be equal.

`IMSLS_A_MATRIX`, *float* `*a` (Input)
`n` by `n` random orthogonal matrix. A random correlation matrix is generated using the orthogonal matrix input in `a`. The option `IMSLS_EIGENVALUES` must also be supplied if `IMSLS_A_MATRIX` is used.

`IMSLS_A_COL_DIM`, *int* `a_col_dim` (Input)
Column dimension of the matrix `a`.
Default: `a_col_dim` = `n`

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n` × `n` containing the random correlation matrix.

Description

Routine `imsls_f_random_orthogonal_matrix` generates a pseudorandom orthogonal matrix from the invariant Haar measure. For each column, a random vector from a uniform distribution on a hypersphere is selected and then is projected onto the orthogonal complement of the columns already formed. The method is described by Heiberger (1978). (See also Tanner and Thisted 1982.)

If the optional argument `IMSLS_EIGENVALUES` is used, a correlation matrix is formed by applying a sequence of planar rotations to the matrix $A^T DA$, where D = diag(`eigenvalues`[0], …, `eigenvalues` [`n-1`]), so as to yield ones along the diagonal. The planar rotations are applied in such an order that in the two by two matrix that determines the rotation, one diagonal element is less than 1.0 and one is greater than 1.0. This method is discussed by Bendel and Mickey (1978) and by Lin and Bendel (1985).

The distribution of the correlation matrices produced by this method is not known. Bendel and Mickey (1978) and Johnson and Welch (1980) discuss the distribution.

For larger matrices, rounding can become severe; and the double precision results may differ significantly from single precision results.

Example

In this example, `imsls_f_random_orthogonal_matrix` is used to generate a 4 by 4 pseudorandom correlation matrix with eigenvalues in the ratio 1:2:3:4.

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

void main()
{
  int   i, n = 4;
  float *a, *cor;
  float ev[] = {1., 2., 3., 4.};

  for (i=0;i<4;i++) ev[i] = 4.*ev[i]/10.;

  imsls_random_seed_set(123457);

  a = imsls_f_random_orthogonal_matrix(n, 0);
  imsls_f_write_matrix("Random orthogonal matrix",
                    4, 4, (float*)a, 0);

  cor = imsls_f_random_orthogonal_matrix(n,
                              IMSLS_EIGENVALUES, ev,
                              IMSLS_A_MATRIX, a,
                              0);
  imsls_f_write_matrix("Random correlation matrix",
                    4, 4, (float*)cor, 0);

}
```

Output

```
             Random orthogonal matrix
             1           2           3           4
1      -0.8804     -0.2417      0.4065     -0.0351
2       0.3088     -0.3002      0.5520      0.7141
```

```
3     -0.3500      0.5256     -0.3874      0.6717
4     -0.0841     -0.7584     -0.6165      0.1941

            Random correlation matrix
            1           2           3           4
1       1.000      -0.236      -0.326      -0.110
2      -0.236       1.000       0.191      -0.017
3      -0.326       0.191       1.000      -0.435
4      -0.110      -0.017      -0.435       1.000
```

random_mvar_from_data

Generates pseudorandom numbers from a multivariate distribution determined from a given sample.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_mvar_from_data` (*int* `n_random`, *int* `ndim`, *int* `nsamp`, *float* `x[]`, *int* `nn`, ..., 0)

The type *double* function is `imsls_d_random_mvar_from_data`.

Required Arguments

int `n_random` (Input)
Number of random multivariate vectors to generate.

int `ndim` (Input)
The length of the multivariate vectors, that is, the number of dimensions.

int `nsamp` (Input)
Number of given data points from the distribution to be simulated.

float `x[]` (Input)
Array of size `nsamp` × `ndim` matrix containing the given sample.

int `nn` (Input)
Number of nearest neighbors of the randomly selected point in `x` that are used to form the output point in the result.

Return Value

`n_random` × `ndim` matrix containing the random multivariate vectors in its rows. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `* imsls_f_random_mvar_from_data` (*int* `n_random`, *int* `ndim`,
int `nsamp`, *float* `x[]`, *int* `nn`,

```
          IMSLS_X_COL_DIM, int x_col_dim,
          IMSLS_RETURN_USER, float r[],
          0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of the matrix `x`.
Default: `x_col_dim` = `ndim`

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` × `ndim` containing the random correlation matrix.

Description

Given a sample of size *n* (= `nsamp`) of observations of a *k*-variate random variable, `imsls_f_random_mvar_from_data` generates a pseudorandom sample with approximately the same moments as the given sample. The sample obtained is essentially the same as if sampling from a Gaussian kernel estimate of the sample density. (See Thompson 1989.) Routine `imsls_f_random_mvar_from_data` uses methods described by Taylor and Thompson (1986).

Assume that the (vector-valued) observations x_i are in the rows of *x*. An observation, x_j, is chosen randomly; its nearest *m* (= `nn`) neighbors,

$$x_{j_1}, x_{j_2}, \ldots, x_{j_m}$$

are determined; and the mean

$$\overline{x}_j$$

of those nearest neighbors is calculated. Next, a random sample

$u_1, u_2, \ldots, u_m$ is generated from a uniform distribution with lower bound

$$\frac{1}{m} - \sqrt{\frac{3(m-1)}{m^2}}$$

and upper bound

$$\frac{1}{m} + \sqrt{\frac{3(m-1)}{m^2}}$$

The random variate delivered is

$$\sum_{l=1}^{m} u_l \left(x_{jl} - \overline{x}_j \right) + \overline{x}_j$$

The process is then repeated until `n_random` such simulated variates are generated and stored in the rows of the result.

Example

In this example, `imsls_f_random_mvar_from_data` is used to generate 5 pseudorandom vectors of length 4 using the initial and final systolic pressure and the initial and final diastolic pressure from Data Set A in Afifi and Azen (1979) as the fixed sample from the population to be modeled. (Values of these four variables are in the seventh, tenth, twenty-first, and twenty-fourth columns of data set number nine in routine `imsls_f_data_sets`, Chapter 14, Utilities.)

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

void main()
{
  int i, nrrow, nrcol, nr = 5, k=4, nsamp = 113, nn = 5;
  float x[113][4], rdata[113][34], *r;

  imsls_random_seed_set(123457);


  imsls_f_data_sets(9,
                  IMSLS_N_OBSERVATIONS, &nrrow,
                  IMSLS_N_VARIABLES, &nrcol,
                  IMSLS_RETURN_USER, rdata,
                  0);
  for (i=0;i<nrrow;i++) x[i][0] = rdata[i][6];
  for (i=0;i<nrrow;i++) x[i][1] = rdata[i][9];
  for (i=0;i<nrrow;i++) x[i][2] = rdata[i][20];
  for (i=0;i<nrrow;i++) x[i][3] = rdata[i][23];

  r = imsls_f_random_mvar_from_data(nr, k, nsamp, x, nn, 0);
  imsls_f_write_matrix("Random variates", 5, 4, r, 0);
 }
```

Output

```
                 Random variates
            1           2           3           4
1       162.8        90.5       153.7       104.9
2       153.4        78.3       176.7        85.2
3        93.7        48.2       153.5        71.4
4       101.8        54.2       113.1        56.3
5        91.7        58.8        48.4        28.1
```

random_multinomial

Generates pseudorandom numbers from a multinomial distribution.

Synopsis

#include `<imsls.h>`

int `*imsls_random_multinomial` (*int* `n_random`, *int* `n`, *int* `k`,
float `p[]`, ..., 0)

Required Arguments

int `n_random` (Input)
Number of random multinomial vectors to generate.

int `n` (Input)
Multinomial parameter indicating the number of independent trials.

int `k` (Input)
The number of mutually exclusive outcomes on any trial. `k` is the length of the multinomial vectors. `k` must be greater than or equal to 2.

float `p[]` (Input)
Vector of length `k` containing the probabilities of the possible outcomes. The elements of `p` must be positive and must sum to 1.0.

Return Value

`n_random` by `k` matrix containing the random multinomial vectors in its rows. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_multinomial` (*int* `n_random`, *int* `n`, *int* `k`,
float `p[]`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `n_random` × `k` containing the random deviates.

Description

Routine `imsls_random_multinomial` generates pseudorandom numbers from a `K`-variate multinomial distribution with parameters `n` and `p`. `k` and `n` must be positive. Each element of `p` must be positive and the elements must sum to 1. The probability function (with n = `n`, k = `k`, and p_i = `p[`i+1`]`) is

$$f\left(x_1, x_2, \ldots, x_k\right) = \frac{n!}{x_1!x_2!\ldots x_k!} p_1^{x_1} p_2^{x_2} \ldots p_k^{x_k}$$

for $x_i \geq 0$ and

$$\sum_{i=0}^{k-1} x_i = n$$

The deviate in each row of r is produced by generation of the binomial deviate x_0 with parameters n and p_i and then by successive generations of the conditional binomial deviates x_j given $x_0, x_1, \ldots, x_{j-2}$ with parameters $n - x_0 - x_1 - \ldots - x_{j-2}$ and $p_j /(1 - p_0 - p_1 - \ldots - p_{j-2})$.

Example

In this example, `imsls_random_multinomial` is used to generate five pseudorandom 3-dimensional multinomial variates with parameters `n` = 20 and `p` = [0.1, 0.3, 0.6].

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

void main()
{
  int nr = 5, n = 20, k = 3, *ir;
  float p[3] = {.1, .3, .6};

  imsls_random_seed_set(123457);

  ir = imsls_random_multinomial(nr, n, k, p, 0);

  imsls_i_write_matrix("Multinomial random_deviates", 5, 3, ir,
                       IMSLS_NO_ROW_LABELS,
                       IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
Multinomial random_deviates
         5    4   11
         3    6   11
         3    3   14
         5    5   10
         4    5   11
```

random_sphere

Generates pseudorandom points on a unit circle or `K`-dimensional sphere

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_sphere` (*int* `n_random`, *int* `k`,..., 0)

The type *double* function is `imsls_d_random_sphere`.

Required Arguments

int `n_random` (Input)
Number of random numbers to generate.

int `k` (Input)
Dimension of the circle (`k` = 2) or of the sphere.

Return Value

`n_random` by `k` matrix containing the random Cartesian coordinates on the unit circle or sphere. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_sphere` (*int* `n_random`, *int* `k`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of size `n_random` by `k` containing the random Cartesian coordinates on the unit circle or sphere.

Description

Routine `imsls_f_random_sphere` generates pseudorandom coordinates of points that lie on a unit circle or a unit sphere in `K`-dimensional space. For points on a circle (`k` = 2), pairs of uniform (− 1, 1) points are generated and accepted only if they fall within the unit circle (the sum of their squares is less than 1), in which case they are scaled so as to lie on the circle.

For spheres in three or four dimensions, the algorithms of Marsaglia (1972) are used. For three dimensions, two independent uniform (− 1, 1) deviates U_1 and U_2 are generated and accepted only if the sum of their squares S_1 is less than 1. Then, the coordinates

$$Z_1 = 2U_1\sqrt{1-S_1}, \; Z_2 = 2U_2\sqrt{1-S_1}, \text{ and } Z_3 = 1-2S_1$$

are formed. For four dimensions, U_1, U_2, and S_1 are produced as described above. Similarly, U_3, U_4, and S_2 are formed. The coordinates are then

$$Z_1 = U_1, \; Z_2 = U_2, \; Z_3 = U_3\sqrt{\left(1-S_1\right)/S_2}$$

and

$$Z_4 = U_4\sqrt{\left(1-S_1\right)/S_2}$$

For spheres in higher dimensions, `K` independent normal deviates are generated and scaled so as to lie on the unit sphere in the manner suggested by Muller (1959).

Example

In this example, `imsls_f_random_sphere` is used to generate two uniform random deviates from the surface of the unit sphere in three space.

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

void main()
{
  int n_random = 2;
  int k = 3;
  float *z;
  char *rlabel[] = {"First point",
                   "Second point"};

  imsls_random_seed_set(123457);

  z = imsls_f_random_sphere(n_random, k, 0);

  imsls_f_write_matrix("Coordinates", n_random, k, z,
                      IMSLS_ROW_LABELS, rlabel,
                      IMSLS_NO_COL_LABELS,
                      0);
 }
```

Output

```
                   Coordinates
First point       0.8893      0.2316      0.3944
Second point      0.1901      0.0396     -0.9810
```

random_table_twoway

Generates a pseudorandom two-way table.

Synopsis

#include `<imsls.h>`

int `*imsls_random_table_twoway` (*int* `nrow`, *int* `ncol`, *int* `nrtot[]`, *int* `nctot[]`,..., 0)

Required Arguments

int `nrow` (Input)
Number of rows in the table.

int `ncol` (Input)
Number of columns in the table.

int `nrtot[]` (Input)
Array of length `nrow` containing the row totals.

int `nctot[]` (Input)
Array of length `ncol` containing the column totals. (Input)
The elements of `nrtot` and `nctot` must be nonnegative and must sum to the same quantity.

Return Value

`nrow` by `ncol` random matrix with the given row and column totals. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_table_twoway` (*int* `nrow`, *int* `ncol`, *int* `nrtot[]`,
int `nctot[]`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of size `nrow` by `ncol` containing the random matrix with the given row and column totals.

Description

Routine `imsls_random_table_twoway` generates pseudorandom entries for a two-way contingency table with fixed row and column totals. The method depends on the size of the table and the total number of entries in the table. If the total number of entries is less than twice the product of the number of rows and columns, the method described by Boyette (1979) and by Agresti, Wackerly, and Boyette (1979) is used. In this method, a work vector is filled with row indices so that the number of times each index appears equals the given row total. This vector is then randomly permuted and used to increment the entries in each row so that the given row total is attained.

For tables with larger numbers of entries, the method of Patefield (1981) is used. This method can be considerably faster in these cases. The method depends on the conditional probability distribution of individual elements, given the entries in the previous rows. The probabilities for the individual elements are computed starting from their conditional means.

Example

In this example, `imsls_random_table_twoway` is used to generate a two by three table with row totals 3 and 5, and column totals 2, 4, and 2.

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

void main()
{
  int *itable, nrow = 2, ncol = 3;
  int nrtot[2] = {3, 5};
  int nctot[3] = {2, 4, 2};
  char *title = "A random contingency table with fixed marginal totals";

  imsls_random_seed_set(123457);


  itable = imsls_random_table_twoway(nrow, ncol, nrtot, nctot, 0);

  imsls_i_write_matrix(title, nrow, ncol, itable,
                      IMSLS_NO_ROW_LABELS,
                      IMSLS_NO_COL_LABELS,
                      0);
  }
```

Output

```
A random contingency table with fixed marginal totals
                      0   2   1
                      2   2   1
```

random_order_normal

Generates pseudorandom order statistics from a standard normal distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_order_normal` (*int* `ifirst`, *int* `ilast`, *int* `n`,...,
0)

The type *double* function is `imsls_d_random_order_normal`.

Required Arguments

int `ifirst` (Input)
First order statistic to generate.

int `ilast` (Input)
Last order statistic to generate.
`ilast` must be greater than or equal to `ifirst`. The full set of order

statistics from `ifirst` to `ilast` is generated. If only one order statistic is desired, set `ilast` = `ifirst`.

int `n` (Input)
Size of the sample from which the order statistics arise.

Return Value

An array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.
The first element is the `ifirst` order statistic in a random sample of size `n` from the standard normal distribution. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_order_normal` (*int* `ifirst`, *int* `ilast`, *int* `n`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.

Description

Routine `imsls_f_random_order_normal` generates the `ifirst` through the `ilast` order statistics from a pseudorandom sample of size `N` from a normal (0, 1) distribution. Routine `imsls_f_random_order_normal` uses the routine `imsls_f_random_order_uniform` (page 829) to generate order statistics from the uniform (0, 1) distribution and then obtains the normal order statistics using the inverse CDF transformation.

Each call to `imsls_f_random_order_normal` yields an independent event so order statistics from different calls may not have the same order relations with each other.

Example

In this example, `imsls_f_random_order_normal` is used to generate the fifteenth through the nineteenth order statistics from a sample of size twenty.

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

void main()
{
  float *r = NULL;

  imsls_random_seed_set(123457);
  r = imsls_f_random_order_normal(15, 19, 20, 0);

  printf("The 15th through the 19th order statistics from a \n");
  printf("random sample of size 20 from a normal distribution\n");
  imsls_f_write_matrix("", 5, 1, r, 0);
}
```

Output

```
The 15th through the 19th order statistics from a
random sample of size 20 from a normal distribution

1       0.4056
2       0.4681
3       0.4697
4       0.9067
5       0.9362
```

random_order_uniform

Generates pseudorandom order statistics from a uniform (0, 1) distribution.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_order_uniform` (*int* `ifirst`, *int* `ilast`,
int `n`,..., 0)

The type *double* function is `imsls_d_random_order_uniform`.

Required Arguments

int `ifirst` (Input)
First order statistic to generate.

int `ilast` (Input)
Last order statistic to generate.
`ilast` must be greater than or equal to `ifirst`. The full set of order statistics from `ifirst` to `ilast` is generated. If only one order statistic is desired, set `ilast` = `ifirst`.

int `n` (Input)
Size of the sample from which the order statistics arise.

Return Value

An array of length `ilast` + 1 – `ifirst` containing the random order statistics in ascending order.
The first element is the `ifirst` order statistic in a random sample of size `n` from the uniform (0, 1) distribution. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_order_uniform` (*int* `ifirst`, *int* `ilast`, *int* `n`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `ilast` + 1 − `ifirst` containing the random order statistics in ascending order.

Description

Routine `imsls_f_random_order_uniform` generates the `ifirst` through the `ilast` order statistics from a pseudorandom sample of size `n` from a uniform (0, 1) distribution. Depending on the values of `ifirst` and `ilast`, different methods of generation are used to achieve greater efficiency. If `ifirst` = 1 and `ilast` = `n`, that is, if the full set of order statistics are desired, the spacings between successive order statistics are generated as ratios of exponential variates. If the full set is not desired, a beta variate is generated for one of the order statistics, and the others are generated as extreme order statistics from conditional uniform distributions. Extreme order statistics from a uniform distribution can be obtained by raising a uniform deviate to an appropriate power.

Each call to `imsls_f_random_order_uniform` yields an independent event. This means, for example, that if on one call the fourth order statistic is requested and on a second call the third order statistic is requested, the “fourth” may be smaller than the “third”. If both the third and fourth order statistics from a given sample are desired, they should be obtained from a single call to `imsls_f_random_order_uniform` (by specifying `ifirst` less than or equal to 3 and `ilast` greater than or equal to 4).

Example

In this example, `imsls_f_random_order_uniform` is used to generate the fifteenth through the nineteenth order statistics from a sample of size twenty.

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

void main()
{
  float *r = NULL;

  imsls_random_seed_set(123457);

  r = imsls_f_random_order_uniform(15, 19, 20, 0);

  printf("The 15th through the 19th order statistics from a \n");
  printf("random sample of size 20 from a uniform distribution\n");
  imsls_f_write_matrix("", 5, 1, r, 0);
}
```

Output

```
The 15th through the 19th order statistics from a
random sample of size 20 from a uniform distribution

1      0.6575
2      0.6802
3      0.6807
4      0.8177
5      0.8254
```

random_arma

Generates a time series from a specific ARMA model.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_arma` (*int* `n_observations`, *int* `p`, *float* `ar[]`,
int `q`, *float* `ma[]`, ..., 0)

The type double function is `imsls_d_random_arma`.

Required Arguments

int `n_observations` (Input)
Number of observations to be generated. Parameter `n_observations` must be greater than or equal to one.

int `p` (Input)
Number of autoregressive parameters. Paramater `p` must be greater than or equal to zero.

float `ar[]` (Input)
Array of length `p` containing the autoregressive parameters.

int `q` (Input)
Number of moving average parameters. Parameter `q` must be greater than or equal to zero.

float `ma[]` (Input)
Array of length `q` containing the moving average parameters.

Return Value

An array of length `n_observations` containing the generated time series.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_arma` (*int* `n_observations`, *int* `p`, *float* `ar[]`,
int q, *float* `ma[]`,
`IMSLS_ARMA_CONSTANT`, *float* `constant`,
`IMSLS_VAR_NOISE`, *float* `*a_variance`,
`IMSLS_INPUT_NOISE`, *float* `*a_input`,
`IMSLS_OUTPUT_NOISE`, *float* `**a_return`,
`IMSLS_OUTPUT_NOISE_USER`, *float* `a_return[]`,
`IMSLS_NONZERO_ARLAGS`, *int* `*ar_lags`,
`IMSLS_NONZERO_MALAGS`, *int* `*ma_lags`,
`IMSLS_INITIAL_W`, *float* `*w_initial`,
`IMSLS_ACCEPT_REJECT_METHOD`,
`IMSLS_RETURN_USER`, *float* `w[]`,
0)

Optional Arguments

`IMSLS_ARMA_CONSTANT`, *float* `constant` (Input)
Overall constant. See “Description”.
Default: `constant` = 0

`IMSLS_VAR_NOISE`, *float* `a_variance` (Input)
If `IMSLS_VAR_NOISE` is specified (and `IMSLS_INPUT_NOISE` is *not* specified) the noise a_t will be generated from a normal distribution with mean 0 and variance `a_variance`.
Default: `a_variance` = 1.0

`IMSLS_INPUT_NOISE`, *float* `*a_input` (Input)
If `IMSLS_INPUT_NOISE` is specified, the user will provide an array of length `n_observations` + max (`ma_lags`[*i*]) containing the random noises. If this option is specified, then `IMSLS_VAR_NOISE` should not be specified (a warning message will be issued and the option `IMSLS_VAR_NOISE` will be ignored).

`IMSLS_OUTPUT_NOISE`, *float* `**a_return` (Output)
An address of a pointer to an internally allocated array of length `n_observations` + max (`ma_lags`[*i*]) containing the random noises.

`IMSLS_OUTPUT_NOISE_USER`, *float* `a_return[]` (Output)
Storage for array `a_return` is provided by user. See `IMSLS_OUTPUT_NOISE`.

`IMSLS_NONZERO_ARLAGS`, *int* `ar_lags[]` (Input)
An array of length `p` containing the order of the nonzero autoregressive parameters.
Default: `ar_lags` = [1, 2, ..., `p`]

`IMSLS_NONZERO_MALAGS`, *int* `ma_lags` (Input)
An array of length `q` containing the order of the nonzero moving average parameters.
Default: `ma_lags` = [1, 2, ..., `q`]

`IMSLS_INITIAL_W`, *float* `w_initial[]` (Input)
: Array of length max (`ar_lags`[*i*]) containing the initial values of the time series.
Default: all the elements in `w_initial` = `constant`/(1 − `ar` [0] − `ar` [1] − … − `ar` [`p` − 1])

`IMSLS_ACCEPT_REJECT_METHOD` (Input)
: If `IMSLS_ACCEPT_REJECT_METHOD` is specified, the random noises will be generated from a normal distribution using an acceptance/rejection method. If `IMSLS_ACCEPT_REJECT_METHOD` is not specified, the random noises will be generated using an inverse normal CDF method. This argument will be ignored if `IMSLS_INPUT_NOISE` is specified.

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
: User-supplied array of length `n_random` containing the generated time series.

Description

Function `imsls_f_random_arma` simulates an ARMA(p, q) process, $\{W_t\}$, for $t = 1, 2, ..., n$ (with n = `n_observations`, p = `p`, and q = `q`). The model is

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

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^P$$

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$$

Let μ be the mean of the time series $\{W_t\}$. The overall constant θ_0 (`constant`) is

$$\theta_0 = \begin{cases} \mu & p = 0 \\ \mu\left(1 - \sum_{i=1}^{p} \phi_i\right) & p > 0 \end{cases}$$

Time series whose innovations have a nonnormal distribution may be simulated by providing the appropriate innovations in `a_input` and start values in `w_initial`.

The time series is generated according to the followng model:

$$X[i] = \texttt{constant} + \texttt{ar[0]} \cdot X[i - \texttt{ar\_lags[0]}] + \ldots +$$
$$\texttt{ar[p − 1]} \cdot X[i - \texttt{ar\_lags[p − 1]}] +$$
$$A[i] - \texttt{ma[0]} \cdot A[i - \texttt{ma\_lags[0]}] - \ldots -$$
$$\texttt{ma[q − 1]} \cdot A[i - \texttt{ma\_lags[q − 1]}]$$

where the constant is related to the mean of the series,

$$\overline{W}$$

as follows:

$$\text{constant} = \overline{W} \cdot (1 - \text{ar}[0] - \ldots - \text{ar}[q-1])$$

and where

$X[t] = W[t], \qquad t = 0, 1, \ldots,$ n_observations $- 1$

and

$W[t] =$ w_initial$[t + p], \qquad t = -p, -p + 1, \ldots, -2, -1$

and A is either a_input (if IMSLS_INPUT_NOISE is specified) or a_return (otherwise).

Examples

Example 1

In this example, imsls_f_random_arma is used to generate a time series of length five, using an ARMA model with three autoregressive parameters and two moving average parameters. The start values are 0.1000, 0.0500, and 0.0375.

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

void main()
{
    int   n_random = 5;
    int   np = 3;
    float phi[3] = {0.5, 0.25, 0.125};
    int   nq = 2;
    float theta[2] = {-0.5, -0.25};
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_arma(n_random, np, phi, nq, theta, 0);
    imsls_f_write_matrix("ARMA random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                ARMA random deviates:
     0.863       0.809       1.904       0.110       2.266
```

Example 2

In this example, a time series of length 5 is generated using an ARMA model with 4 autoregressive parameters and 2 moving average parameters. The start values are 0.1, 0.05 and 0.0375.

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

void main()
{
    int   n_random = 5;
    int   np = 3;
    float phi[3] = {0.5, 0.25, 0.125};
    int   nq = 2;
    float theta[2] = {-0.5, -0.25};
    float wi[3] = {0.1, 0.05, 0.0375};
    float theta0 = 1.0;
    float avar   = 0.1;
    float *r;

    imsls_random_seed_set(123457);
    r = imsls_f_random_arma(n_random, np, phi, nq, theta,
        IMSLS_ACCEPT_REJECT_METHOD,
        IMSLS_INITIAL_W, wi,
        IMSLS_ARMA_CONSTANT, theta0,
        IMSLS_VAR_NOISE, avar,
        0);
    imsls_f_write_matrix("ARMA random deviates:",
        1, n_random, r, IMSLS_NO_COL_LABELS, 0);
}
```

Output

```
                 ARMA random deviates:
     1.403       2.220       2.286       2.888       2.832
```

Warning Errors

`IMSLS_RNARM_NEG_VAR`	VAR(a) = “a_variance” = #, VAR(a) must be greater than 0. The absolute value of # is used for VAR(a).
`IMSLS_RNARM_IO_NOISE`	Both `IMSLS_INPUT_NOISE` and `IMSLS_OUTPUT_NOISE` are specified. `IMSLS_INPUT_NOISE` is used.

random_npp

Generates pseudorandom numbers from a nonhomogeneous Poisson process.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_npp` (*float* `tbegin`, *float* `tend`, *float* `ftheta()`,
    float `theta_min`, *float* `theta_max`, *int* `neub`, *int* `*ne`, ..., 0)

The type *double* function is `imsls_d_random_npp`.

Required Arguments

float `tbegin` (Input)
    Lower endpoint of the time interval of the process.
    `tbegin` must be nonnegative. Usually, `tbegin` = 0.

float `tend` (Input)
: Upper endpoint of the time interval of the process.
`tend` must be greater than `tbegin`.

float `ftheta(`*float* `t)` (Input)
: User-supplied function to provide the value of the rate of the process as a function of time. This function must be defined over the interval from `tbegin` to `tend` and must be nonnegative in that interval.

float `theta_min` (Input)
: Minimum value of the rate function `ftheta()` in the interval (`tbegin`, `tend`).
If the actual minimum is unknown, set `theta_min` = 0.0.

float `theta_max` (Input)
: Maximum value of the rate function `ftheta()` in the interval (`tbegin`, `tend`).
If the actual maximum is unknown, set `theta_max` to a known upper bound of the maximum. The efficiency of `imsls_f_random_npp` is less the greater `theta_max` exceeds the true maximum.

int `neub` (Input)
: Upper bound on the number of events to be generated.
In order to be reasonably sure that the full process through time `tend` is generated, calculate `neub` as `neub` = `X` + 10.0 `*` `SQRT`(`X`), where `X` = `theta_max` `*` (`tend` – `tbegin`).

int `*ne` (Output)
: Number of events actually generated.
If `ne` is less that `neub`, the time `tend` is reached before `neub` events are realized.

Return Value

An array of length `neub` containing the the times to events in the first `ne` elements. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_npp` (*float* `tbegin`, *float* `tend`, *float* `ftheta()`,
float `theta_min`, *float* `theta_max`, *int* `neub`, *int* `*ne`,
`IMSLS_RETURN_USER`, *float* `r[]`,
`IMSLS_FCN_W_DATA`, *float* `ftheta()`, *void* `*data`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length `neub` containing the the times to events in the first `ne` elements.

`IMSLS_FCN_W_DATA`, *float* `ftheta(`*float* `t)`, *void* `*data`, (Input)
User-supplied function to provide the value of the rate of the process as a function of time, which also accepts a pointer to data that is supplied by the user. `data` is a pointer to the data to be passed to the user-supplied function. See the *Introduction, Passing Data to User-Supplied Functions* at the beginning of this manual for more details.

Description

Routine `imsls_f_random_npp` simulates a one-dimensional nonhomogeneous Poisson process with rate function `ftheta` in a fixed interval (`tbegin`, `tend`].

Let $\lambda(t)$ be the rate function and t_0 = `tbegin` and t_1 = `tend`. Routine `imsls_f_random_npp` uses a method of thinning a nonhomogeneous Poisson process $\{N^*(t), t \geq t_0\}$ with rate function $\lambda^*(t) \geq \lambda(t)$ in $(t_0, t_1]$, where the number of events, N^*, in the interval $(t_0, t_1]$ has a Poisson distribution with parameter

$$\mu_0 = \int_{t_0}^{t_1} \lambda(t)\, dt$$

The function

$$\Lambda(t) = \int_0^{t'} \lambda(t)\, dt$$

is called the *integrated rate function*.) In `imsls_f_random_npp`, $\lambda^*(t)$ is taken to be a constant λ^*(= `theta_max`) so that at time t_i, the time of the next event t_{i+1} is obtained by generating and cumulating exponential random numbers

$$E^*_{1,i}, E^*_{2,i}, \ldots,$$

with parameter λ^*, until for the first time

$$u_{j,i} \leq \left(t_i + E^*_{1,i} + \ldots + E^*_{j,i}\right) / \lambda^*$$

where the $u_{j,i}$ are independent uniform random numbers between 0 and 1. This process is continued until the specified number of events, `neub`, is realized or until the time, `tend`, is exceeded. This method is due to Lewis and Shedler (1979), who also review other methods. The most straightforward (and most efficient) method is by inverting the integrated rate function, but often this is not possible.

If `theta_max` is actually greater than the maximum of $\lambda(t)$ in $(t_0, t_1]$, the routine will work, but less efficiently. Also, if $\lambda(t)$ varies greatly within the interval, the efficiency is reduced. In that case, it may be desirable to divide the time interval into subintervals within which the rate function is less variable. This is possible because the process is without memory.

If no time horizon arises naturally, `tend` must be set large enough to allow for the required number of events to be realized. Care must be taken, however, that `ftheta` is defined over the entire interval.

After simulating a given number of events, the next event came be generated by setting `tbegin` to the time of the last event (the sum of the elements in `R`) and calling `imsls_f_random_npp` again. Cox and Lewis (1966) discuss modeling applications of nonhomogeneous Poisson processes.

Example

In this example, `imsls_f_random_npp` is used to generate the first five events in the time 0 to 20 (if that many events are realized) in a nonhomogeneous process with rate function

$$\lambda(t) = 0.6342 \text{ e}0.001427^{t}$$

for $0 < t \le 20$.

Since this is a monotonically increasing function of t, the minimum is at $t = 0$ and is 0.6342, and the maximum is at $t = 20$ and is 0.6342 e0.02854 = 0.652561.

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

void main()
{
  int i, neub = 5, ne;
  float  *r, tmax= .652561, tmin = .6342, tbeg=0., tend=20.;

  imsls_random_seed_set(123457);

  r = imsls_f_random_npp(tbeg, tend, ftheta, tmin, tmax, neub, &ne, 0);

  printf("Inter-event times for the first %d events in the process:\n", ne);
  for (i=0; i<ne; i++) printf("\t%f\n", r[i]);

}
```

Output

```
Inter-event times for the first 5 events in the process:
       0.052660
       0.407979
       0.258399
       0.019767
       0.167641
```

random_permutation

Generates a pseudorandom permutation.

Synopsis

#include `<imsls.h>`

int `*imsls_random_permutation` (*int* `k`, ..., 0)

Required Arguments

int `k` (Input)
Number of integers to be permuted.

Return Value

An array of length `k` containing the random permutation of the integers from 1 to `k`. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_permutation` (*int* `k`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of length `k` containing the random permutation of the integers from 1 to `k`.

Description

Routine `imsls_random_permutation` generates a pseudorandom permutation of the integers from 1 to `k`. It begins by filling a vector of length `k` with the consecutive integers 1 to `k`. Then, with M initially equal to `k`, a random index J between 1 and M (inclusive) is generated. The element of the vector with the index M and the element with index J swap places in the vector. M is then decremented by 1 and the process repeated until $M = 1$.

Example

In this example, `imsls_random_permutation` is called to produce a pseudorandom permutation of the integers from 1 to 10.

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

void main()
{
  int *ir, k = 10;

  imsls_random_seed_set(123457);

  ir = imsls_random_permutation(k, 0);

  printf("Random permutation of the integers from 1 to 10\n");
  imsls_i_write_matrix("", 1, k, ir,
                     IMSLS_NO_COL_LABELS, 0);
 }
```

Output

```
Random permutation of the integers from 1 to 10

  5    9    2    8    1    6    4    7    3   10
```

random_sample_indices

Generates a simple pseudorandom sample of indices.

Synopsis

#include `<imsls.h>`

int `*imsls_random_sample_indices` (*int* `nsamp`, *int* `npop`, ..., 0)

Required Arguments

int `nsamp` (Input)
Sample size desired.

int `npop` (Input)
Number of items in the population.

Return Value

An array of length `nsamp` containing the indices of the sample. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

int `*imsls_random_sample_indices` (*int* `nsamp`, *int* `npop`,
`IMSLS_RETURN_USER`, *int* `ir[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `ir[]` (Output)
User-supplied array of length `nsamp` containing the indices of the sample.

Description

Routine `imsls_random_sample_indices` generates the indices of a pseudorandom sample,without replacement, of size `nsamp` numbers from a population of size `npop`. If `nsamp` is greater than `npop`/2, the integers from 1 to `npop` are selected sequentially with a probability conditional on the number selected and the number remaining to be considered. If, when the *i*-th population index is considered, *j* items have been included in the sample, then the index *i* is included with probability (`nsamp` $- j$)/(`npop` $+ 1 - i$).

If `nsamp` is not greater than `npop`/2, a O(`nsamp`) algorithm due to Ahrens and Dieter (1985) is used. Of the methods discussed by Ahrens and Dieter, the one called SG* is used in `imsls_random_sample_indices`. It involves a preliminary selection of q indices using a geometric distribution for the distances between each index and the next one. If the preliminary sample size q is less than `nsamp`, a new preliminary sample is chosen, and this is continued until a preliminary sample greater in size than `nsamp` is chosen. This preliminary sample is then thinned using the same kind of sampling as described above for the case in which the sample size is greater than half of the population size. Routine `imsls_random_sample_indices` does not store the preliminary sample indices, but rather restores the state of the generator used in selecting the sample initially, and then passes through once again, making the final selection as the preliminary sample indices are being generated.

Example

In this example, `imsls_random_sample_indices` is used to generate the indices of a pseudorandom sample of size 5 from a population of size 100.

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

void main()
{
  int *ir, nsamp = 5, npop = 100;

  imsls_random_seed_set(123457);

  ir = imsls_random_sample_indices(nsamp, npop, 0);

  imsls_i_write_matrix("Random Sample", 1, nsamp, ir,
                       IMSLS_NO_COL_LABELS, 0);
 }
```

Output

```
Random Sample

             2    22    53    61    79
```

random_sample

Generates a simple pseudorandom sample from a finite population.

Synopsis

#include `<imsls.h>`

float `*imsls_f_random_sample` (*int* `nrow`, *int* `nvar`, *float* `population[]`,
int `nsamp`,..., 0)

The type *double* function is `imsls_d_random_sample`.

Required Arguments

int `nrow` (Input)
Number of rows of data in `population`.

int `nvar` (Input)
Number of variables in the population and in the sample.

float `population[]` (Input)
`nrow` by `nvar` matrix containing the population to be sampled. If either of the optional arguments `IMSLS_FIRST_CALL` or `IMSLS_ADDITIONAL_CALL` are specified, then `population` contains a different part of the population on each invocation, otherwise `population` contains the entire population.

int `nsamp` (Input)
The sample size desired.

Return Value

`nsamp` by `nvar` matrix containing the sample. To release this space, use `free`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_random_sample` (*int* `nrow`, *int* `nvar`, *float* `population[]`,
int `nsamp`,
`IMSLS_FIRST_CALL`, *int* `**index`, *int* `*npop`
`IMSLS_FIRST_CALL_USER`, *int* `index[]`, *int* `*npop`
`IMSLS_ADDITIONAL_CALL`, *int* `*index`, *int* `*npop`, *float* `*samp`,
`IMSLS_POPULATION_COL_DIM`, *int* `population_col_dim`,

`IMSLS_RETURN_USER`, *int* `samp[]`,
0)

Optional Arguments

`IMSLS_FIRST_CALL`, *int* `**index`, *int* `*npop` (Output)
: This is the first invocation with this data; additional calls to `imsls_f_random_sample` may be made to add to the population. Additional calls should be made using the optional argument `IMSLS_ADDITIONAL_CALL`. Argument `index` is the address of a pointer to an internally allocated array of length `nsamp` containing the indices of the sample in the population. Argument `npop` returns the number of items in the population. If the population is input a few items at a time, the first call to `imsls_f_random_sample` should use `IMSLS_FIRST_CALL`, and subsequent calls should use `IMSLS_ADDITIONAL_CALL`. See example 2.

`IMSLS_FIRST_CALL_USER`, *int* `index[]`, *int* `*npop` (Output)
: Storage for `index` is provided by the user. See `IMSLS_FIRST_CALL`.

`IMSLS_ADDITIONAL_CALL`, *int* `*index`, *int* `*npop`, *float* `*samp`
: (Input/Output)
This is an additional invocation of `imsls_f_random_sample`, and updating for the subpopulation in `population` is performed. Argument `index` is a pointer to an array of length `nsamp` containing the indices of the sample in the population, as returned using optional argument `IMSLS_FIRST_CALL`. Argument `npop`, also obtained using optional argument `IMSLS_FIRST_CALL`, returns the number of items in the population. It is not necessary to know the number of items in the population in advance. `npop` is used to cumulate the population size and should not be changed between calls to `imsls_f_random_sample`. Argument `samp` is a pointer to the array of size `nsamp` by `nvar` containing the sample. `samp` is the result of calling `imsls_f_random_sample` with optional argument `IMSLS_FIRST_CALL`. See example 2

`IMSLS_POPULATION_COL_DIM`, *int* `population_col_dim` (Input)
: Column dimension of the matrix `population`.
Default: `x_col_dim` = `nvar`

`IMSLS_RETURN_USER`, *int* `samp[]` (Output)
: User-supplied array of size `nrow` by `nvar` containing the sample. This option should not be used if `IMSLS_ADDITIONAL_CALL` is used.

Description

Routine `imsls_f_random_sample` generates a pseudorandom sample from a given population, without replacement, using an algorithm due to McLeod and Bellhouse (1983).

The first `nsamp` items in the population are included in the sample. Then, for each successive item from the population, a random item in the sample is replaced by that item from the population with probability equal to the sample size divided by the number of population items that have been encountered at that time.

Example 1

In this example, `imsls_f_random_sample` is used to generate a sample of size 5 from a population stored in the matrix `population`.

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

void main()
{
  int nrow = 176, nvar = 2, nsamp = 5;
  float *population;
  float *sample;

  population = imsls_f_data_sets(2, 0);

  imsls_random_seed_set(123457);

  sample = imsls_f_random_sample(nrow, nvar, population, nsamp, 0);

  imsls_f_write_matrix("The sample", nsamp, nvar, sample,
                      IMSLS_NO_ROW_LABELS,
                      IMSLS_NO_COL_LABELS,
                      0);
}
```

Output

```
      The sample
      1764         36
      1828         62
      1923          6
      1773         35
      1769        106
```

Example 2

Routine `imsls_f_random_sample` is now used to generate a sample of size 5 from the same population as in the example above except the data are input to `RNSRS` one observation at a time. This is the way `imsls_f_random_sample` may be used to sample from a file on disk or tape. Notice that the number of records need not be known in advance.

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

```
void main()
{
  int i, nrow = 176, nvar = 2, nsamp = 5;
  int *index, npop;
  float *population;
  float *sample;

  population = imsls_f_data_sets(2, 0);

  imsls_random_seed_set(123457);

  sample = imsls_f_random_sample(1, 2, population, nsamp,
                                 IMSLS_FIRST_CALL, &index, &npop,
                                 0);
  for (i = 1; i < 176; i++) {
    imsls_f_random_sample(1, 2, &population[2*i], nsamp,
                          IMSLS_ADDITIONAL_CALL, index, &npop, sample,
                          0);
  }
  printf("The population size is %d\n", npop);
  imsls_i_write_matrix("Indices of random sample", 5, 1, index, 0);


  imsls_f_write_matrix("The sample", nsamp, nvar, sample,
                       IMSLS_NO_ROW_LABELS,
                       IMSLS_NO_COL_LABELS,
                       0);
 }
```

Output

```
The population size is 176

Indices of random sample
         1    16
         2    80
         3   175
         4    25
         5    21

      The sample
      1764         36
      1828         62
      1923          6
      1773         35
      1769        106
```

random_option

Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator or a generalized feedback shift register (GFSR) method.

Synopsis

#include `<imsls.h>`

void `imsls_random_option` (*int* `generator_option`)

Required Arguments

int `generator_option` (Input)
Indicator of the generator. Argument `generator_option` is used to choose the multiplier and whether or not shuffling is done, or the `GFSR` method.

`generator_option`	Generator
1	The multiplier 16807 is used.
2	The multiplier 16807 is used with shuffling.
3	The multiplier 397204094 is used.
4	The multiplier 397204094 is used with shuffling.
5	The multiplier 950706376 is used.
6	The multiplier 950706376 is used with shuffling.
7	GFSR, with the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$ is used

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).

Example

See function `imsls_random_GFSR_table_get` (page 853).

random_option_get

Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.

Synopsis

#include `<imsls.h>`

int `imsls_random_option_get` ()

Return Value

Indicator of the generator.

`result`	**Generator**
1	The multiplier 16807 is used.
2	The multiplier 16807 is used with shuffling.
3	The multiplier 397204094 is used.
4	The multiplier 397204094 is used with shuffling.
5	The multiplier 950706376 is used.
6	The multiplier 950706376 is used with shuffling.
7	GFSR, with the recursion $X_t = X_{t\text{-}1563} \oplus X_{t\text{-}96}$ is used

Description

The routine `imsls_random_option_get` retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator or the `GRSR` method. The uniform pseudorandom number generators use a multiplicative congruential method, with or without shuffling. The value of the multiplier and whether or not to use shuffling are determined by `imsls_random_option`.

random_seed_get

Retrieves the current value of the seed used in the random number generators.

Synopsis

#include `<imsls.h>`

int `imsls_random_seed_get` ()

Return Value

The value of the seed.

Description

Function `imsls_random_seed_get` retrieves the current value of the “seed” used in the random number generators. A reason for doing this would be to restart a simulation, using function `imsls_random_seed_set` to reset the seed.

Example

This example illustrates the statements required to restart a simulation using `imsls_random_seed_get` and `imsls_random_seed_set`. The example shows that restarting the sequence of random numbers at the value of the seed last generated is the same as generating the random numbers all at once.

```
#include <imsls.h>

#define     N_RANDOM     5

main()
{
    int         seed = 123457;
    float      *r1, *r2, *r;

    imsls_random_seed_set(seed);
    r1 = imsls_f_random_uniform(N_RANDOM, 0);
    imsls_f_write_matrix ("First Group of Random Numbers", 1,
                          N_RANDOM, r1, 0);
    seed = imsls_random_seed_get();

    imsls_random_seed_set(seed);
    r2 = imsls_f_random_uniform(N_RANDOM, 0);
    imsls_f_write_matrix ("Second Group of Random Numbers", 1,
                          N_RANDOM, r2, 0);

    imsls_random_seed_set(123457);
    r = imsls_f_random_uniform(2*N_RANDOM, 0);
    imsls_f_write_matrix ("Both Groups of Random Numbers", 1,
                          2*N_RANDOM, r, 0);
}
```

Output

```
              First Group of Random Numbers
     1           2           3           4           5
0.9662      0.2607      0.7663      0.5693      0.8448

             Second Group of Random Numbers
     1           2           3           4           5
0.0443      0.9872      0.6014      0.8964      0.3809

                    Both Groups of Random Numbers
     1           2           3           4           5           6
0.9662      0.2607      0.7663      0.5693      0.8448      0.0443

     7           8           9          10
0.9872      0.6014      0.8964      0.3809
```

random_substream_seed_get

Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.

Synopsis

#include `<imsls.h>`

int `imsls_random_substream_seed_get` (*int* `iseed1`)

Required Arguments

int `iseed1` (Input)
The seed that yields the first stream.

Return Value

The seed that yields a stream beginning 100,000 numbers beyond the stream that begins with `iseed1`.

Description

Given a seed, `iseed1`, `imsls_random_substream_seed_get` determines another seed, such that if one of the IMSL multiplicative congruential generators, using no shuffling, went through 100,000 generations starting with `iseed1`, the next number in that sequence would be the first number in the sequence that begins with the returned seed.

Note that `imsls_random_substream_seed_get` works only when a multiplicative congruential generator without shuffling is used. This means that either the routine `imsls_random_option` has not been called at all or that it has been last called with `generator_option` taking a value of 1, 3, or 5.

For many of the IMSL generators for nonuniform distributions that do not use the inverse CDF method, the distance between the sequences generated starting with `iseed1` and starting with the returned seed may be less than 100,000. This is because the nonuniform generators that use other techniques may require more than one uniform deviate for each output deviate.

The reason that one may want two seeds that generate sequences a known distance apart is for blocking Monte Carlo experiments or for running parallel streams

Example

In this example, `imsls_random_substream_seed_get` is used to determine seeds for 4 separate streams, each 200,000 numbers apart, for a multiplicative congruential generator without shuffling. (Since `imsls_random_option` is not invoked to select a generator, the multiplier is 16807.) Since the streams are 200,000 numbers apart, each seed requires two invocations of `imsls_random_substream_seed_get`. All of the streams are non-overlapping, since the period of the underlying generator is 2,147,483,646. The resulting seed are then verified by checking the seed after generating random sequences of length 200,000.

```
#include <imsls.h>

main()
{
  int i, is1, is2, is3, is4;
  float *r;
  is1 = 123457;
  is2 = imsls_random_substream_seed_get(is1);
  is2 = imsls_random_substream_seed_get(is2);
  is3 = imsls_random_substream_seed_get(is2);
  is3 = imsls_random_substream_seed_get(is3);
  is4 = imsls_random_substream_seed_get(is3);
  is4 = imsls_random_substream_seed_get(is4);
  printf("Seeds for four separate streams:\n");
  printf("%d\t%d\t%d\t%d\n\n", is1, is2, is3, is4);

  imsls_random_seed_set(is1);
  for (i=0;i<3;i++) {
    r = imsls_f_random_uniform(200000, 0);
    printf("seed after %d random numbers: %d\n", (i+1)*200000,
           imsls_random_seed_get());
    if (r) free(r);
  }
}
```

Output

```
Seeds for four separate streams:
123457 2016130173   85016329     979156171

seed after 200000 random numbers: 2016130173
seed after 400000 random numbers: 85016329
seed after 600000 random numbers: 979156171
```

random_seed_set

Initializes a random seed for use in the random number generators.

Synopsis

#include `<imsls.h>`

void `imsls_random_seed_set` (*int* `seed)`

Required Arguments

int `seed` (Input)
The seed of the random number generator. The argument `seed` must be in the range (0, 2147483646). If `seed` is 0, a value is computed using the system clock; hence, the results of programs using the random number generators will be different at various times.

Description

Function `imsls_random_seed_set` is used to initialize the seed used in the random number generators. The form of the generators is as follows:

$$x_i \equiv cx_{i-1} \bmod (2^{31} - 1)$$

The value of x_0 is the seed. If the seed is not initialized prior to invocation of any of the functions for random number generation by calling `imsls_random_seed_set`, the seed is initialized by the system clock. The seed can be reinitialized to a clock-dependent value by calling `imsls_random_seed_set` with `seed` set to 0.

The effect of `imsls_random_seed_set` is to set some global values used by the random number generators. A common use of `imsls_random_seed_set` is in conjunction with function `imsls_random_seed_get` to restart a simulation.

Example

See function `imsls_random_seed_get` (page 850).

random_table_set

Sets the current table used in the shuffled generator.

Synopsis

#include `<imsls.h>`

void `imsls_f_random_table_set` (*float* `table[]`)

The type *double* function is `imsls_d_random_table_set`.

Required Arguments

float `table[]` (Input)
Array of length 128 used in the shuffled generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive in except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_table_set`, on the next invocation of a routine to generate random numbers using a shuffled method , the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get` (page 853).

random_table_get

Retrieves the current table used in the shuffled generator.

Synopsis

#include `<imsls.h>`

void `imsls_f_random_table_get` (*float* `**table`, ..., 0)

The type *double* function is `imsls_d_random_table_get`.

Required Arguments

float `**table` (Output)
Address of a pointer to an array of length 128 containing the table used in the shuffled generators. Typically, *float* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_random_table_get` (*float* `**table`,
`IMSLS_RETURN_USER`, *float* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `r[]` (Output)
User-supplied array of length 1565 containing the table used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_table_set`, on the next invocation of a routine to generate random numbers using a shuffled method , the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get` (page 853).

random_GFSR_table_set

Sets the current table used in the GFSR generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_GFSR_table_set` (*int* `table[]`)

Required Arguments

int `table []` (Input)
: Array of length 1565 used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_GFSR_table_set`, on the next invocation of a routine to generate random numbers using a GFSR method , the appropriate array will be reinitialized.

Example

See function `imsls_random_GFSR_table_get` (page 853).

random_GFSR_table_get

Retrieves the current table used in the GFSR generator.

Synopsis

#include `<imsls.h>`

void `imsls_random_GFSR_table_get` (*int* `**table`, ..., 0)

Required Arguments

int `**table` (Output)
: Address of a pointer to an array of length 1565 containing the table used in the GFSR generators. Typically, *int* `*table` is declared and `&table` is used as an argument.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_random_GFSR_table_get` (*int* `**table`,
`IMSLS_RETURN_USER`, *int* `r[]`,
0)

Optional Arguments

`IMSLS_RETURN_USER`, *int* `r[]` (Output)
User-supplied array of length 1565 containing the table used in the GFSR generators.

Description

The values in `table` are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of `table` is set to a nonpositive value on the call to `imsls_random_GFSR_table_set`, on the next invocation of a routine to generate random numbers using a GFSR method, the appropriate array will be reinitialized.

Example

In this example, three separate simulation streams are used, each with a different form of the generator. Each stream is stopped and restarted. (Although this example is obviously an artificial one, there may be reasons for maintaining separate streams and stopping and restarting them because of the nature of the usage of the random numbers coming from the separate streams.)

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

void main()
{
  float *r, *table;
  int  nr, iseed1, iseed2, iseed7;
  int *itable;

  nr = 5;
  iseed1 = 123457;
  iseed2 = 123457;
  iseed7 = 123457;

  /* Begin first stream, iopt = 1 (by default) */
  imsls_random_seed_set (iseed1);
  r = imsls_f_random_uniform (nr, 0);
  iseed1 = imsls_random_seed_get ();
  imsls_f_write_matrix ("First stream output", 1, 5, r,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed1);
  free(r);
```

```
/* Begin second stream, iopt = 2 */
imsls_random_option (2);
imsls_random_seed_set (iseed2);
r = imsls_f_random_uniform (nr, 0);
iseed2 = imsls_random_seed_get ();
imsls_f_random_table_get (&table, 0);
imsls_f_write_matrix ("Second stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
printf("    Output seed\t%d\n\n", iseed2);
free(r);

/* Begin third stream, iopt = 7 */
imsls_random_option (7);
imsls_random_seed_set (iseed7);
r = imsls_f_random_uniform (nr, 0);
iseed7 = imsls_random_seed_get ();
imsls_random_GFSR_table_get (&itable, 0);
imsls_f_write_matrix ("Third stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
printf("    Output seed\t%d\n\n", iseed7);
free(r);

/* Reinitialize seed and resume first stream */
imsls_random_option (1);
imsls_random_seed_set (iseed1);
r = imsls_f_random_uniform (nr, 0);
iseed1 = imsls_random_seed_get ();
imsls_f_write_matrix ("First stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
printf("    Output seed\t%d\n\n", iseed1);
free(r);

/*
 * Reinitialize seed and table for shuffling and
 * resume second stream
 */
imsls_random_option (2);
imsls_random_seed_set (iseed2);
imsls_f_random_table_set (table);
r = imsls_f_random_uniform (nr, 0);
iseed2 = imsls_random_seed_get ();
imsls_f_write_matrix ("Second stream output", 1, 5, r,
                  IMSLS_NO_COL_LABELS,
                  IMSLS_NO_ROW_LABELS, 0);
printf("    Output seed\t%d\n\n", iseed2);
free(r);

/*
 * Reinitialize seed and table for GFSR and
 * resume third stream.
 */
imsls_random_option (7);
imsls_random_seed_set (iseed7);
imsls_random_GFSR_table_set (itable);
r = imsls_f_random_uniform (nr, 0);
```

```
  iseed7 = imsls_random_seed_get ();
  imsls_f_write_matrix ("Third stream output", 1, 5, r,
                    IMSLS_NO_COL_LABELS,
                    IMSLS_NO_ROW_LABELS, 0);
  printf("    Output seed\t%d\n\n", iseed7);
  free(r);

}
```

Output

```
                    First stream output
    0.9662      0.2607      0.7663      0.5693      0.8448
 Output seed 1814256879


                   Second stream output
    0.7095      0.1861      0.4794      0.6038      0.3790
 Output seed 1965912801


                    Third stream output
    0.3914      0.0263      0.7622      0.0281      0.8997
 Output seed 1932158269


                    First stream output
    0.0443      0.9872      0.6014      0.8964      0.3809
 Output seed 817878095


                   Second stream output
    0.2557      0.4788      0.2258      0.3455      0.5811
 Output seed 2108806573


                    Third stream output
    0.7519      0.5084      0.9070      0.0910      0.6917
 Output seed 1485334679
```

faure_next_point

Computes a shuffled Faure sequence.

Synopsis

#include `<imsls.h>`

*Imsls_faure** `imsls_faure_sequence_init` (*int* `ndim`, …, `0`)

*float** `imsls_f_faure_next_point` (*Imsls_faure* `*state`, …, `0`)

void `imsls_faure_sequence_free` (*Imsls_faure* `*state`)

The type *double* function is `imsls_d_faure_next_point`. The functions `imsls_faure_sequence_init` and `imsls_faure_sequence_free` are precision independent.

Required Arguments for imsls_faure_sequence_init

int `ndim` (Input)
The dimension of the hyper-rectangle.

Return Value for imsls_faure_sequence_init

Returns a structure that contains information about the sequence. The structure should be freed using `imsls_faure_sequence_free` after it is no longer needed.

Required Arguments for imsls_faure_next_point

Imsls_faure `*state` (Input/Output)
Structure created by a call to `imsls_faure_sequence_init`.

Return Value for imsls_faure_next_point

Returns the next point in the shuffled Faure sequence. To release this space, use `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>`

float `*imsls_faure_sequence_init` (*int* `ndim`,
        `IMSLS_BASE`, *int* `base`,
        `IMSLS_SKIP`, *int* `skip`,
        `0`)

*float** `imsls_f_faure_next_point` (*Imsls_faure* `*state`,
        `IMSLS_RETURN_USER`, *float* `*user`,
        `IMSLS_RETURN_SKIP`, *int* `*skip`,
        `0`)

Optional Arguments

`IMSLS_BASE`, *int* `base` (Input)
The base of the Faure sequence.
Default: The smallest prime greater than or equal to `ndim`.

`IMSLS_SKIP`, *int* `*skip` (Input)
The number of points to be skipped at the beginning of the Faure

sequence.

Default: $\lfloor \texttt{base}^{m/2-1} \rfloor$, where $m = \lfloor \log \mathrm{B} / \log \texttt{base} \rfloor$ and *B* is the largest representable integer.

`IMSLS_RETURN_USER`, *float* *user (Output)
User-supplied array of length `ndim` containing the current point in the sequence.

`IMSLS_RETURN_SKIP`, *int* *`skip` (Output)
The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for `IMSLS_SKIP`, and using the same dimension for `ndim`.

Description

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set $x_1, \ldots, x_n \in [0,1]^d$, $d \geq 1$, is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = [0, t_1) \times \ldots \times [0, t_d),\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d\ ,$$

λ is the Lebesque measure, and $A(E;n)$ is the number of the x_j contained in *E*.

The sequence $x_1, x_2, \ldots$ of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant *c*(*d*), depending only on *d*, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

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 `IMSLS_BASE` defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer *n* in its *b*-ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers, $0 \leq a_i(n) < b$.

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series, $c_{k\,d}^{(j)}$, is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and $c_{k\,d}$ is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!\left(d-c\right)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the b-ary Gray code. The function $G(n)$ maps the positive integer n into the integer given by its b-ary expansion.

The sequence computed by this function is $\boldsymbol{x}(G(n))$, where $\boldsymbol{x}$ is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that `imsls_faure_sequence_init` is used to create a structure that holds the state of the sequence. Each call to `imsls_f_faure_next_point` returns the next point in the sequence and updates the *Imsls_faure* structure. The final call to `imsls_faure_sequence_free` frees data items, stored in the structure, that were allocated by `imsls_faure_sequence_init`.

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


void main()
{
       Imsl_faure    *state;
       float         *x;
       int            ndim = 3;
       int            k;

       state = imsl_faure_sequence_init(ndim, 0);

       for (k = 0;  k < 5;  k++) {
             x = imsl_f_faure_next_point(state, 0);
```

```
                printf("%10.3f %10.3f  %10.3f\n", x[0], x[1], x[2]);
               free(x);
        }

        imsl_faure_sequence_free(state);
}
```

Output

```
     0.334      0.493       0.064
     0.667      0.826       0.397
     0.778      0.270       0.175
     0.111      0.604       0.509
     0.445      0.937       0.842
```

Chapter 13: Printing Functions

Routines

write_matrix

Prints a rectangular matrix (or vector) stored in contiguous memory locations.

Synopsis

#include `<imsls.h>`

void `imsls_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`, …, 0)

For *int* `a[]`, use `imsls_i_write_matrix`.
For *double* `a[]`, use `imsls_d_write_matrix`.

Required Arguments

char `*title` (Input)
Matrix title. Use `\n` within a title to create a new line. Long titles are automatically wrapped.

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

int `nca` (Input)
Number of columns in the matrix.

float `a[]` (Input)
Array of size `nra` × `nca` containing the matrix to be printed.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_f_write_matrix` (*char* `*title`, *int* `nra`, *int* `nca`, *float* `a[]`,
`IMSLS_TRANSPOSE`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,
`IMSLS_PRINT_ALL`, *or*
`IMSLS_PRINT_LOWER`, *or*
`IMSLS_PRINT_UPPER`, *or*
`IMSLS_PRINT_LOWER_NO_DIAG`, *or*
`IMSLS_PRINT_UPPER_NO_DIAG`,
`IMSLS_WRITE_FORMAT`, *char* `*fmt`,
`IMSLS_NO_ROW_LABELS`, *or*
`IMSLS_ROW_NUMBER`, *or*
`IMSLS_ROW_NUMBER_ZERO`, *or*
`IMSLS_ROW_LABELS`, *char* `*rlabel[]`,
`IMSLS_NO_COL_LABELS`, *or*
`IMSLS_COL_NUMBER`, *or*
`IMSLS_COL_NUMBER_ZERO`, *or*
`IMSLS_COL_LABELS`, *char* `*clabel[]`,
0)

Optional Arguments

`IMSLS_TRANSPOSE`
Print a^T.

`IMSLS_A_COL_DIM`, *int* `a_col_dim` (Input)
Column dimension of *a*.
Default: `a_col_dim` = `nca`

`IMSLS_PRINT_ALL`, *or*
`IMSLS_PRINT_LOWER`, *or*
`IMSLS_PRINT_UPPER`, *or*
`IMSLS_PRINT_LOWER_NO_DIAG`, *or*
`IMSLS_PRINT_UPPER_NO_DIAG`
Exactly one of these optional arguments can be specified to indicate that either a triangular part of the matrix or the entire matrix is to be printed. If omitted, the entire matrix is printed.

Keyword	Action
`IMSLS_PRINT_ALL`	Entire matrix is printed (the default).
`IMSLS_PRINT_LOWER`	Lower triangle of the matrix is printed, including the diagonal.
`IMSLS_PRINT_UPPER`	Upper triangle of the matrix is printed, including the diagonal.

Keyword	Action
`IMSLS_PRINT_LOWER_NO_DIAG`	Lower triangle of the matrix is printed, without the diagonal.
`IMSLS_PRINT_UPPER_NO_DIAG`	Upper triangle of the matrix is printed, without the diagonal.

`IMSLS_WRITE_FORMAT`, *char* `*fmt` (Input)

Character string containing a list of C conversion specifications (formats) to be used when printing the matrix. Any list of C conversion specifications suitable for the data type can be given. For example, `fmt` = `"%10.3f"` specifies the conversion character `f` for the entire matrix. For the conversion character `f`, the matrix must be of type *float* or *double*. Alternatively, `fmt` = `"%10.3e%10.3e%10.3f%10.3f%10.3f"` specifies the conversion character `e` for columns 1 and 2 and the conversion character `f` for columns 3, 4, and 5. If the end of `fmt` is encountered and if some columns of the matrix remain, format control continues with the first conversion specification in `fmt`.

Aside from restarting the format from the beginning, other exceptions to the usual C formatting rules are as follows:

1. Characters not associated with a conversion specification are not allowed. For example, in the format `fmt` = `"1%d2%d"`, the characters `1` and `2` are not allowed and result in an error.

2. A conversion character `d` can be used for floating-point values (matrices of type *float* or *double*). The integer part of the floating-point value is printed.

3. For printing numbers whose magnitudes are unknown, the conversion character `g` is useful; however, the decimal points will generally not be aligned when printing a column of numbers. The `w` (or `W`) conversion character is a special conversion character used by this function to select a conversion specification so that the decimal points will be aligned. The conversion specification ending with `w` is specified as `"%n.dw"`. Here, `n` is the field width and `d` is the number of significant digits generally printed. Valid values for `n` are 3, 4, …, 40. Valid values for `d` are 1, 2, …, `n` – 2. If `fmt` specifies one conversion specification ending with `w`, all elements of `a` are examined to determine one conversion specification for printing. If `fmt` specifies more than one conversion specification, separate conversion specifications are generated for each conversion specification ending with `w`. Set `fmt` = `"10.4w"` for a single conversion specification selected automatically with field width 10 and with four significant digits.

`IMSLS_NO_ROW_LABELS`, *or*
`IMSLS_ROW_NUMBER`, *or*
`IMSLS_ROW_NUMBER_ZERO`, *or*

`IMSLS_ROW_LABELS`, *char* `*rlabel[]` (Input)

If `IMSLS_ROW_LABELS` is specified, `rlabel` is a vector of length `nra` containing pointers to the character strings comprising the row labels. Here, `nra` is the number of rows in the printed matrix. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no row labels are desired, use the `IMSLS_NO_ROW_LABELS` optional argument. If the numbers 1, 2, …, `nra` are desired, use the `IMSLS_ROW_NUMBER` optional argument. If the numbers 0, 1, 2, …, `nra` – 1 are desired, use the `IMSLS_ROW_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nra` are used for the row labels by default whenever `nra` > 1. If `nra` = 1, the default is no row labels.

`IMSLS_NO_COL_LABELS`, *or*
`IMSLS_COL_NUMBER`, *or*
`IMSLS_COL_NUMBER_ZERO`, *or*
`IMSLS_COL_LABELS`, *char* `*clabel[]` (Input)

If `IMSLS_COL_LABELS` is specified, `clabel` is a vector of length `nca` + 1 containing pointers to the character strings comprising the column headings. The heading for the row labels is `clabel` [0]; `clabel` [*i*], *i* = 1, …, `nca`, is the heading for the *i*-th column. Use `\n` within a label to create a new line. Long labels are automatically wrapped. If no column labels are desired, use the `IMSLS_NO_COL_LABELS` optional argument. If the numbers 1, 2, …, `nca`, are desired, use the `IMSLS_COL_NUMBER` optional argument. If the numbers 0, 1, …, `nca` – 1 are desired, use the `IMSLS_COL_NUMBER_ZERO` optional argument. If none of these optional arguments is used, the numbers 1, 2, 3, …, `nca` are used for the column labels by default whenever `nca` > 1. If `nca` = 1, the default is no column labels.

Description

Function `imsls_write_matrix` prints a real rectangular matrix (stored in *a*) with optional row and column labels (specified by `rlabel` and `clabel`, respectively, regardless of whether a or a^T is printed). An optional format, `fmt`, can be used to specify a conversion specification for each column of the matrix.

In addition, the write matrix functions can restrict printing to the elements of the upper or lower triangles of a matrix by using the `IMSLS_PRINT_UPPER`, `IMSLS_PRINT_LOWER`, `IMSLS_PRINT_UPPER_NO_DIAG`, and `IMSLS_PRINT_LOWER_NO_DIAG` options. Generally, these options are used with symmetric matrices, but this is not required. Vectors can be printed by specifying a row or column dimension of 1.

Output is written to the file specified by the function `imsls_output_file` (Chapter 14, "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` (page 867).

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` (page 868).

Examples

Example 1

This example is representative of the most common situation in which no optional arguments are given.

```
#include <imsls.h>

#define NRA 3
#define NCA 4

main()
{
    int     i, j;
    float   a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }

    }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, (float*) a, 0);
}
```

Output

```
                    matrix
                       a
          1            2            3            4
1       1.1          1.2          1.3          1.4
2       2.1          2.2          2.3          2.4
3       3.1          3.2          3.3          3.4
```

Example 2

In this example, some of the optional arguments available in the `imsls_write_matrix` functions are demonstrated.

```
#include <imsls.h>

#define NRA     3
#define NCA     4

main()
{
    int         i, j;
    float       a[NRA][NCA];
    char        *fmt = "%10.6W";
    char        *rlabel[] = {"row 1", "row 2", "row 3"};
    char        *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1+(j+1)*0.1);
        }
    }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, (float *)a,
        IMSLS_WRITE_FORMAT, fmt,
        IMSLS_ROW_LABELS, rlabel,
        IMSLS_COL_LABELS, clabel,
        IMSLS_PRINT_UPPER_NO_DIAG,
        0);
}
```

Output

```
                        matrix
                           a
            col 2       col 3       col 4
row 1         1.2         1.3         1.4
row 2                     2.3         2.4
row 3                                 3.4
```

Example 3

In this example, a row vector of length four is printed.

```
#include <imsls.h>

#define NRA 1
#define NCA 4

main()
{
    int         i;
    float       a[NCA];
    char        *clabel[] = {"", "col 1", "col 2", "col 3", "col 4"};

    for (i = 0; i < NCA; i++) {
    a[i] = i + 1;
   }
                                /* Write matrix */
    imsls_f_write_matrix ("matrix\na", NRA, NCA, a,
        IMSLS_COL_LABELS, clabel,
        0);
}
```

Output

```
                     matrix
                        a
     col 1       col 2       col 3       col 4
         1           2           3           4
```

page

Sets or retrieves the page width or length.

Synopsis

#include `<imsls.h>`

void `imsls_page` (*Imsls_page_options* `option`, *int* `*page_attribute`)

Required Arguments

Imsls_page_options `option` (Input)
Option giving which page attribute is to be set or retrieved. The possible values are shown in the table below.

Keyword	Description
`IMSLS_SET_PAGE_WIDTH`	Sets the page width.
`IMSLS_GET_PAGE_WIDTH`	Retrieves the page width.
`IMSLS_SET_PAGE_LENGTH`	Sets the page length.
`IMSLS_GET_PAGE_LENGTH`	Retrieves the page length.

int `*page_attribute` (Input, if the attribute is set; Output, otherwise.)
The value of the page attribute to be set or retrieved. The page width is the number of characters per line of output (default 78), and the page length is the number of lines of output per page (default 60). Ten or more characters per line and 10 or more lines per page are required.

Example

The following example illustrates the use of `imsls_page` to set the page width to 40 characters. Function `imsls_f_write_matrix` is then used to print a 3×4 matrix A, where $a_{ij} = i + j/10$.

```
#include <imsls.h>

#define NRA 3
#define NCA 4
main()
{
    int         i, j, page_attribute;
    float       a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
    page_attribute = 40;
    imsls_page(IMSLS_SET_PAGE_WIDTH, &page_attribute);
    imsls_f_write_matrix("a", NRA, NCA, (float *)a, 0);
}
```

Output

```
                    a
            1           2           3
1         1.1         1.2         1.3
2         2.1         2.2         2.3
3         3.1         3.2         3.3

            4
1         1.4
2         2.4
3         3.4
```

write_options

Sets or retrieves an option for printing a matrix.

Synopsis

#include `<imsls.h>`

void `imsls_write_options` (*Imsls_write_options* `option`,
    int `*option_value`)

Required Arguments

Imsls_write_options `option` (Input)
Option giving the type of the printing attribute to set or retrieve.

Keyword for Setting	Keyword for Retrieving	Attribute Description
`IMSLS_SET_DEFAULTS`		uses the default settings for all parameters
`IMSLS_SET_CENTERING`	`IMSLS_GET_CENTERING`	horizontal centering
`IMSLS_SET_ROW_WRAP`	`IMSLS_GET_ROW_WRAP`	row wrapping
`IMSLS_SET_PAGING`	`IMSLS_GET_PAGING`	paging
`IMSLS_SET_NAN_CHAR`	`IMSLS_GET_NAN_CHAR`	method for printing NaN
`IMSLS_SET_TITLE_PAGE`	`IMSLS_GET_TITLE_PAGE`	whether or not titles appear on each page
`IMSLS_SET_FORMAT`	`IMSLS_GET_FORMAT`	default format for real and complex numbers

int `*option_value` (Input, if `option` is to be set; Output, otherwise)
Value of the option attribute selected by `option`. The values to be used when setting attributes are described in a table in the description section.

Description

Function `imsls_write_options` allows the user to set or retrieve an option for printing a matrix. Options controlled by `imsls_write_options` are horizontal centering, method for printing large matrices, paging, method for printing NaN, method for printing titles, and the default format for real and complex numbers. (NaN can be retrieved by functions `imsls_f_machine` and `imsls_d_machine` (Chapter 14, "Utilities").

The following values can be used for the attributes:

Keyword	Value	Meaning
`CENTERING`	0	Matrix is left justified.
	1	Matrix is centered.
`ROW_WRAP`	0	Complete row is printed before the next row is printed. Wrapping is used if necessary.
	m	Here, *m* is a positive integer. Let n_1 be the maximum number of columns that fit across the page, as determined by the widths in the conversion specifications starting with column 1. First, columns 1 through n_1 are printed for rows 1 through *m*. Let n_2 be the maximum number of columns that fit across the page, starting with column n_1+1. Second, columns n_1+1 through n_1+n_2 are printed for rows 1 through *m*. This continues until the last columns are printed for rows 1 through *m*. Printing continues in this fashion for the next *m* rows, etc.

Keyword	Value	Meaning
PAGING	–2	No paging occurs.
	–1	Paging is on. Every invocation of an function imsls_write_matrix begins on a new page, and paging occurs within each invocation as is needed.
	0	Paging is on. The first invocation of an imsls_f_write_f_matrix function begins on a new page, and subsequent paging occurs as is needed. Paging occurs in the second and all subsequent calls to an imsls_f_write_matrix function only as needed.
	k	Turn paging on and set the number of lines printed on the current page to *k* lines. If *k* is greater than or equal to the page length, then the first invocation of an imsls_write_matrix function begins on a new page. In any case, subsequent paging occurs as is needed.
NAN_CHAR	0	. . . . is printed for NaN.
	1	A blank field is printed for NaN.
TITLE_PAGE	0	Title appears only on first page.
	1	Title appears on the first page and all continuation pages.
FORMAT	0	Format is "%10.4x".
	1	Format is "%12.6w".
	2	Format is "%22.5e".

The w conversion character used by the FORMAT option is a special conversion character that can be used to automatically select a pretty C conversion specification ending in either e, f, or d. The conversion specification ending with w is specified as "%n.dw". Here, n is the field width, and d is the number of significant digits generally printed.

Function imsls_write_options can be invoked repeatedly before using a function imsls_f_write_matrix to print a matrix. The matrix printing functions retrieve the values set by imsls_write_options to determine the printing options. It is not necessary to call imsls_write_options if a default value of a printing option is desired. The defaults are as follows:

Keyword	Default Value	Meaning
`CENTERING`	0	left justified
`ROW_WRAP`	1000	lines before wrapping
`PAGING`	–2	no paging
`NAN_CHAR`	0	. . . .
`TITLE_PAGE`	0	title appears only on the first page
`FORMAT`	0	%10.4w

Example

The following example illustrates the effect of `imsls_write_options` when printing a 3 × 4 real matrix *A* with function `imsls_f_write_matrix`, where $a_{ij} = i + j/10$. The first call to `imsls_f_write_options` sets horizontal centering so that the matrix is printed centered horizontally on the page. In the next invocation of `imsls_f_write_matrix`, the left-justification option has been set by function `imsls_write_options` so the matrix is left justified when printed.

```
#include <imsls.h>

#define NRA 4
#define NCA 3

main()
{
    int         i, j, option_value;
    float       a[NRA][NCA];

    for (i = 0; i < NRA; i++) {
        for (j = 0; j < NCA; j++) {
            a[i][j] = (i+1) + (j+1)/10.0;
        }
    }
                                /* Activate centering option */
    option_value = 1;
    imsls_write_options (IMSLS_SET_CENTERING, &option_value);
                                /* Write a matrix */
    imsls_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
                                /* Activate left justification */
    option_value = 0;
    imsls_write_options (IMSLS_SET_CENTERING, &option_value);
    imsls_f_write_matrix ("a", NRA, NCA, (float*) a, 0);
}
```

Output

```
                                   a
                            1             2             3
              1           1.1           1.2           1.3
              2           2.1           2.2           2.3
              3           3.1           3.2           3.3
              4           4.1           4.2           4.3

                       a
              1             2             3
1           1.1           1.2           1.3
2           2.1           2.2           2.3
3           3.1           3.2           3.3
4           4.1           4.2           4.3
```

Chapter 14: Utilities

Routines

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

main()
{
    FILE         *ofile;
    float       x[] = {3.0, 2.0, 1.0};
    imsls_f_write_matrix ("x (default file)", 1, 3, x, 0);

    ofile = fopen("myfile", "w");
    imsls_output_file(IMSLS_SET_OUTPUT_FILE, ofile,
                     0);
    imsls_f_write_matrix ("x (myfile)", 1, 3, x, 0);
}
```

Output

```
         x (default file)
         1           2           3
         3           2           1
```

File myfile

```
x (myfile)
1           2           3
3           2           1
```

Example 2

The following example illustrates how to direct output from IMSL routines that run in separate threads to different files. First, two threads are created, each calling a different IMSL function, then the results are printed by calling `imsls_f_write_matrix` from within each thread. Note that `imsls_output_file` is called from within each thread to change the default output file.

```
#include <pthread.h>

#include <stdio.h>

#include "imsls.h"

void *ex1(void* arg);

void *ex2(void* arg);

void main()

{

  pthread_t       thread1;

  pthread_t       thread2;


  /* Disable IMSL signal trapping. */

  imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);


  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}
void *ex1(void* arg)
{
  float *rand_nums = NULL;
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex1.out", "w");
  /* Set the output file for this thread. */
  imsls_output_file(IMSLS_SET_OUTPUT_FILE, file_ptr, 0);
  /* Compute 5 random numbers. */
  imsls_random_seed_set(12345);
  rand_nums = imsls_f_random_uniform(5, 0);
  /* Output random numbers. */
  imsls_f_write_matrix("Random Numbers", 5, 1, rand_nums, 0);
  if (rand_nums) free(rand_nums);
  fclose(file_ptr);
}
void *ex2(void* arg)
{
  int n_intervals=10;
  int n_observations=30;
  float *table;
  float x[] = {0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
             2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
             0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
             1.89, 0.90, 2.05};
  FILE  *file_ptr;
  /* Open a file to write the result in. */
  file_ptr = fopen("ex2.out", "w");
  /* Set the output file for this thread. */
  imsls_output_file(IMSLS_SET_OUTPUT_FILE, file_ptr, 0);
  table = imsls_f_table_oneway (n_observations, x, n_intervals, 0);
  imsls_f_write_matrix("counts", 1, n_intervals, table, 0);

  if (table) free(table);
  fclose(file_ptr);
}
```

ex1.out

```
Random Numbers
 1      0.4919
 2      0.3909
 3      0.2645
 4      0.1814
 5      0.7546
```

ex2.out

```
                                counts
        1           2           3           4           5           6
        4           8           5           5           3           1

        7           8           9          10
        3           0           0           1
```

version

Returns information describing the version of the library, serial number, operating system, and compiler.

Synopsis

#include `<imsls.h>`

char `*imsls_version` (*Imsls_keyword* `code`)

Required Arguments

Imsls_keyword `code` (Input)
Index indicating which value is to be returned. It must be `IMSLS_LIBRARY_VERSION`, `IMSLS_OS_VERSION`, `IMSLS_COMPILER_VERSION`, or `IMSLS_LICENSE_NUMBER`.

Return Value

The requested value is returned. If `code` is out of range, then `NULL` is returned. Use `free` to release the returned string.

Description

Function `imsls_version` returns information describing the version of the library, the version of the operating system under which it was compiled, the compiler used, and the IMSL serial number.

Example

This example prints all the values returned by `imsls_version` on a particular machine. The output is omitted because the results are system dependent.

```
#include <imsls.h>

main()
{
    char    *library_version, *os_version;
    char    *compiler_version, *license_number;

    library_version  = imsls_version(IMSLS_LIBRARY_VERSION);
    os_version       = imsls_version(IMSLS_OS_VERSION);
    compiler_version = imsls_version(IMSLS_COMPILER_VERSION);
    license_number   = imsls_version(IMSLS_LICENSE_NUMBER);

    printf("Library version = %s\n", library_version);
    printf("OS version = %s\n", os_version);
    printf("Compiler version = %s\n", compiler_version);
    printf("Serial number = %s\n", license_number);
}
```

error_options

Sets various error handling options.

Synopsis with Optional Arguments

#include `<imsls.h>`

void `imsls_error_options` (
    `IMSLS_SET_PRINT`, *Imsls_error* `type`, *int* `setting`,
    `IMSLS_SET_STOP`, *Imsls_error* `type`, *int* `setting`,
    `IMSLS_SET_TRACEBACK`, *Imsls_error* `type`, *int* `setting`,
    `IMSLS_FULL_TRACEBACK`, *int* `setting`,
    `IMSLS_GET_PRINT`, *Imsls_error* type, *int* `*psetting`,
    `IMSLS_GET_STOP`, *Imsls_error* `type`, *int* `*psetting`,
    `IMSLS_GET_TRACEBACK`, *Imsls_error* `type`, *int* `*psetting`,
    `IMSLS_SET_ERROR_FILE`, *FILE* `*file`,
    `IMSLS_GET_ERROR_FILE`, *FILE* `**pfile`,
    `IMSLS_ERROR_MSG_PATH`, *char* `*path`,
    `IMSLS_ERROR_MSG_NAME`, *char* `*name`,
    `IMSLS_ERROR_PRINT_PROC`, *Imsls_error_print_proc* `print_proc`,
    `IMSLS_SET_SIGNAL_TRAPPING`, *int* `setting`,
    0)

Optional Arguments

`IMSLS_SET_PRINT`, *Imsls_error* type, *int* `setting` (Input)
Printing of type `type` error messages is turned off if `setting` is 0; otherwise, printing is turned on.
Default: Printing turned on for `IMSLS_WARNING`, `IMSLS_FATAL`, `IMSLS_TERMINAL`, `IMSLS_FATAL_IMMEDIATE`, `and` `IMSLS_WARNING_IMMEDIATE` messages

`IMSLS_SET_STOP`, *Imsls_error* `type`, *int* `setting` (Input)
Stopping on type `type` error messages is turned off if `setting` is 0; otherwise, stopping is turned on.
Default: Stopping turned on for `IMSLS_FATAL` and `IMSLS_TERMINAL` and `IMSLS_FATAL_IMMEDIATE` messages

`IMSLS_SET_TRACEBACK`, *Imsls_error* `type`, *int* `setting` (Input)
Printing of a traceback on type `type` error messages is turned off if `setting` is 0; otherwise, printing of the traceback turned on.
Default: Traceback turned off for all message types

`IMSLS_FULL_TRACEBACK`, *int* `setting` (Input)
Only documented functions are listed in the traceback if `setting` is 0; otherwise, internal function names also are listed.
Default: Full traceback turned off

`IMSLS_GET_PRINT`, *Imsls_error* `type`, *int* `*psetting` (Output)
: Sets the integer pointed to by `psetting` to the current setting for printing of type type error messages.

`IMSLS_GET_STOP`, *Imsls_error* `type`, *int* `*psetting` (Output)
: Sets the integer pointed to by `psetting` to the current setting for stopping on type type error messages.

`IMSLS_GET_TRACEBACK`, *Imsls_error* type, *int* `*psetting` (Output)
: Sets the integer pointed to by `psetting` to the current setting for printing of a traceback for type `type` error messages.

`IMSLS_SET_ERROR_FILE`, *FILE* `*file` (Input)
: Sets the error output file.
Default: `file` = `stderr`

`IMSLS_GET_ERROR_FILE`, *FILE* `**pfile` (Output)
: Sets the *FILE* `*` pointed to by `pfile` to the error output file.

`IMSLS_ERROR_MSG_PATH`, *char* `*path` (Input)
: Sets the error message file path. On UNIX systems, this is a colon-separated list of directories to be searched for the file containing the error messages.
Default: system dependent

`IMSLS_ERROR_MSG_NAME`, *char* `*name` (Input)
: Sets the name of the file containing the error messages.
Default: `file` = `"imsls_e.bin"`

`IMSLS_ERROR_PRINT_PROC`, *Imsls_error_print_proc* `print_proc` (Input)
: Sets the error printing function. The procedure `print_proc` has the form *void* `print_proc` (*Imsls_error* `type`, *long* `code`, *char* `*function_name`, *char* `*message`).

  In this case, `type` is the error message type number (`IMSLS_FATAL`, etc.), code is the error message code number (`IMSLS_MAJOR_VIOLATION`, etc.), `function_name` is the name of the function setting the error, and message is the error message to be printed. If `print_proc` is `NULL`, then the default error printing function is used.

`IMSLS_SET_SIGNAL_TRAPPING`, *int* `setting` (Input)
: C/Stat/Library will use its own signal handler if setting is 1; otherwise the C/Stat/Library signal handler is not used. If C/Stat/Library is called from a multi-threaded application, signal handling by C/Stat/Library must be turned off. See Example 3 for details.

  Default: setting = 1

Return Value

The return value is void.

Description

This function allows the error handling system to be customized.

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options (excluding `IMSLS_SET_SIGNAL_TRAPPING`) for each thread by calling `imsls_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);
```

See Example 3 and Example 4 for multithreaded examples.

Examples

Example 1

In this example, the `IMSLS_TERMINAL` print setting is retrieved. Next, stopping on `IMSLS_TERMINAL` errors is turned off, output to standard output is redirected, and an error is deliberately caused by calling `imsls_error_options` with an illegal value.

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

main()
{
    int         setting;
                              /* Turn off stopping on IMSLS_TERMINAL */
                              /* error messages and write error */
                              /* messages to standard output */
    imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                       IMSLS_SET_ERROR_FILE, stdout,
                       0);
                              /* Call imsls_error_options() with */
                              /* an illegal value */
    imsls_error_options(-1);
                              /* Get setting for IMSLS_TERMINAL */
    imsls_error_options(IMSLS_GET_PRINT, IMSLS_TERMINAL, &setting,
                       0);
    printf("IMSLS_TERMINAL error print setting = %d\n", setting);
}
```

Output

```
*** TERMINAL Error from imsls_error_options.  There is an error with
*** argument number 1.  This may be caused by an incorrect number of
*** values following a previous optional argument name.

IMSLS_TERMINAL error print setting = 1
```

Example 2

In this example, IMSL's error printing function has been substituted for the standard function. Only the first four lines are printed below.

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

void         print_proc(Imsls_error, long, char*, char*);

main()
{
                            /* Turn off tracebacks on IMSLS_TERMINAL */
                            /* error messages and use a custom */
                            /* print function */
    imsls_error_options(IMSLS_ERROR_PRINT_PROC, print_proc,
                       0);
                            /* Call imsls_error_options() with an */
                            /* illegal value */
    imsls_error_options(-1);
}

void print_proc(Imsls_error type, long code, char *function_name,
                char *message)
{
    printf("Error message type %d\n", type);
    printf("Error code %d\n", code);
    printf("From function %s\n", function_name);
    printf("%s\n", message);
}
```

Output

```
Error message type 5
Error code 103
From function imsls_error_options
There is an error with argument number 1.  This may be caused by an
incorrect number of values following a previous optional argument name.
```

Example 3

In this example, two threads are created and error options is called within each thread to set the error handling options slightly different for each thread. Since we expect to generate terminal errors in each thread, we must turn off stopping on terminal errors for each thread. Also notice that `imsls_error_options` is called from `main` to disable the IMSL signal-trapping mechanism.
See Example 4 for a similar example, using WIN32 threads. Note since multiple threads are executing, the order of the errors output may differ on some systems.

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

void *ex1(void* arg);
void *ex2(void* arg);
void main()
```

```
{
  pthread_t       thread1;
  pthread_t       thread2;

  /* Disable IMSL signal trapping. */
  imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);

  /* Create two threads. */
  if (pthread_create(&thread1, NULL ,ex1, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);
  if (pthread_create(&thread2, NULL ,ex2, (void *)NULL) != 0)
    perror("pthread_create"), exit(1);

  /* Wait for threads to finish. */
  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);
  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

}

void *ex1(void* arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0, 0);
  res = imsls_f_beta(-1.0, .5);
}
void *ex2(void* arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.  Notice that tracebacks are
   * turned on for IMSLS_TERMINAL errors.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                      IMSLS_SET_TRACEBACK, IMSLS_TERMINAL, 1, 0);
  res = imsls_f_gamma(-1.0);
}
```

Output

```
*** TERMINAL Error from imsls_f_beta.  Both "x" = -1.000000e+00 and "y" =
***          5.000000e-01 must be greater than zero.


*** TERMINAL Error from imsls_f_gamma.  The argument for the function can
***          not be a negative integer. Argument "x" = -1.000000e+00.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code              Routine
  ----------        ----------              -------
 IMSLS_TERMINAL    IMSLS_NEGATIVE_INTEGER    imsls_f_gamma
```

Example 4

In this example the WIN32 API is used to demonstrate the same functionality as shown in Example 3 above. Note since multiple threads are executing, the order of the errors output may differ on some systems.

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

DWORD WINAPI ex1(void *arg);
DWORD WINAPI ex2(void *arg);

int main(int argc, char* argv[])
{
       HANDLE thread[2];

       imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);

       thread[0] = CreateThread(NULL, 0, ex1, NULL, 0, NULL);
       thread[1] = CreateThread(NULL, 0, ex2, NULL, 0, NULL);

       WaitForMultipleObjects(2, thread, TRUE, INFINITE);

}
DWORD WINAPI ex1(void *arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.
   */
imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                   0);
  res = imsls_f_beta(-1.0, .5);
       return(0);
}
DWORD WINAPI ex2(void *arg)
{
  float res;
  /*
   * Call imsls_error_options to set the error handling
   * options for this thread.  Notice that tracebacks are
   * turned on for IMSLS_TERMINAL errors.
   */
  imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                   IMSLS_SET_TRACEBACK, IMSLS_TERMINAL, 1,
                   0);
  res = imsls_f_gamma(-1.0);
  return(0);
}
```

Output

```
*** TERMINAL Error from imsls_f_beta.  Both "x" = -1.000000e+000 and "y" =
***          5.000000e-001 must be greater than zero.


*** TERMINAL Error from imsls_f_gamma.  The argument for the function can
***          not be a negative integer. Argument "x" = -1.000000e+000.

Here is a traceback of the calls in reverse order.
  Error Type        Error Code               Routine
  ----------        ----------               -------
 IMSLS_TERMINAL    IMSLS_NEGATIVE_INTEGER    imsls_f_gamma USER
```

error_code

Gets the code corresponding to the error message from the last function called.

Synopsis

#include `<imsls.h>`

long `imsls_error_code` ( )

Return Value

This function returns the error message code from the last function called. The include file *imsls.h* defines a name for each error code.

Example

In this example, stopping on `IMSLS_TERMINAL` error messages is turned off and an error is then generated by calling function `imsls_error_options` with an illegal value for `IMSLS_SET_PRINT`. The error message code number is then retrieved and printed. In *imsls.h*, `IMSLS_INTEGER_OUT_OF_RANGE` is defined to be 132.

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

main()
{
    long        code;
                                /* Turn off stopping IMSLS_TERMINAL */
                                /* messages and print error messages */
                                /* on standard output */
    imsls_error_options(IMSLS_SET_STOP, IMSLS_TERMINAL, 0,
                       IMSLS_SET_ERROR_FILE, stdout,
                       0);
                                /* Call imsls_error_options() with */
                                /* an illegal value */
    imsls_error_options(IMSLS_SET_PRINT, 100, 0,
                       0);
                                /* Get the error message code */
    code = imsls_error_code();
    printf("error code = %d\n", code);
}
```

Output

```
*** TERMINAL error from imsls_error_options.  "type" must be between 1 and
***          5, but "type" = 100.

error code = 132
```

machine (integer)

Returns integer information describing the computer's arithmetic.

Synopsis

#include `<imsls.h>`

int `imsls_i_machine` (*int* `n`)

Required Arguments

int `n` (Input)
Index indicating which value is to be returned. It must be between 0 and 12.

Return Value

The requested value is returned. If `n` is out of range, NaN is returned.

Description

Function `imsls_i_machine` returns information describing the computer's arithmetic. This can be used to make programs machine independent.

`imsls_i_machine`(0) = Number of bits per byte

Assume that integers are represented in *M*-digit, base-*A* form as

$$\sigma \sum_{k=0}^{M} x_k \; A^k$$

where σ is the sign and $0 \le x_k < A$ for $k = 0, \ldots, M$. Then,

n	Definition
0	*C*, bits per character
1	*A*, the base
2	M_s, the number of base-*A* digits in a *short int*
3	$A^{M_s} - 1$, the largest *short int*

n	Definition
4	M_l, the number of base-A digits in a *long int*
5	$A^{M_l} - 1$, the largest *long int*

Assume that floating-point numbers are represented in N-digit, base B form as

$$\sigma B^E \sum_{k-1}^{N} x_k B^{-k}$$

where σ is the sign and $0 \le x_k < B$ for $k = 1, \ldots, N$ and $E_{\min} \le E \le E_{\max}$. Then

n	Definition
6	B, the base
7	N_f, the number of base-B digits in *float*
8	$E_{\min_f}$, the smallest *float* exponent
9	$E_{\max_f}$, the largest *float* exponent
10	N_d, the number of base-B digits in *double*
11	$E_{\min_d}$, the largest *long int*
12	$E_{\max_d}$, the number of base-B digits in *double*

Example

In this example, all the values returned by `imsls_i_machine` on a machine with IEEE (Institute for Electrical and Electronics Engineer) arithmetic are printed.

```
#include <imsls.h>

main()
{
    int         n, ans;

    for (n = 0;  n <= 12;  n++) {
        ans = imsls_i_machine(n);
        printf("imsls_i_machine(%d) = %d\n", n, ans);
    }
}
```

Output

```
imsls_i_machine(0) = 8
imsls_i_machine(1) = 2
imsls_i_machine(2) = 15
imsls_i_machine(3) = 32767
imsls_i_machine(4) = 31
imsls_i_machine(5) = 2147483647
imsls_i_machine(6) = 2
imsls_i_machine(7) = 24
imsls_i_machine(8) = -125
imsls_i_machine(9) = 128
imsls_i_machine(10) = 53
imsls_i_machine(11) = -1021
imsls_i_machine(12) = 1024
```

machine (float)

Returns information describing the computer's floating-point arithmetic.

Synopsis

#include `<imsls.h>`

float `imsls_f_machine` (*int* `n`)

The type *double* function is `imsls_d_machine`.

Required Arguments

int `n` (Input)
Index indicating which value is to be returned. The index must be between 1 and 8.

Return Value

The requested value is returned. If `n` is out of range, NaN is returned.

Description

Function `imsls_f_machine` returns information describing the computer's floating-point arithmetic. This can be used to make programs machine independent. In addition, some of the functions are also important in setting missing values.

Assume that *float* numbers are represented in N_f-digit, base B form as

$$\sigma B^E \sum_{k=1}^{N_f} x_k B^{-k}$$

where σ is the sign; $0 \le x_k < B$ for $k = 1, 2, \ldots, N_f$; and

$$E_{\min_f} \le E \le E_{\max_f}$$

Note that B = `imsls_i_machine`(6); N_f = `imsls_i_machine`(7);

$$E_{\min_f} = \texttt{imsls\_i\_machine}(8)$$

and

$$E_{\max_f} = \texttt{imsls\_i\_machine}(9)$$

The ANSI/IEEE 754-1985 standard for binary arithmetic uses NaN as the result of various otherwise illegal operations, such as computing 0/0. On computers that do not support NaN, a value larger than `imsls_d_machine`(2) is returned for `imsls_f_machine`(6). On computers that do not have a special representation for infinity, `imsls_f_machine`(2) returns the same value as `imsls_f_machine`(7).

Function `imsls_f_machine` is defined by the following table:

n	Definition
1	$B^{E_{\min_f}-1}$, the smallest positive number
2	$B^{E_{\max_f}}(1-B^{-N_f})$, the largest number
3	B^{-N_f}, the smallest relative spacing
4	B^{1-N_f}, the largest relative spacing
5	$\log_{10}(B)$
6	NaN
7	positive machine infinity
8	negative machine infinity

Function `imsls_d_machine` retrieves machine constants that define the computer's double arithmetic. Note that for *double B* = `imsls_i_machine`(6), N_d = `imsls_i_machine`(10),

$$E_{\min_d} = \texttt{imsls\_i\_machine}(11)$$

and

$$E_{\max_d} = \texttt{imsls\_i\_machine}(12)$$

Missing values in functions are always indicated by NaN. This is `imsls_f_machine`(6) in single precision and `imsls_d_machine`(6) in double precision. There is no missing-value indicator for integers. Users will almost always have to convert from their missing value indicators to NaN.

Example

In this example, all eight values returned by `imsls_f_machine` and by `imsls_d_machine` on a machine with IEEE arithmetic are printed.

```
#include <imsls.h>

main()
{
    int             n;
    float           fans;
    double          dans;
    for (n = 1;  n <= 8;  n++) {
        fans = imsls_f_machine(n);
        printf("imsls_f_machine(%d) = %g\n", n, fans);
    }

    for (n = 1;  n <= 8;  n++) {
        dans = imsls_d_machine(n);
        printf("imsls_d_machine(%d) = %g\n", n, dans);
    }
}
```

Output

```
imsls_f_machine(1) = 1.17549e-38
imsls_f_machine(2) = 3.40282e+38
imsls_f_machine(3) = 5.96046e-08
imsls_f_machine(4) = 1.19209e-07
imsls_f_machine(5) = 0.30103
imsls_f_machine(6) = NaN
imsls_f_machine(7) = Inf
imsls_f_machine(8) = -Inf
imsls_d_machine(1) = 2.22507e-308
imsls_d_machine(2) = 1.79769e+308
imsls_d_machine(3) = 1.11022e-16
imsls_d_machine(4) = 2.22045e-16
imsls_d_machine(5) = 0.30103
imsls_d_machine(6) = NaN
imsls_d_machine(7) = Inf
imsls_d_machine(8) = -Inf
```

data_sets

Retrieves a commonly analyzed data set.

Synopsis

#include `<imsls.h>`

float `*imsls_f_data_sets` (*int* `data_set_choice`, ..., 0)

The type *double* function is `imsls_d_data_sets`.

Required Arguments

int `data_set_choice` (Input)
Data set indicator. Set `data_set_choice` = 0 to print a description of all nine data sets. In this case, any optional arguments are ignored.

data_set_choice	N_observations	n_variables	Description of Data Set
1	16	7	Longley
2	176	2	Wolfer sunspot
3	150	5	Fisher iris

data_set_choice	N_observations	n_variables	Description of Data Set
4	144	1	Box and Jenkins Series G
5	13	5	Draper and Smith Appendix B
6	197	1	Box and Jenkins Series A
7	296	2	Box and Jenkins Series J
8	100	4	Robinson Multichannel Time Series
9	113	34	Afifi and Azen Data Set A

Return Value

If `data_set_choice` ≠ 0, the requested data set is returned. If `data_set_choice` = 0 or an error occurs, `NULL` is returned.

Synopsis with Optional Arguments

```
#include <imsls.h>

float *imsls_f_data_sets (int data_set_choice,
        IMSLS_X_COL_DIM, int x_col_dim,
        IMSLS_N_OBSERVATIONS, int *n_observations,
        IMSLS_N_VARIABLES, int *n_variables,
        IMSLS_PRINT_NONE,
        IMSLS_PRINT_BRIEF,
        IMSLS_PRINT_ALL,
        IMSLS_RETURN_USER, float x[],
        0)
```

Optional Arguments

`IMSLS_X_COL_DIM`, *int* `x_col_dim` (Input)
Column dimension of user allocated space.

`IMSLS_N_OBSERVATIONS`, *int* `*n_observations` (Output)
Number of observations or rows in the output matrix.

`IMSLS_N_VARIABLES`, *int* `*n_variables` (Output)
Number of variables or columns in the output matrix.

`IMSLS_PRINT_NONE`
No printing is performed. This option is the default.

IMSLS_PRINT_BRIEF
Rows 1 through 10 of the data set are printed.

IMSLS_PRINT_ALL
All rows of the data set are printed.

IMSLS_RETURN_USER, *float* x[] (Output)
User-supplied array containing the data set.

Description

Function imsls_f_data_sets retrieves a standard data set frequently cited in statistics text books or in this manual. The following tables gives the references for each data set:

data_set_choice	Reference
1	Longley (1967)
2	Anderson (1971, p.660)
3	Fisher (1936); Mardia et al. (1979, Table 1.2.2)
4	Box and Jenkins (1976, p. 531)
5	Draper and Smith (1981, pp. 629-630)
6	Box and Jenkins (1976, p. 525)
7	Box and Jenkins (1976, pp. 532-533)
8	Robinson (1976, p. 204)
9	Afifi and Azen (1979, pp. 16-22)

Example

In this example, imsls_f_data_sets is used to copy the Draper and Smith (1981, Appendix B) data set into x.

```
#include <imsls.h>

main()
{
    float *x;

    x = imsls_f_data_sets (5, 0);

    imsls_f_write_matrix("Draper and Smith, Appendix B", 13, 5, x, 0);
}
```

Output

```
                Draper and Smith, Appendix B
            1            2            3            4            5
1         7.0         26.0          6.0         60.0         78.5
2         1.0         29.0         15.0         52.0         74.3
3        11.0         56.0          8.0         20.0        104.3
4        11.0         31.0          8.0         47.0         87.6
```

```
 5         7.0        52.0         6.0        33.0        95.9
 6        11.0        55.0         9.0        22.0       109.2
 7         3.0        71.0        17.0         6.0       102.7
 8         1.0        31.0        22.0        44.0        72.5
 9         2.0        54.0        18.0        22.0        93.1
10        21.0        47.0         4.0        26.0       115.9
11         1.0        40.0        23.0        34.0        83.8
12        11.0        66.0         9.0        12.0       113.3
13        10.0        68.0         8.0        12.0       109.4
```

mat_mul_rect

Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any triple product.

Synopsis

#include `<imsls.h>`

float `*imsls_f_mat_mul_rect` (*char* `*string`, ..., 0)

The type *double* function is `imsls_d_mat_mul_rect`.

Required Arguments

char `*string` (Input)
String indicating operation to be performed. See the "Description" section below for more details."

Return Value

The result of the operation. This is always a pointer to a *float*, even if the result is a single number. If no answer was computed, `NULL` is returned.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_mat_mul_rect` (*char* `*string`,
`IMSLS_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *float* `a[]`,
`IMSLS_A_COL_DIM`, *int* `a_col_dim`,
`IMSLS_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *float* `b[]`,
`IMSLS_B_COL_DIM`, *int* `b_col_dim`,
`IMSLS_X_VECTOR`, *int* `nx`, *float* `*x`,
`IMSLS_Y_VECTOR`, *int* `ny`, *float* `*y`,
`IMSLS_RETURN_USER`, *float* `ans[]`,
`IMSLS_RETURN_COL_DIM`, *int* `return_col_dim`,
0)

Optional Arguments

`IMSLS_A_MATRIX`, *int* `nrowa`, *int* `ncola`, *float* `a[]` (Input)
The `nrowa` × `ncola` matrix *A*.

`IMSLS_A_COL_DIM`, *int* `a_col_dim` (Input)
Column dimension of *A*.
Default: `a_col_dim` = `ncola`

`IMSLS_B_MATRIX`, *int* `nrowb`, *int* `ncolb`, *float* `b[]` (Input)
The `nrowb` × `ncolb` matrix *A*.

`IMSLS_B_COL_DIM`, *int* `b_col_dim` (Input)
Column dimension of *B*.
Default: `b_col_dim = ncolb`

`IMSLS_X_VECTOR`, *int* `nx`, *float* `*x` (Input)
Vector *x* of size `nx`.

`IMSLS_Y_VECTOR`, *int* `ny`, *float* `*y` (Input)
Vector *y* of size `ny`.

`IMSLS_RETURN_USER`, *float* `ans[]` (Output)
User-allocated array containing the result.

`IMSLS_RETURN_COL_DIM`, *int* `return_col_dim` (Input)
Column dimension of the answer.
Default: `return_col_dim` = the number of columns in the answer

Description

This function computes a matrix-vector product, a matrix-matrix product, a bilinear form of a matrix, or a triple product according to the specification given by `string`. For example, if "`A*x`" is given, *Ax* is computed. In `string`, the matrices *A* and *B* and the vectors *x* and *y* can be used. Any of these four names can be used with `trans`, indicating transpose. The vectors *x* and *y* are treated as $n \times 1$ matrices.

If `string` contains only one item, such as "`x`" or "`trans(A)`", then a copy of the array, or its transpose, is returned. If `string` contains one multiplication, such as "`A*x`" or "`B*A`", then the indicated product is returned. Some other legal values for `string` are "`trans(y)*A`", "`A*trans(B)`", "`x*trans(y)`", or "`trans(x)*y`".

The matrices and/or vectors referred to in `string` must be given as optional arguments. If `string` is "`B*x`", then `IMSLS_B_MATRIX` and `IMSLS_X_VECTOR` must be given.

Example

Let A, B, x, and y equal the following matrices:

$$A = \begin{bmatrix} 1 & 2 & 9 \\ 5 & 4 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 2 \\ 7 & 4 \\ 9 & 1 \end{bmatrix} \quad x = \begin{bmatrix} 7 \\ 2 \\ 1 \end{bmatrix} \quad y = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

The arrays A^T, Ax, x^TA^T, AB, B^TA^T, x^Ty, xy^T and x^TAy are computed and printed.

```
#include <imsls.h>

main()
{
    float       A[] = {1, 2, 9,
                       5, 4, 7};
    float       B[] = {3, 2,
                       7, 4,
                       9, 1};
    float       x[] = {7, 2, 1};
    float       y[] = {3, 4, 2};
    float       *ans;

    ans = imsls_f_mat_mul_rect("trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        0);
    imsls_f_write_matrix("trans(A)", 3, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("A*x",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_X_VECTOR, 3, x,
        0);
    imsls_f_write_matrix("A*x", 1, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_X_VECTOR, 3, x,
        0);
    imsls_f_write_matrix("trans(x)*trans(A)", 1, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("A*B",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_B_MATRIX, 3, 2, B,
        0);
    imsls_f_write_matrix("A*B", 2, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(B)*trans(A)",
        IMSLS_A_MATRIX, 2, 3, A,
        IMSLS_B_MATRIX, 3, 2, B,
        0);
    imsls_f_write_matrix("trans(B)*trans(A)", 2, 2, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*y",
        IMSLS_X_VECTOR, 3, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("trans(x)*y", 1, 1, ans, 0);
    ans = imsls_f_mat_mul_rect("x*trans(y)",
        IMSLS_X_VECTOR, 3, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("x*trans(y)", 3, 3, ans, 0);

    ans = imsls_f_mat_mul_rect("trans(x)*A*y",
        IMSLS_A_MATRIX, 2, 3, A,
                                /* use only the first 2 components of x */
        IMSLS_X_VECTOR, 2, x,
        IMSLS_Y_VECTOR, 3, y,
        0);
    imsls_f_write_matrix("trans(x)*A*y", 1, 1, ans, 0);
}
```

Output

```
        trans(A)
            1           2
1           1           5
2           2           4
3           9           7

          A*x
         1           2
        20          50

   trans(x)*trans(A)
         1           2
        20          50

           A*B
            1           2
1          98          19
2         106          33

    trans(B)*trans(A)
            1           2
1          98         106
2          19          33

trans(x)*y
        31

             x*trans(y)
            1           2           3
1          21          28          14
2           6           8           4
3           3           4           2

trans(x)*A*y
        293
```

permute_vector

Rearranges the elements of a vector as specified by a permutation.

Synopsis

#include `<imsls.h>`

float `*imsls_f_permute_vector` (*int* `n_elements`, *float* `x[]`,
    int `permutation[]`, *Imsls_permute* `permute`, ..., 0)

The type *double* function is `imsls_d_permute_vector`.

Required Arguments

int `n_elements` (Input)
    Number of elements in the input vector `x`.

float `x[]` (Input)
    Array of length `n_elements` to be permuted.

int `permutation[]` (Input)
    Array of length `n_elements` containing the permutation.

Imsls_permute `permute` (Input)
    Keyword of type *Imsls_permute*. Argument `permute` must be either `IMSLS_FORWARD_PERMUTATION` or `IMSLS_BACKWARD_PERMUTATION`. If `IMSLS_FORWARD_PERMUTATION` is specified, then a forward permutation is performed, i.e., `x(permutation[i])` is moved to location *i* in the return vector. If `IMSLS_BACKWARD_PERMUTATION` is specified, then a backward permutation is performed, i.e., `x[i]` is moved to location `permutation[i]` in the return vector.

Return Value

An array of length `n_elements` containing the input vector `x` permuted.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_permute_vector` (*int* `n_elements`, *float* `x[]`,
    int `permutation[]`, *Imsls_permute* `permute`,
    `IMSLS_RETURN_USER`, *float* `permuted_result[]`,
    0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `permuted_result[]` (Output)
    User-allocated array containing the result of the permutation.

Description

Function `imsls_f_permute_vector` rearranges the elements of a vector according to a permutation vector. The function can perform both forward and backward permutation.

Example

This example rearranges the vector `x` using `permutation`. A forward permutation is performed.

```
#include <imsls.h>

void main()
{
    float x[] = {5.0, 6.0, 1.0, 4.0};
    int permutation[] = {2, 0, 3, 1};
    float     *output;
    int        n_elements = 4;

    output = imsls_f_permute_vector (n_elements, x, permutation,
        IMSLS_FORWARD_PERMUTATION, 0);

    imsls_f_write_matrix ("permuted result", 1, n_elements, output,
                           IMSLS_COL_NUMBER_ZERO, 0);
}
```

Output

```
                permuted result
         0           1           2           3
         1           5           4           6
```

permute_matrix

Permutes the rows or columns of a matrix.

Synopsis

#include `<imsls.h>`

float `*imsls_f_permute_matrix` (*int* `n_rows`, *int* `n_columns`, *float* `a[]`, *int* `permutation[]`, *Imsls_permute* `permute`, ..., 0)

The type *double* function is `imsls_d_permute_matrix`.

Required Arguments

int `n_rows` (Input)
Number of rows in the input matrix `a`.

int `n_columns` (Input)
Number of columns in the input matrix `a`.

float `a[]` (Input)
: Matrix of size `n_rows` × `n_columns` to be permuted.

int `permutation[]` (Input)
: Array of length `n_elements` containing the permutation.

Imsls_permute `permute` (Input)
: Keyword of type *Imsls_permute*. Argument `permute` must be either `IMSLS_PERMUTE_ROWS`, if the rows of `a` are to be interchanged, or `IMSLS_PERMUTE_COLUMNS`, if the columns of `a` are to be interchanged.

Return Value

Array of size `n_rows` × `n_columns` containing the permuted input matrix `a`.

Synopsis with Optional Arguments

#include `<imsls.h>`

float `*imsls_f_permute_matrix` (*int* `n_rows`, *int* `n_columns`,
  float `a[]`,
  int `permutation[]`, *Imsls_permute* `permute`,
  `IMSLS_RETURN_USER`, *float* `permuted_result[]`,
  0)

Optional Arguments

`IMSLS_RETURN_USER`, *float* `permuted_result[]` (Output)
: User-allocated array of size `n_rows` × `n_columns` containing the result of the permutation.

Description

Function `imsls_f_permute_matrix` interchanges the rows or columns of a matrix using a permutation vector. The function permutes a column (row) at a time using function `imsls_f_permute_vector`. This process is continued until all the columns (rows) are permuted. On completion, let B = result and p_i = permutation [i], then $B_{ij} = A_{p_i j}$ for all i, j.

Example

This example permutes the columns of a matrix `a`.

```
#include <imsls.h>

void main()
{
    float a[] = {3.0, 5.0, 1.0, 2.0, 4.0,
                 3.0, 5.0, 1.0, 2.0, 4.0,
                 3.0, 5.0, 1.0, 2.0, 4.0};
    int permutation[] = {2, 3, 0, 4, 1};
    float      *output;
    int         n_rows = 3;
    int         n_columns = 5;
    output = imsls_f_permute_matrix (n_rows, n_columns, a, permutation,
        IMSLS_PERMUTE_COLUMNS,
        0);

    imsls_f_write_matrix ("permuted matrix", n_rows, n_columns, output,
        IMSLS_ROW_NUMBER_ZERO,
        IMSLS_COL_NUMBER_ZERO,
        0);
}
```

Output

```
                     permuted matrix
            0            1            2            3            4
0           1            2            3            4            5
1           1            2            3            4            5
2           1            2            3            4            5
```

binomial_coefficient

Evaluates the binomial coefficient.

Synopsis

#include `<imsls.h>`

int `imsls_f_binomial_coefficient` (*int* `n`, *int* `m`)

The type *double* procedure is `imsls_d_binomial_coefficient`.

Required Arguments

int `n` (Input)
First parameter of the binomial coefficient. Argument `n` must be nonnegative.

int `m` (Input)
Second parameter of the binomial coefficient. Argument `m` must be nonnegative.

Return Value

The binomial coefficient

$$\binom{n}{m}$$

is returned.

Description

The binomial function is defined to be

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

with $n \geq m \geq 0$. Also, n must not be so large that the function overflows.

Example

In this example, $\binom{9}{5}$ is computed and printed.

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

main()
{
    int       n = 9;
    int       m = 5;
    int       ans;

    ans = imsls_f_binomial_coefficient(n, m);
    printf("binomial coefficient = %d\n", ans);
}
```

Output

```
binomial coefficient = 126
```

beta

Evaluates the complete beta function.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta` (*float* `a`, *float* `b`)

The type *double* procedure is `imsls_d_beta`.

Required Arguments

float `a` (Input)
First beta parameter. It must be positive.

float `b` (Input)
Second beta parameter. It must be positive.

Return Value

The value of the beta function β(`a`, `b`). If no result can be computed, then NaN is returned.

Description

The beta function, β(a, b), is defined to be

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} = \int_0^1 t^{a-1}(1-t)^{b-1}\,dt$$

Example

Evaluate the beta function β(0.5, 0.2).

```
#include <imsls.h>

main()
{
    float       x = 0.5;
    float       y = 0.2;
    float       ans;

    ans = imsls_f_beta(x, y);
    printf("beta(%f,%f) = %f\n", x, y, ans);
}
```

Output

```
beta(0.500000,0.200000) = 6.268653
```

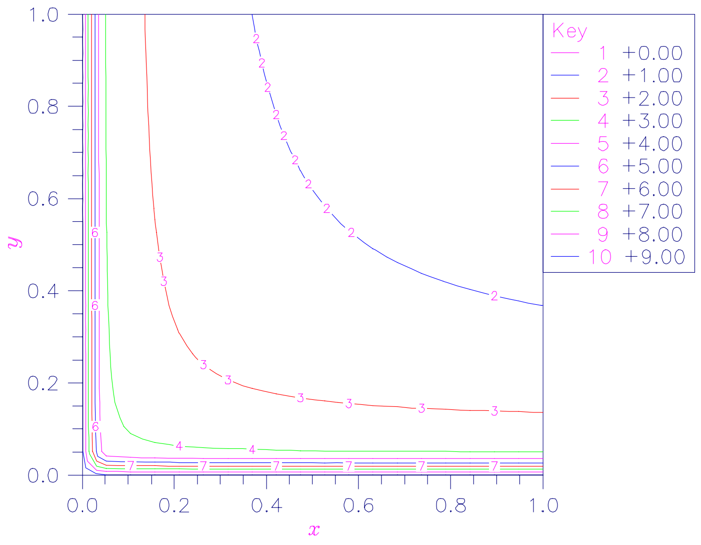


Figure 14–1   Plot of β (*x*, *b*)

The beta function requires that $a > 0$ and $b > 0$. It underflows for large arguments.

Alert Errors

`IMSLS_BETA_UNDERFLOW`	The arguments must not be so large that the result underflows.

Fatal Errors

`IMSLS_ZERO_ARG_OVERFLOW`	One of the arguments is so close to zero that the result overflows.

beta_incomplete

Evaluates the real incomplete beta function $I_x = \beta_x(a, b)/\beta(a, b)$.

Synopsis

#include `<imsls.h>`

float `imsls_f_beta_incomplete` (*float* `x`, *float* `a`, *float* `b`)

The type *double* procedure is `imsls_d_beta_incomplete`.

Required Arguments

float `x` (Input)
Point at which the incomplete beta function is to be evaluated.

float `a` (Input)
Point at which the incomplete beta function is to be evaluated.

float `b` (Input)
Point at which the incomplete beta function is to be evaluated.

Return Value

The value of the incomplete beta function.

Description

The incomplete beta function is defined to be

$$I_x(a,b) = \frac{\beta_x(a,b)}{\beta(a,b)} = \frac{1}{\beta(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}\,dt$$

The incomplete beta function requires that $0 \le x \le 1$, $a > 0$, and $b > 0$. It underflows for sufficiently small x and large a. This underflow is not reported as an error. Instead, the value zero is returned.

Example

Evaluate the log of the incomplete beta function $I_{0.61} = \beta_{0.61}(2.2,3.7)/\beta(2.2,3.7)$.

```
#include <imsls.h>

main()
{
    float       x = 0.61;
    float       a = 2.2;
    float       b = 3.7;
    float       ans;

    ans = imsls_f_beta_incomplete(x, a, b);
    printf("beta incomplete = %f\n", ans);
}
beta incomplete = 0.8822;
```

log_beta

Evaluates the logarithm of the real beta function ln β(*x*, *y*).

Synopsis

#include `<imsls.h>`

float `imsls_f_log_beta` (*float* `x`, *float* `y`)

The type *double* procedure is `imsls_d_log_beta`.

Required Arguments

float `x` (Input)
Point at which the logarithm of the beta function is to be evaluated. It must be positive.

float `y` (Input)
Point at which the logarithm of the beta function is to be evaluated. It must be positive.

Return Value

The value of the logarithm of the beta function β(*x*, *y*).

Description

The beta function, β(*x*, *y*), is defined to be

$$\beta(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$$

and `imsls_f_log_beta` returns ln β(*x*, *y*).

The logarithm of the beta function requires that $x > 0$ and $y > 0$. It can overflow for very large arguments.

Warning Errors

`IMSLS_X_IS_TOO_CLOSE_TO_NEG_1`	The result is accurate to less than one precision because the expression $-x/(x + y)$ is too close to -1.

Example

Evaluate the log of the beta function ln β(0.5, 0.2).

```
#include <imsls.h>

main()
{
    float       x = 0.5;
    float       y = 0.2;
    float       ans;

    ans = imsls_f_log_beta(x, y);
    printf("log beta(%f,%f) = %f\n", x, y, ans);
}
```

Output

```
log beta(0.500000,0.200000) = 1.835562
```

gamma

Evaluates the real gamma function.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma` (*float* `x`)

The type *double* procedure is `imsls_d_gamma`.

Required Arguments

float `x` (Input)
Point at which the gamma function is to be evaluated.

Return Value

The value of the gamma function $\Gamma(x)$.

Description

The gamma function, $\Gamma(x)$, is defined to be

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

For $x < 0$, the above definition is extended by analytic continuation.

The gamma function is not defined for integers less than or equal to zero. It underflows for $x << 0$ and overflows for large x. It also overflows for values near negative integers.

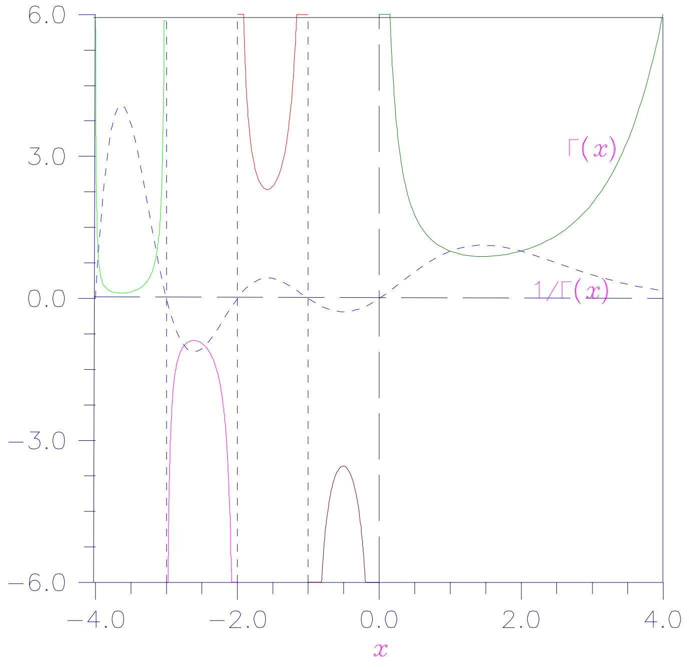


Figure 14-2 Plot of Γ(*x*) and 1/Γ(*x*)

Alert Errors

`IMSLS_SMALL_ARG_UNDERFLOW`	The argument x must be large enough that $\Gamma(x)$ does not underflow. The underflow limit occurs first for arguments close to large negative half integers. Even though other arguments away from these half integers may yield machine-representable values of $\Gamma(x)$, such arguments are considered illegal.

Warning Errors

`IMSLS_NEAR_NEG_INT_WARN`	The result is accurate to less than one-half precision because x is too close to a negative integer.

Example

In this example, Γ(1.5) is computed and printed.

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

main()
{
    float       x = 1.5;
    float       ans;

    ans = imsls_f_gamma(x);
    printf("Gamma(%f) = %f\n", x, ans);
}
```

Output

```
Gamma(1.500000) = 0.886227
```

Fatal Errors

`IMSLS_ZERO_ARG_OVERFLOW`	The argument for the gamma function is too close to zero.
`IMSLS_NEAR_NEG_INT_FATAL`	The argument for the function is too close to a negative integer.
`IMSLS_LARGE_ARG_OVERFLOW`	The function overflows because *x* is too large.
`IMSLS_CANNOT_FIND_XMIN`	The algorithm used to find x_{min} failed. This error should never occur.
`IMSLS_CANNOT_FIND_XMAX`	The algorithm used to find x_{max} failed. This error should never occur.

gamma_incomplete

Evaluates the incomplete gamma function $\gamma(a, x)$.

Synopsis

#include `<imsls.h>`

float `imsls_f_gamma_incomplete` (*float* `a`, *float* `x`)

The type *double* procedure is `imsls_d_gamma_incomplete`.

Required Arguments

float `a` (Input)
Parameter of the incomplete gamma function is to be evaluated. It must be positive.

float x (Input)
Point at which the incomplete gamma function is to be evaluated. It must be nonnegative.

Return Value

The value of the incomplete gamma function $\gamma(a, x)$.

Description

The incomplete gamma function, $\gamma(a, x)$, is defined to be

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt$$

for $x > 0$. The incomplete gamma function is defined only for $a > 0$. Although $\gamma(a, x)$ is well defined for $x > -\infty$, this algorithm does not calculate $\gamma(a, x)$ for negative x. For large a and sufficiently large x, $\gamma(a, x)$ may overflow. $\gamma(a, x)$ is bounded by $\Gamma(a)$, and users may find this bound a useful guide in determining legal values for a.

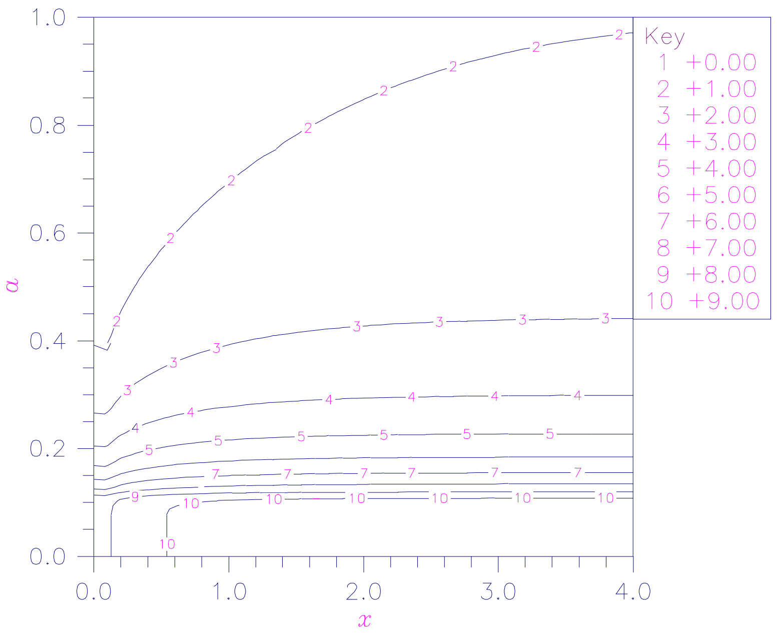


Figure 14-3 Contour Plot of $\gamma(a, x)$

Example

Evaluates the incomplete gamma function at $a = 1$ and $x = 3$.

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

main()
{
    float       x = 3.0;
    float       a = 1.0;
    float       ans;

    ans = imsls_f_gamma_incomplete(a, x);
    printf("incomplete gamma(%f,%f) = %f\n", a, x, ans);
}
```

Output

```
incomplete gamma(1.000000,3.000000) = 0.950213
```

Fatal Errors

`IMSLS_NO_CONV_200_TS_TERMS`	The function did not converge in 200 terms of Taylor series.
`IMSLS_NO_CONV_200_CF_TERMS`	The function did not converge in 200 terms of the continued fraction.

log_gamma

Evaluates the logarithm of the absolute value of the gamma function log $|\Gamma(x)|$.

Synopsis

#include `<imsls.h>`

float `imsls_f_log_gamma` (*float* `x`)

The type *double* procedure is `imsls_d_log_gamma`.

Required Arguments

float `x` (Input)
Point at which the logarithm of the absolute value of the gamma function is to be evaluated.

Return Value

The value of the logarithm of gamma function log $|\Gamma(x)|$.

Description

The logarithm of the absolute value of the gamma function log |Γ(*x*)| is computed.

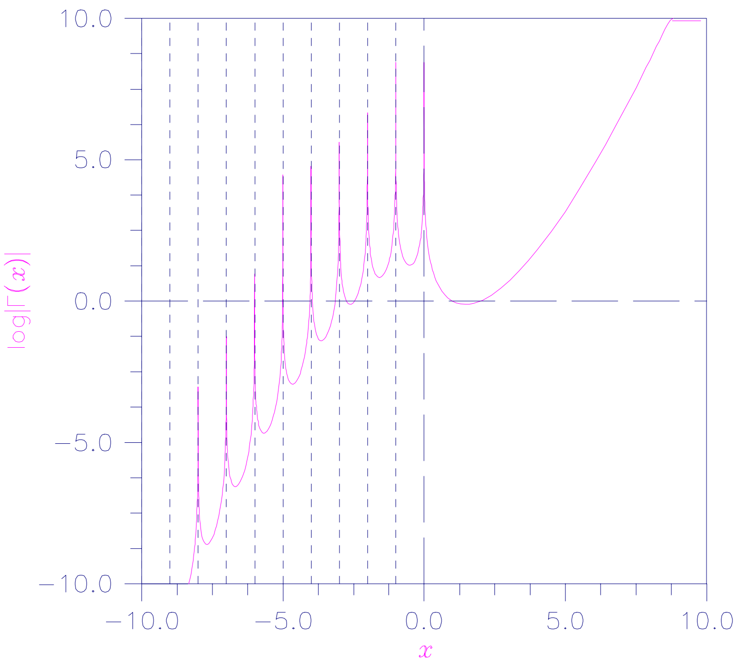


Figure 14-4   Plot of log|Γ(*x*)|

Example

In this example, log |Γ(3.5)| is computed and printed.

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

main()
{
    float       x = 3.5;
    float       ans;
    ans = imsls_f_log_gamma(x);
    printf("log gamma(%f) = %f\n", x, ans);
}
```

Output

```
log gamma(3.500000) = 1.200974
```

Warning Errors

`IMSLS_NEAR_NEG_INT_WARN`	The result is accurate to less than one-half precision because `x` is too close to a negative integer.

Fatal Errors

`IMSLS_NEGATIVE_INTEGER`	The argument for the function cannot be a negative integer.
`IMSLS_NEAR_NEG_INT_FATAL`	The argument for the function is too close to a negative integer.
`IMSLS_LARGE_ABS_ARG_OVERFLOW`	$\|x\|$ must not be so large that the result overflows.

ctime

Returns the number of CPU seconds used.

Synopsis

#include `<imsls.h>`

double `imsls_ctime` ()

Return Value

The number of CPU seconds used by the program.

Example

The CPU time needed to compute

$$\sum_{k=0}^{1,000,000} k$$

is obtained and printed. The time needed is machine dependent. The CPU time needed will varies slightly from run to run on the same machine.

```
#include <imsls.h>

main()
{
    int     k;
    double  sum, time;
                                /* Sum 1 million values */
    for (sum=0, k=1;  k<=1000000; k++)
         sum += k;
                                /* Get amount of CPU time used */
    time = imsls_ctime();
    printf("sum = %f\n", sum);
    printf("time = %f\n", time);
}
```

Output

```
sum = 500000500000.000000
time = 0.820000
```

Reference Material

User Errors

IMSL functions attempt to detect user errors and handle them in a way that provides as much information to the user as possible. To do this, various levels of severity of errors are recognized, and the extent of the error in the context of the purpose of the function also is considered; a trivial error in one situation can be serious in another. IMSL attempts to report as many errors as can reasonably be detected. Multiple errors present a difficult problem in error detection because input is interpreted in an uncertain context after the first error is detected.

What Determines Error Severity

In some cases, the user's input may be mathematically correct, but because of limitations of the computer arithmetic and of the algorithm used, it is not possible to compute an answer accurately. In this case, the assessed degree of accuracy determines the severity of the error. In cases where the function computes several output quantities, some are not computable but most are, an error condition exists. The severity of the error depends on an assessment of the overall impact of the error.

Kinds of Errors and Default Actions

Five levels of severity of errors are defined in IMSL C/Stat/Library. Each level has an associated PRINT attribute and a STOP attribute. These attributes have default settings (YES or NO), but they may also be set by the user. The purpose of having multiple error types is to provide independent control of actions to be taken for errors of different levels of severity. Upon return from an IMSL function, exactly one error state exists. (A code 0 "error" is no error.) Even if more than one informational error occurs, only one message is printed (if the PRINT attribute is YES). Multiple errors for which no corrective action within the calling program is reasonable or necessary result in the printing of multiple messages (if the PRINT attribute for their severity level is YES). Errors of any of the severity levels except `IMSLS_TERMINAL` may be informational errors. The include file, *imsls.h*, defines each of `IMSLS_NOTE`, `IMSLS_ALERT`, `IMSLS_WARNING`, `IMSLS_FATAL`, `IMSLS_TERMINAL`, `IMSLS_WARNING_IMMEDIATE`, and `IMSLS_FATAL_IMMEDIATE` as enumerated data type *Imsls_error*.

`IMSLS_NOTE`. A *note* is issued to indicate the possibility of a trivial error or simply to provide information about the computations.
Default attributes: PRINT=NO, STOP=NO

`IMSLS_ALERT`. An *alert* indicates that a function value has been set to 0 due to underflow.
Default attributes: PRINT=NO, STOP=NO

`IMSLS_WARNING`. A *warning* indicates the existence of a condition that may require corrective action by the user or calling function. A warning error may be issued because the results are accurate to only a few decimal places; because some of the output may be erroneous, but most of the output is correct; or because some assumptions underlying the analysis technique are violated. Usually no corrective action is necessary, and the condition can be ignored.
Default attributes: PRINT=YES, STOP=NO

`IMSLS_FATAL`. A *fatal* error indicates the existence of a condition that may be serious. In most cases, the user or calling function must take corrective action to recover.
Default attributes: PRINT=YES, STOP=YES

`IMSLS_TERMINAL`. A *terminal* error is serious. It usually is the result of an incorrect specification, such as specifying a negative number as the number of equations. These errors can also be caused by various programming errors impossible to diagnose correctly in C. The resulting error message may be perplexing to the user. In such cases, the user is advised to compare carefully the actual arguments passed to the function with the dummy argument descriptions given in the documentation. Special attention should be given to checking argument order and data types.

A terminal error is not an informational error, because corrective action within the program is generally not reasonable. In normal use, execution is terminated immediately when a terminal error occurs. Messages relating to more than one terminal error are printed if they occur.
Default attributes: PRINT=YES, STOP=YES

`IMSLS_WARNING_IMMEDIATE`. An *immediate warning* error is identical to a warning error, except it is printed immediately.
Default attributes: PRINT=YES, STOP=NO

`IMSLS_FATAL_IMMEDIATE`. An *immediate fatal* error is identical to a fatal error, except it is printed immediately.
Default attributes: PRINT=YES, STOP=YES

The user can set PRINT and STOP attributes by calling function `imsls_error_options` as described in Chapter 14, “Utilities.”

Errors in Lower-level Functions

It is possible that a user's program may call an IMSL function that in turn calls a nested sequence of lower-level IMSL functions. If an error occurs at a lower level in such a nest of functions and if the lower-level function cannot pass the information up to the original user-called function, then a traceback of the functions is produced. The only common situation in which this can occur is when an IMSL function calls a user-supplied routine that in turn calls another IMSL function.

Functions for Error Handling

The user may interact in two ways with the IMSL error-handling system: (1) to change the default actions and (2) to determine the code of an informational error so as to take corrective action. The IMSL functions to use are `imsls_error_options` and `imsls_error_code`. Function `imsls_error_options` sets the actions to be taken when errors occur. Function `imsls_error_code` retrieves the integer code for an informational error. These functions are documented in Chapter 14, "Utilities."

Threads and Error Handling

If multiple threads are used then default settings are valid for each thread but can be altered for each individual thread. When using threads it is necessary to set options using `imsls_error_options` (excluding `IMSLS_SET_SIGNAL_TRAPPING`) for each thread by calling `imsls_error_options` from within each thread.

The IMSL signal-trapping mechanism must be disabled when multiple threads are used. The IMSL signal-trapping mechanism can be disabled by making the following call before any threads are created:

```
imsls_error_options(IMSLS_SET_SIGNAL_TRAPPING, 0, 0);
```

See Chapter 14, "Utilities", examples 3 and 4 of `imsls_error_options` for multithreaded examples.

Use of Informational Error to Determine Program Action

In the program segment below, a factor analysis is to be performed on the matrix covariances. If it is determined that the matrix is singular (and often this is not immediately obvious), the program is to take a different branch.

```
x = imsls_f_factor_analysis (nobs, covariances,
        n_factors, 0);
if (imsls_error_code() == IMSLS_COV_IS_SINGULAR) {
        /*  Handle a singular matrix  */
}
```

Additional Examples

See functions `imsls_error_options` and `imsls_error_code` in Chapter 14, “Utilities” for additional examples.

Product Support

Contacting Visual Numerics Support

Users within support warranty may contact Visual Numerics regarding the use of the IMSL C Numerical Libraries. Visual Numerics can consult on the following topics:

- Clarity of documentation
- Possible Visual Numerics-related programming problems
- Choice of IMSL Libraries functions or procedures for a particular problem
- Evolution of the IMSL Libraries

Not included in these consultation topics are mathematical/statistical consulting and debugging of your program.

Consultation

Contact Visual Numerics Product Support emailing:

- `support@houston.vni.com`

Electronic addresses are not handled uniformly across the major networks, and some local conventions for specifying electronic addresses might cause further variations to occur; contact your E-mail postmaster for further details.

The following describes the procedure for consultation with Visual Numerics:

1. Include license number
2. Include the product name and version number: IMSL C/Stat/Library Version 5.5
3. Include compiler and operating system version numbers

4. Include the name of the routine for which assistance is needed and a description of the problem

Appendix A: References

Abramowitz and Stegun

Abramowitz, Milton and Irene A. Stegun (editors) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards, Washington.

Afifi and Azen

Afifi, A.A. and S.P. Azen (1979), *Statistical Analysis: A Computer Oriented Approach*, 2d ed., Academic Press, New York.

Agresti, Wackerly, and Boyette

Agresti, Alan, Dennis Wackerly, and James M. Boyette (1979), Exact conditional tests for cross-classifications: Approximation of attained significance levels, *Psychometrika*, **44**, 75-83.

Ahrens and Dieter

Ahrens, J.H. and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing,* **12**, 223−246.

Ahrens, J.H., and U. Dieter (1985), Sequential random sampling, *ACM Transactions on Mathematical Software*, **11**, 157−169.

Anderberg

Anderberg, Michael R. (1973), *Cluster Analysis for Applications*, Academic Press, New York.

Anderson

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, John Wiley & Sons, New York.

Anderson and Bancroft

Anderson, R.L. and T.A. Bancroft (1952), *Statistical Theory in Research*, McGraw-Hill Book Company, New York.

Atkinson

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141−145.

Atkinson, A.C. (1985), *Plots, Transformations, and Regression*, Claredon Press, Oxford.

Barrodale and Roberts

Barrodale, I., and F.D.K. Roberts (1973), An improved algorithm for discrete L_1 approximation, *SIAM Journal on Numerical Analysis*, **10**, 839−848.

Barrodale, I., and F.D.K. Roberts (1974), Solution of an overdetermined system of equations in the l_1 norm, *Communications of the ACM*, **17**, 319−320.

Barrodale, I., and C. Phillips (1975), Algorithm 495. Solution of an overdetermined system of linear equations in the Chebyshev norm, *ACM Transactions on Mathematical Software*, **1**, 264−270.

Bartlett, M. S.

Bartlett, M.S. (1935), Contingency table interactions, *Journal of the Royal Statistics Society Supplement*, **2**, 248−252.

Bartlett, M. S. (1937) Some examples of statistical methods of research in agriculture and applied biology*, Supplement to the Journal of the Royal Statistical Society*, **4**, 137-183.

Bartlett, M. (1937), The statistical conception of mental factors, *British Journal of Psychology*, **28**, 97–104.

Bartlett, M.S. (1946), On the theoretical specification and sampling properties of autocorrelated time series, *Supplement to the Journal of the Royal Statistical Society*, **8**, 27–41.

Bartlett, M.S. (1978), *Stochastic Processes,* 3rd. ed., Cambridge University Press, Cambridge.

Bays and Durham

Bays, Carter and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59−64.

Bendel and Mickey

Bendel, Robert B., and M. Ray Mickey (1978), Population correlation matrices for sampling experiments, *Communications in Statistics*, **B7**, 163−182.

Best and Fisher

Best, D.J., and N.I. Fisher (1979), Efficient simulation of the von Mises distribution, *Applied Statistics*, **28**, 152–157.

Bishop et al

Bishop, Yvonne M.M., Stephen E. Feinberg, and Paul W. Holland (1975), *Discrete Multivariate Analysis: Theory and Practice*, MIT Press, Cambridge, Mass.

Bjorck and Golub

Bjorck, Ake, and Gene H. Golub (1973), Numerical Methods for Computing Angles Between Subspaces, *Mathematics of Computation,* **27**, 579–594.

Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

Bosten and Battiste

Bosten, Nancy E. and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156s–157.

Box and Jenkins

Box, George E.P. and Gwilym M. Jenkins (1976), *Time Series Analysis: Forecasting and Control*, revised ed., Holden-Day, Oakland.

Box and Pierce

Box, G.E.P., and David A. Pierce (1970), Distribution of residual autocorrelations in autoregressive-integrated moving average time series models, *Journal of the American Statistical Association*, **65**, 1509–1526.

Box and Tidwell

Box, G.E.P. and P.W. Tidwell (1962), Transformation of the independent variables, *Technometrics*, **4**, 531–550.

Boyette

Boyette, James M. (1979), Random RC tables with given row and column totals, *Applied Statistics*, **28**, 329–332.

Bradley

Bradley, J.V. (1968), *Distribution-Free Statistical Tests*, Prentice-Hall, New Jersey.

Breslow

Breslow, N.E. (1974), Covariance analysis of censored survival data, *Biometrics*, **30**, 89–99.

Brown

Brown, Morton E. (1983), MCDP4F, two-way and multiway frequency tables-measures of association and the log-linear model (complete and incomplete tables), in *BMDP Statistical Software, 1983 Printing with Additions*, (edited by W.J. Dixon), University of California Press, Berkeley.

Brown and Benedetti

Brown, Morton B. and Jacqualine K. Benedetti (1977), Sampling behavior and tests for correlation in two-way contingency tables, *Journal of the American Statistical Association*, **42**, 309–315.

Cheng

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.

Chiang

Chiang, Chin Long (1968), *Introduction to Stochastic Processes in Statistics*, John Wiley & Sons, New York.

Conover

Conover, W.J. (1980), *Practical Nonparametric Statistics*, 2d ed., John Wiley & Sons, New York.

Conover and Iman

Conover, W.J. and Ronald L. Iman (1983), *Introduction to Modern Business Statistics*, John Wiley & Sons, New York.

Conover, W. J., Johnson, M. E., and Johnson, M. M

Conover, W. J., Johnson, M. E., and Johnson, M. M. (1981) A comparative study of tests for homogeneity of variances, with applications to the outer continental shelf bidding data, *Technometrics*, **23**, 351-361.

Cook and Weisberg

Cook, R. Dennis and Sanford Weisberg (1982), *Residuals and Influence in Regression*, Chapman and Hall, New York.

Cooper

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190−192.

Cox

Cox, David R. (1970), *The Analysis of Binary Data*, Methuen, London.

Cox, D.R. (1972), Regression models and life tables (with discussion), *Journal of the Royal Statistical Society*, Series B, *Methodology*, **34**, 187–220.

Cox and Lewis

Cox, D.R., and P.A.W. Lewis (1966), *The Statistical Analysis of Series of Events*, Methuen, London.

Cox and Oakes

Cox, D.R., and D. Oakes (1984), *Analysis of Survival Data*, Chapman and Hall, London.

Cox and Stuart

Cox, D.R., and A. Stuart (1955), Some quick sign tests for trend in location and dispersion, *Biometrika*, **42**, 80–95.

D'Agostino and Stevens

D'Agostino, Ralph B. and Michael A. Stevens (1986), *Goodness-of-Fit Techniques*, Marcel Dekker, New York.

Dallal and Wilkinson

Dallal, Gerald E. and Leland Wilkinson (1986), An analytic approximation to the distribution of Lilliefor's test statistic for normality, *The American Statistician*, **40**, 294–296.

Dennis and Schnabel

Dennis, J.E., Jr. and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

Devore

Devore, Jay L (1982), *Probability and Statistics for Engineering and Sciences*, Brooks/Cole Publishing Company, Monterey, Calif.

Draper and Smith

Draper, N.R. and H. Smith (1981), *Applied Regression Analysis*, 2d ed., John Wiley & Sons, New York.

Durbin

Durbin, J. (1960), The fitting of time series models, *Revue Institute Internationale de Statistics*, **28**, 233–243.

Efroymson

Efroymson, M.A. (1960), Multiple regression analysis, *Mathematical Methods for Digital Computers*, Volume 1, (edited by A. Ralston and H. Wilf), John Wiley & Sons, New York, 191−203.

Ekblom

Ekblom, Hakan (1973), Calculation of linear best L_p-approximations, *BIT*, **13**, 292−300.

Ekblom, Hakan (1987), The L_1-estimate as limiting case of an L_p or Huber-estimate, in *Statistical Data Analysis Based on the* L_1*-Norm and Related Methods* (edited by Yadolah Dodge), North-Holland, Amsterdam, 109−116.

Elandt-Johnson and Johnson

Elandt-Johnson, Regina C., and Norman L. Johnson (1980), *Survival Models and Data Analysis*, John Wiley & Sons, New York, 172−173.

Emmett

Emmett, W.G. (1949), Factor analysis by Lawless method of maximum likelihood, *British Journal of Psychology, Statistical Section*, **2**, 90−97.

Engle

Engle, C. (1982), Autoregressive conditional heteroskedasticity with estimates of the variance of U.K. inflation, *Econometrica ,* **50**, 987−1008.

Fisher

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *The Annals of Eugenics*, **7**, 179−188.

Fishman

Fishman, George S. (1978), *Principles of Discrete Event Simulation*, John Wiley & Sons, New York.

Fishman and Moore

Fishman, George S. and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus , *Journal of the American Statistical Association*, **77**, 129–136.

Forsythe

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74–88.

Fuller

Fuller, Wayne A. (1976), *Introduction to Statistical Time Series*, John Wiley & Sons, New York.

Furnival and Wilson

Furnival, G.M. and R.W. Wilson, Jr. (1974), Regressions by leaps and bounds, *Technometrics*, **16**, 499–511.

Fushimi

Fushimi, Masanori (1990), Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$, *Journal of Computational and Applied Mathematics*, **31**, 105–118.

Gentleman

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448–454.

Gibbons

Gibbons, J.D. (1971), *Nonparametric Statistical Inference*, McGraw-Hill, New York.

Girschick

Girschick, M.A. (1939), On the sampling theory of roots of determinantal equations, *Annals of Mathematical Statistics*, **10**, 203–224.

Golub and Van Loan

Golub, Gene H. and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Md.

Gonin and Money

Gonin, Rene, and Arthur H. Money (1989), *Nonlinear L_p-Norm Estimation*, Marcel Dekker, New York.

Goodnight

Goodnight, James H. (1979), A tutorial on the SWEEP operator, *The American Statistician*, **33**, 149–158.

Graybill

Graybill, Franklin A. (1976), *Theory and Application of the Linear Model*, Duxbury Press, North Scituate, Mass.

Griffin and Redish

Griffin, R. and K.A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

Gross and Clark

Gross, Alan J., and Virginia A. Clark (1975), *Survival Distributions: Reliability Applications in the Biomedical Sciences*, John Wiley & Sons, New York.

Gruenberger and Mark

Gruenberger, F., and A.M. Mark (1951), The d^2 test of random digits, *Mathematical Tables and Other Aids in Computation*, **5**, 109–110.

Guerra et al.

Guerra, Victor O., Richard A. Tapia, and James R. Thompson (1976), A random number generator for continuous random variables based on an interpolation procedure of Akima, in *Proceedings of the Ninth Interface Symposium on Computer Science and Statistics*, (edited by David C. Hoaglin and Roy E. Welsch), Prindle, Weber & Schmidt, Boston, 228–230.

Haldane

Haldane, J.B.S. (1939), The mean and variance of when used as a test of homogeneity, when expectations are small, *Biometrika*, **31**, 346.

Harman

Harman, Harry H. (1976), *Modern Factor Analysis*, 3d ed. revised, University of Chicago Press, Chicago.

Hart et al

Hart, John F., E.W. Cheney, Charles L. Lawson, Hans J. Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, Jr., and Christoph Witzgall (1968), *Computer Approximations*, John Wiley & Sons, New York.

Hartigan

Hartigan, John A. (1975), *Clustering Algorithms*, John Wiley & Sons, New York.

Hartigan and Wong

Hartigan, J.A. and M.A. Wong (1979), Algorithm AS 136: A *K*-means clustering algorithm, *Applied Statistics*, **28**, 100−108.

Hayter

Hayter, Anthony J. (1984), A proof of the conjecture that the Tukey-Kramer multiple comparisons procedure is conservative, *Annals of Statistics*, **12**, 61−75.

Heiberger

Heiberger, Richard M. (1978), Generation of random orthogonal matrices, *Applied Statistics*, **27**, 199−206.

Hemmerle.

Hemmerle, William J. (1967), *Statistical Computations on a Digital Computer*, Blaisdell Publishing Company, Waltham, Mass.

Herraman

Herraman, C. (1968), Sums of squares and products matrix, *Applied Statistics*, **17**, 289−292.

Hill

Hill, G.W. (1970), Student's *t*-distribution, *Communications of the ACM*, **13**, 617−619.

Hill, G.W. (1970), Student's *t*-quantiles, *Communications of the ACM*, **13**, 619−620.

Hinkelmann, K and Kemthorne

Hinkelmann, K and Kemthorne, O (1994) *Design and Analysis of Experiments – Vol 1*, John Wiley.

Hinkley

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67−69.

Hoaglin and Welsch

Hoaglin, David C. and Roy E. Welsch (1978), The hat matrix in regression and ANOVA, *The American Statistician*, **32**, 17−22.

Hocking

Hocking, R.R. (1972), Criteria for selection of a subset regression: Which one should be used?, *Technometrics*, **14**, 967−970.

Hocking, R.R. (1973), A discussion of the two-way mixed model, *The American Statistician*, **27**, 148−152.

Hocking, R.R. (1985), *The Analysis of Linear Models*, Brooks/Cole Publishing Company, Monterey, California.

Huber

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

Hughes and Saw

Hughes, David T., and John G. Saw (1972), Approximating the percentage points of Hotelling's generalized T_0^2 statistic, *Biometrika*, **59**, 224–226.

Iman and Davenport

Iman, R.L., and J.M. Davenport (1980), Approximations of the critical region of the Friedman statistic, *Communications in Statistics*, **A9(6)**,
571−595.

Jennrich and Robinson

Jennrich, R.I. and S.M. Robinson (1969), A Newton-Raphson algorithm for maximum likelihood factor analysis, *Psychometrika*, **34**, 111−123.

Jennrich and Sampson

Jennrich, R.I. and P.F. Sampson (1966), Rotation for simple loadings, *Psychometrika*, **31**, 313–323.

John

John, Peter W.M. (1971), *Statistical Design and Analysis of Experiments*, Macmillan Company, New York.

Jöhnk

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufallszahlen, *Metrika*, **8**, 5−15.

Johnson and Kotz

Johnson, Norman L., and Samuel Kotz (1969), *Discrete Distributions*, Houghton Mifflin Company, Boston.

Johnson, Norman L., and Samuel Kotz (1970a), *Continuous Univariate Distributions*-1, John Wiley & Sons, New York.

Johnson, Norman L., and Samuel Kotz (1970b), *Continuous Univariate Distributions*-2, John Wiley & Sons, New York.

Johnson and Welch

Johnson, D.G., and W.J. Welch (1980), The generation of pseudo-random correlation matrices, *Journal of Statistical Computation and Simulation*, **11**, 55−69.

Jonckheere

Jonckheere, A.R. (1954), A distribution-free *k*-sample test against ordered alternatives, *Biometrika*, **41**, 133−143.

Jöreskog

Jöreskog, K.G. (1977), Factor analysis by least squares and maximum-likelihood methods, *Statistical Methods for Digital Computers*, (edited by Kurt Enslein, Anthony Ralston, and Herbert S. Wilf), John Wiley & Sons, New York, 125−153.

Kachitvichyanukul

Kachitvichyanukul, Voratas (1982), *Computer generation of Poisson, binomial, and hypergeometric random variates*, Ph.D. dissertation, Purdue University, West Lafayette, Indiana.

Kaiser

Kaiser, H.F. (1963), Image analysis, *Problems in Measuring Change*, (edited by C. Harris), University of Wisconsin Press, Madison, Wis.

Kaiser and Caffrey

Kaiser, H.F. and J. Caffrey (1965), Alpha factor analysis, *Psychometrika*, **30**, 1−14.

Kalbfleisch and Prentice

Kalbfleisch, John D., and Ross L. Prentice (1980), *The Statistical Analysis of Failure Time Data*, John Wiley & Sons, New York.

Kemp

Kemp, A.W., (1981), Efficient generation of logarithmically distributed pseudo-random variables, *Applied Statistics,* **30**, 249−253.

Kendall and Stuart

Kendall, Maurice G. and Alan Stuart (1973), *The Advanced Theory of Statistics*, Volume 2: *Inference and Relationship*, 3d ed., Charles Griffin & Company, London.

Kendall, Maurice G. and Alan Stuart (1979), *The Advanced Theory of Statistics*, Volume 2: *Inference and Relationship*, 4th ed., Oxford University Press, New York.

Kendall et al.

Kendall, Maurice G., Alan Stuart, and J. Keith Ord (1983), *The Advanced Theory of Statistics*, Volume 3: *Design and Analysis, and Time Series*, 4th. ed., Oxford University Press, New York.

Kennedy and Gentle

Kennedy, William J., Jr. and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

Kuehl, R. O.

Kuehl, R. O. (2000) *Design of Experiments: Statistical Principles of Research Design and Analysis*, 2nd edition, Duxbury Press.

Kim and Jennrich

Kim, P.J., and R.I. Jennrich (1973), Tables of the exact sampling distribution of the two sample Kolmogorov-Smirnov criterion D_{mn} ($m < n$), in *Selected Tables in Mathematical Statistics*, Volume 1, (edited by H. L. Harter and D.B. Owen), American Mathematical Society, Providence, Rhode Island.

Kinderman and Ramage

Kinderman, A.J., and J.G. Ramage (1976), Computer generation of normal random variables, *Journal of the American Statistical Association*, **71**, 893−896.

Kinderman et al.

Kinderman, A.J., J.F. Monahan, and J.G. Ramage (1977), Computer methods for sampling from Student's *t* distribution, *Mathematics of Computation* **31**, 1009−1018.

Kinnucan and Kuki

Kinnucan, P. and H. Kuki (1968), *A Single Precision INVERSE Error Function Subroutine*, Computation Center, University of Chicago.

Kirk

Kirk, Roger E. (1982), *Experimental Design: Procedures for the Behavioral Sciences*, 2d ed., Brooks/Cole Publishing Company, Monterey, Calif.

Knuth

Knuth, Donald E. (1981), *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, 2d ed., Addison-Wesley, Reading, Mass.

Kshirsagar

Kshirsagar, Anant M. (1972), *Multivariate Analysis*, Marcel Dekker, New York.

Lachenbruch

Lachenbruch, Peter A. (1975), *Discriminant Analysis*, Hafner Press, London.

Lai

Lai, D. (1998a), Local asymptotic normality for location-scale type processes. *Far East Journal of Theorectical Statistics*, (in press).

Lai, D. (1998b), Asymptotic distributions of the correlation integral based statistics. *Journal of Nonparametric Statistics*, (in press).

Lai, D. (1998c), Asymptotic distributions of the estimated BDS statistic and residual analysis of AR Models on the Canadian lynx data. *Journal of Biological Systems*, (in press).

Laird and Oliver

Laird, N.M., and D. Fisher (1981), Covariance analysis of censored survival data using log-linear analysis techniques, *JASA* **76**, 1231−1240.

Lawless

Lawless, J.F. (1982), *Statistical Models and Methods for Lifetime Data*, John Wiley & Sons, New York.

Lawley and Maxwell

Lawley, D.N. and A.E. Maxwell (1971), *Factor Analysis as a Statistical Method*, 2d ed., Butterworth, London.

Learmonth and Lewis

Learmonth, G.P. and P.A.W. Lewis (1973), *Naval Postgraduate School Random Number Generator Package LLRANDOM, NPS55LW73061A*, Naval Postgraduate School, Monterey, Calif.

Lee

Lee, Elisa T. (1980), *Statistical Methods for Survival Data Analysis*, Lifetime Learning Publications, Belmont, Calif.

Lehmann

Lehmann, E.L. (1975), *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, San Francisco.

Levenberg

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

Levene, H.

Levene, H. (1960) In *Contributions to Probability and Statistics: Essays in Honor of Harold Hotelling*, I. Olkin et al. editors, Stanford University Press, 278-292.

Lewis et al.

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/360, *IBM Systems Journal*, **8**, 136–146.

Liffiefors

Lilliefors, H.W. (1967), On the Kolmogorov-Smirnov test for normality with mean and variance unknown, *Journal of the American Statistical Association*, **62**, 534–544.

Ljung and Box

Ljung, G.M., and G.E.P. Box (1978), On a measure of lack of fit in time series models, *Biometrika*, **65**, 297–303.

Longley

Longley, James W. (1967), An appraisal of least-squares programs for the electronic computer from the point of view of the user, *Journal of the American Statistical Association*, **62**, 819–841.

Marsaglia

Marsaglia, George (1964), Generating a variable from the tail of a normal distribution, *Technometrics*, **6**, 101–102.

Marsaglia, G. (1968), Random numbers fall mainly in the planes, *Proceedings of the National Academy of Sciences*, **61**, 25–28.

Marsaglia, G. (1972), The structure of linear congruential sequences, in *Applications of Number Theory to Numerical Analysis*, (edited by S. K. Zaremba), Academic Press, New York, 249−286.

Marsaglia, George (1972), Choosing a point from the surface of a sphere, *The Annals of Mathematical Statistics*, **43**, 645−646.

McKean and Schrader

McKean, Joseph W., and Ronald M. Schrader (1987), Least absolute errors analysis of variance, in *Statistical Data Analysis Based on the L_1-Norm and Related Methods* (edited by Yadolah Dodge), North-Holland, Amsterdam, 297−305.

McKeon

McKeon, James J. (1974), *F* approximations to the distribution of Hotelling's T_0^2, *Biometrika*, **61**, 381–383.

McCullagh and Nelder

McCullagh, P., and J.A. Nelder, (1983), *Generalized Linear Models*, Chapman and Hall, London.

Maindonald

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

Marazzi

Marazzi, Alfio (1985), Robust affine invariant covariances in ROBETH, ROBETH-85 document No. 6, Division de Statistique et Informatique, Institut Universitaire de Medecine Sociale et Preventive, Laussanne.

Mardia et al.

Mardia, K.V. (1970), Measures of multivariate skewness and kurtosis with applications, *Biometrics*, **57**, 519−530.

Mardia, K.V., J.T. Kent, J.M. Bibby (1979), *Multivariate Analysis*, Academic Press, New York.

Mardia and Foster

Mardia, K.V. and K. Foster (1983), Omnibus tests of multinormality based on skewness and kurtosis, *Communications in Statistics A, Theory and Methods*, **12**, 207−221.

Marquardt

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431−441.

Marsaglia

Marsaglia, George (1964), Generating a variable from the tail of a normal distribution, *Technometrics*, **6**, 101−102.

Marsaglia and Bray

Marsaglia, G. and T.A. Bray (1964), A convenient method for generating normal variables, *SIAM Review*, **6**, 260−264.

Marsaglia et al.

Marsaglia, G., M.D. MacLaren, and T.A. Bray (1964), A fast procedure for generating normal random variables, *Communications of the ACM*, **7**, 4−10.

Merle and Spath

Merle, G., and H. Spath (1974), Computational experiences with discrete L_p approximation, *Computing*, **12**, 315−321.

Miller

Miller, Rupert G., Jr. (1980), *Simultaneous Statistical Inference*, 2d ed., Springer-Verlag, New York.

Milliken and Johnson

Milliken, George A. and Dallas E. Johnson (1984), *Analysis of Messy Data, Volume 1: Designed Experiments*, Van Nostrand Reinhold, New York.

Moran

Moran, P.A.P. (1947), Some theorems on time series I, *Biometrika*, **34**, 281–291.

Moré et al.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for [4] MINPACK-1*, Argonne National Laboratory Report ANL-80_74, Argonne, Ill.

Morrison

Morrison, Donald F. (1976), *Multivariate Statistical Methods*, 2nd. ed. McGraw-Hill Book Company, New York.

Muller

Muller, M.E. (1959), A note on a method for generating points uniformly on N-dimensional spheres, *Communications of the ACM*, **2**, 19–20.

Nelson

Nelson, D. B. (1991), Conditional heteroskedasticity in asset returns: A new approach. *Econometrica*, , **59**, 347–370.

Nelson

Nelson, Peter (1989), Multiple Comparisons of Means Using Simultaneous Confidence Intervals, *Journal of Quality Technology*, **21**, 232–241.

Neter

Neter, John (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Ill.

Neter and Wasserman

Neter, John and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Ill.

Noether

Noether, G.E. (1956), Two sequential tests against trend, *Journal of the American Statistical Association*, **51**, 440–450.

Owen

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley Publishing Company, Reading, Mass.

Owen, D.B. (1965), A special case of the bivariate non-central *t* distribution, *Biometrika*, **52**, 437–446.

Palm

Palm, F. C. (1996), GARCH models of volatility. In *Handbook of Statistics*, Vol. 14, 209-240. Eds: Maddala and Rao. Elsevier,New York.

Patefield

Patefield, W.M. (1981), An efficient method of generating $R \times C$ tables with given row and column totals, *Applied Statistics*, **30**, 91–97.

Patefield and Tandy

Patefield, W.M. (1981), and Tandy D. (2000) Fast and Accurate Calculation of Owen’s T-Function, *J. Statistical Software*, **5**, Issue 5.

Peixoto

Peixoto, Julio L. (1986), Testable hypotheses in singular fixed linear models, Communications in Statistics: *Theory and Methods*, **15**,
1957−1973.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

Pillai

Pillai, K.C.S. (1985), Pillai’s trace, in *Encyclopedia of Statistical Sciences, Volume 6*, (edited by Samuel Kotz and Norman L. Johnson), John Wiley & Sons, New York, 725−729.

Pregibon

Pregibon, Daryl (1981), Logistic regression diagnostics, *The Annals of Statistics*, **9**, 705−724.

Prentice

Prentice, Ross L. (1976), A generalization of the probit and logit methods for dose response curves, *Biometrics*, **32**, 761−768.

Priestley

Priestley, M.B. (1981), *Spectral Analysis and Time Series*, Volumes 1 and 2, Academic Press, New York.

Rao

Rao, C. Radhakrishna (1973), *Linear Statistical Inference and Its Applications*, 2d ed., John Wiley & Sons, New York.

Robinson

Robinson, Enders A. (1967), *Multichannel Time Series Analysis with Digital Computer Programs*, Holden-Day, San Francisco.

Royston

Royston, J.P. (1982a), An extension of Shapiro and Wilk's *W* test for normality to large samples, *Applied Statistics*, **31**, 115−124.

Royston, J.P. (1982b), The *W* test for normality, *Applied Statistics*, **31**, 176−180.

Royston, J.P. (1982c), Expected normal order statistics (exact and approximate), *Applied Statistics*, **31**, 161−165.

Sallas

Sallas, William M. (1990), An algorithm for an L_p norm fit of a multiple linear regression model, *American Statistical Association 1990 Proceedings of the Statistical Computing Section*, 131−136.

Sallas and Lionti

Sallas, William M. and Abby M. Lionti (1988), *Some useful computing formulas for the nonfull rank linear model with linear equality restrictions*, IMSL Technical Report 8805, IMSL, Houston.

Savage

Savage, I. Richard (1956), Contributions to the theory of rank order statistics-the two-sample case, *Annals of Mathematical Statistics*, **27**, 590−615.

Scheffe

Scheffe, Henry (1959), *The Analysis of Variance*, John Wiley & Sons, New York.

Schmeiser

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154−160.

Schmeiser and Babu

Schmeiser, Bruce W. and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917−926.

Schmeiser and Kachitvichyanukul

Schmeiser, Bruce and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81−4, School of Industrial Engineering, Purdue University, West Lafayette, Ind.

Schmeiser and Lal

Schmeiser, Bruce W. and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679−682.

Searle

Searle, S.R. (1971), *Linear Models*, John Wiley & Sons, New York.

Seber

Seber, G.A.F. (1984), *Multivariate Observations*, John Wiley & Sons, New York.

Snedecor and Cochran

Snedecor and Cochran (1967) *Statistical Methods*, 6th edition, Iowa State University Press.

Snedecor, George W. & Cochran, William G.

Snedecor, George W. and Cochran, William G. (1967) *Statistical Methods*, 6th edition, Iowa State University Press, 296-298.

Shampine

Shampine, L.F. (1975), Discrete least-squares polynomial fits, *Communications of the ACM*, **18**, 179−180.

Siegal

Siegal, Sidney (1956), *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, New York.

Singleton

Singleton, R.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185−187.

Smirnov

Smirnov, N.V. (1939), Estimate of deviation between empirical distribution functions in two independent samples (in Russian), *Bulletin of Moscow University*, **2**, 3−16.

Smith and Dubey

Smith, H., and S. D. Dubey (1964), "Some reliability problems in the chemical industry", Industrial Quality Control, 21 (2), 1964, 64-70.

Snedecor and Cochran

Snedecor, George W. and William G. Cochran (1967), *Statistical Methods*, 6th ed., Iowa State University Press, Ames, Iowa.

Sposito

Sposito, Vincent A. (1989), Some properties of L_p-estimators, in *Robust Regression*: *Analysis and Applications* (edited by Kenneth D. Lawrence and Jeffrey L. Arthur), Marcel Dekker, New York, 23−58.

Spurrier and Isham

Spurrier, John D. and Steven P. Isham (1985), Exact simultaneous confidence intervals for pairwise comparisons of three normal means, *Journal of the American Statistical Association*, **80**, 438–442.

Stablein, Carter, and Novak

Stablein, D.M, W.H. Carter, and J.W. Novak (1981), Analysis of survival data with nonproportional hazard functions, *Controlled Clinical Trials*, **2**, 149–159.

Stahel

Stahel, W. (1981), Robuste Schatzugen: Infinitesimale Opimalitat und Schatzugen von Kovarianzmatrizen, Dissertation no. 6881, ETH, Zurich.

Steel and Torrie

Steel and Torrie (1960) *Principles and Procedures of Statistics*, McGraw-Hill.

Stephens

Stephens, M.A. (1974), EDF statistics for goodness of fit and some comparisons, *Journal of the American Statistical Association*, **69**, 730−737.

Stirling

Stirling, W.D. (1981), Least squares subject to linear constraints, *Applied Statistics*, **30**, 204–212. (See correction, p. 357.)

Stoline

Stoline, Michael R. (1981), The status of multiple comparisons: simultaneous estimation of all pairwise comparisons in one-way ANOVA designs, *The American Statistician*, **35**, 134−141.

Strecok

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144−158.

Tanner and Wong

Tanner, Martin A., and Wing H. Wong (1983), The estimation of the hazard function from randomly censored data by the kernel method, *Annals of Statistics*, **11**, 989–993.

Tanner, Martin A., and Wing H. Wong (1984), Data-based nonparametric estimation of the hazard function with applications to model diagnostics and exploratory analysis, *Journal of the American Statistical Association*, **79**, 123–456.

Taylor and Thompson

Taylor, Malcolm S., and James R. Thompson (1986), Data based random number generation for a multivariate distribution via stochastic simulation, *Computational Statistics & Data Analysis*, **4**, 93−101.

Tezuka

Tezuka, S. (1995), *Uniform Random Numbers: Theory and Practice*. Academic Publishers, Boston.

Thompson

Thompson, James R, (1989), *Empirical Model Building*, John Wiley & Sons, New York.

Tucker and Lewis

Tucker, Ledyard and Charles Lewis (1973), A reliability coefficient for maximum likelihood factor analysis, *Psychometrika*, **38**, 1−10.

Tukey

Tukey, John W. (1962), The future of data analysis, *Annals of Mathematical Statistics,* **33**, 1−67.

Velleman and Hoaglin

Velleman, Paul F. and David C. Hoaglin (1981), *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.

Verdooren

Verdooren, L. R. (1963), Extended tables of critical values for Wilcoxon's test statistic, *Biometrika*, **50**, 177−186.

Wallace

Wallace, D.L. (1959), Simplified Beta-approximations to the Kruskal-Wallis H-test, *Journal of the American Statistical Association*, **54**, 225−230.

Weisberg

Weisberg, S. (1985), *Applied Linear Regression*, 2d ed., John Wiley & Sons, New York.

Woodfield

Woodfield, Terry J. (1990), Some notes on the Ljung-Box portmanteau statistic, *American Statistical Association 1990 Proceedings of the Statistical Computing Section*, 155–160.

Yates, F.

Yates, F. (1936) A new method of arranging variety trials involving a large number of varieties. *Journal of Agricultural Science*, **26**, 424-455.

Appendix B: Alphabetical Summary of Routines

Function	Purpose Statement	Page
`anova_balanced`	Analyzes a balanced complete experimental design for a fixed, random, or mixed model.	256
`anova_factorial`	Analyzes a balanced factorial design with fixed effects.	239
`anova_nested`	Analyzes a completely nested random model with possibly unequal numbers in the subgroups.	247
`anova_oneway`	Analyzes a one-way classification model.	230
`arma`	Computes least-square estimates of parameters for an ARMA model.	517
`arma_forecast`	Computes forecasts and their associated probability limits for an ARMA model.	527
`autocorrelation`	Computes the sample autocorrelation function of a stationary time series.	541
`beta`	Evaluates the complete beta function.	901
`beta_cdf`	Evaluates the beta probability distribution function.	730
`beta_incomplete`	Evaluates the real incomplete beta function.	903
`beta_inverse_cdf`	Evaluates the inverse of the beta distribution function.	731
`binomial_cdf`	Evaluates the binomial distribution function.	720
`binomial_coefficient`	Evaluates the binomial coefficient.	900
`binomial_pdf`	Evaluates the binomial probability function.	722
`bivariate_normal_cdf`	Evaluates the bivariate normal distribution function.	732
`box_cox_transform`	Performs a Box-Cox transformation.	537
`categorical_glm`	Analyzes categorical data using logistic, Probit, Poisson, and other generalized linear models.	425
`chi_squared_cdf`	Evaluates the chi-squared distribution function.	734

Function	Purpose Statement	Page
`chi_squared_inverse_cdf`	Evaluates the inverse of the chi-squared distribution function.	736
`chi_squared_test`	Performs a chi-squared goodness-of-fit test.	482
`cluster_hierarchical`	Performs a hierarchical cluster analysis given a distance matrix.	590
`cluster_k_means`	Performs a *K*-means (centroid) cluster analysis.	598
`cluster_number`	Computes cluster membership for a hierarchical cluster tree.	594
`cochran_q_test`	Performs a Cochran *Q* test for related observations.	472
`contingency_table`	Performs a chi-squared analysis of a two-way contingency table.	404
`continuous_table_setup`	Sets up table to generate pseudorandom numbers from a general continuous distribution.	812
`covariances`	Computes the sample variance-covariance or correlation matrix.	185
`cox_stuart_trends_test`	Performs the Cox and Stuart' sign test for trends in location and dispersion.	452
`crd_factorial`	Analyzes data from balanced and unbalanced completely randomized experiments.	267
`crosscorrelation`	Computes the sample cross-correlation function of two stationary time series	546
`ctime`	Returns the number of CPU seconds used.	911
`data_sets`	Retrieves a commonly analyzed data set.	890
`difference`	Differences a seasonal or nonseasonal time series.	532
`discrete_table_setup`	Sets up a table to generate pseudorandom numbers from a general discrete distribution.	781
`discriminant_analysis`	Performs discriminant function analysis.	628
`dissimilarities`	Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.	586
`error_code`	Returns the code corresponding to the error message from the last function called.	885
`error_options`	Sets various error handling options.	879
`exact_enumeration`	Computes exact probabilities in a two-way contingency table, using the total enumeration method.	417
`exact_network`	Computes exact probabilities in a two-way contingency table using the network algorithm.	419

Function	Purpose Statement	Page
`F_cdf`	Evaluates the *F* distribution function.	742
`F_inverse_cdf`	Evaluates the inverse of the *F* distribution function.	744
`factor_analysis`	Extracts initial factor-loading estimates in factor analysis.	609
`faure_next_point`	Computes a shuffled Faure sequence	856
`friedmans_test`	Performs Friedman's test for a randomized complete block design.	467
`gamma`	Evaluates the real gamma functions.	905
`gamma_cdf`	Evaluates the gamma distribution function.	745
`gamma_incomplete`	Evaluates the incomplete gamma function.	907
`gamma_inverse_cdf`	Evaluates the inverse of the gamma distribution function.	747
`garch`	Computes estimates of the parameters of a GARCH(p, q) model	566
`homogeneity`	Conducts Bartlett's and Levene's tests of the homogeneity of variance assumption in analysis of variance.	378
`hypergeometric_cdf`	Evaluates the hypergeometric distribution function.	723
`hypergeometric_pdf`	Evaluates the hypergeometric probability function.	725
`hypothesis_partial`	Constructs a completely testable hypothesis.	96
`hypothesis_scph`	Sums of cross products for a multivariate hypothesis.	101
`hypothesis_test`	Tests for the multivariate linear hypothesis.	106
`k_trends_test`	Performs k-sample trends test against ordered alternatives.	475
`kalman`	Performs Kalman filtering and evaluates the likelihood function for the state-space model.	571
`kaplan_meier_estimates`	Computes Kaplan-Meier estimates of survival probabilities in stratified samples.	654
`kolmogorov_one`	Performs a Kolmogorov-Smirnov's one-sample test for continuos distributions.	494
`kolmogorov_two`	Performs a Kolmogorov-Smirnov's two-sample test	497
`kruskal_wallis_test`	Performs a Kruskal-Wallis's test for identical population medians.	465
`lack_of_fit`	Performs lack-of-fit test for an univariate time series or transfer function given the appropriate correlation function.	563

Function	Purpose Statement	Page
`latin_square`	Analyzes data from latin-square experiments.	288
`lattice`	Analyzes balanced and partially-balanced lattice experiments.	297
`life_tables`	Produces population and cohort life tables.	712
`Lnorm_regression`	Fits a multiple linear regression model using criteria other than least squares.	168
`log_beta`	Evaluates the log of the real beta function.	904
`log_gamma`	Evaluates the logarithm of the absolute value of the gamma function.	909
`machine (float)`	Returns information describing the computer's floating-point arithmetic.	888
`machine (integer)`	Returns integer information describing the computer's arithmetic.	886
`mat_mul_rect`	Computes the transpose of a matrix, a matrix-vector product, a matrix-matrix product, a bilinear form, or any triple product.	893
`multi_crosscorrelation`	Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.	552
`multiple_comparisons`	Performs Student-Newman-Keuls multiple comparisons test.	385
`multivar_normality_test`	Computes Mardia's multivariate measures of skewness and kurtosis and tests for multivariate normality.	501
`noether_cyclical_trend`	Performs the Noether's test for cyclical trend.	449
`non_central_chi_sq`	Evaluates the noncentral chi-squared distribution function.	738
`non_central_chi_sq_inv`	Evaluates the inverse of the noncentral chi-squared function.	740
`non_central_t_cdf`	Evaluates the noncentral Student's *t* distribution function.	754
`non_central_t_inv_cdf`	Evaluates the inverse of the noncentral Student's *t* distribution function.	757
`nonlinear_optimization`	Fits a nonlinear regression model using Powell's algorithm.	159
`nonlinear_regression`	Fits a nonlinear regression model.	149
`nonparam_hazard_rate`	Performs nonparametric hazard rate estimation using kernel functions and quasi-likelihoods.	703

Function	Purpose Statement	Page
`normal_cdf`	Evaluates the standard normal (Gaussian) distribution function.	748
`normal_inverse_cdf`	Evaluates the inverse of the standard normal (Gaussian) distribution function.	750
`normal_one_sample`	Computes statistics for mean and variance inferences using a sample from a normal population.	7
`normal_two_sample`	Computes statistics for mean and variance inferences using samples from two normal population.	11
`normality_test`	Performs a test for normality.	490
`output_file`	Sets the output file or the error message output file.	874
`page`	Sets or retrieves the page width or length.	867
`partial_autocorrelation`	Computes the sample partial autocorrelation function of a stationary time series.	560
`partial_covariances`	Computes partial covariances or partial correlations from the covariance or correlation matrix.	193
`permute_matrix`	Permutes the rows or columns of a matrix.	898
`permute_vector`	Rearranges the elements of a vector as specified by a permutation.	897
`poisson_cdf`	Evaluates the Poisson distribution function.	726
`poisson_pdf`	Evaluates the Poisson probability function.	728
`poly_prediction`	Computes predicted values, confidence intervals, and diagnostics after fitting a polynomial regression model.	140
`poly_regression`	Performs a polynomial least-squares regression.	132
`pooled_covariances`	Computes a pooled variance-covariance from the observations.	198
`principal_components`	Computes principal components.	603
`prop_hazard_gen_lin`	Analyzes time event data via the proportional hazards model.	660
`random_arma`	Generates pseudorandom ARMA process numbers.	831
`random_beta`	Generates pseudorandom numbers from a beta distribution.	786
`random_binomial`	Generates pseudorandom binomial numbers.	765
`random_cauchy`	Generates pseudorandom numbers from a Cauchy distribution.	788
`random_chi_squared`	Generates pseudorandom numbers from a chi-squared distribution.	789

Function	Purpose Statement	Page
`random_exponential`	Generates pseudorandom numbers from a standard exponential distribution.	791
`random_exponential_mix`	Generates pseudorandom mixed numbers from a standard exponential distribution.	792
`random_gamma`	Generates pseudorandom numbers from a standard gamma distribution.	794
`random_general_continuous`	Generates pseudorandom numbers from a general continuous distribution.	810
`random_general_discrete`	Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.	777
`random_geometric`	Generates pseudorandom numbers from a geometric distribution.	766
`random_GFSR_table_get`	Retrieves the current table used in the GFSR generator.	853
`random_GFSR_table_set`	Sets the current table used in the GFSR generator.	853
`random_hypergeometric`	Generates pseudorandom numbers from a hypergeometric distribution.	768
`random_logarithmic`	Generates pseudorandom numbers from a logarithmic distribution.	770
`random_lognormal`	Generates pseudorandom numbers from a lognormal distribution.	796
`random_multinomial`	Generates pseudorandom numbers from a multinomial distribution.	821
`random_mvar_from_data`	Generates pseudorandom numbers from a multivariate distribution determined from a given sample.	819
`random_neg_binomial`	Generates pseudorandom numbers from a negative binomial distribution.	772
`random_normal`	Generates pseudorandom numbers from a standard normal distribution using an inverse CDF method.	798
`random_normal_multivariate`	Generates pseudorandom numbers from a multivariate normal distribution.	815
`random_npp`	Generates pseudorandom numbers from a nonhomogeneous Poisson process.	835
`random_option`	Selects the uniform (0, 1) multiplicative congruential pseudorandom number generator.	845
`random_option_get`	Retrieves the uniform (0, 1) multiplicative congruential pseudorandom number generator.	846

Function	Purpose Statement	Page
`random_order_normal`	Generates pseudorandom order statistics from a standard normal distribution.	827
`random_order_uniform`	Generates pseudorandom order statistics from a uniform (0, 1) distribution	829
`random_orthogonal_matrix`	Generates a pseudorandom orthogonal matrix or a correlation matrix.	816
`random_permutation`	Generates a pseudorandom permutation.	839
`random_poisson`	Generates pseudorandom numbers from a Poisson distribution.	774
`random_sample`	Generates a simple pseudorandom sample from a finite population.	842
`random_sample_indices`	Generates a simple pseudorandom sample of indices.	840
`random_seed_get`	Retrieves the current value of the seed used in the IMSL random number generators.	847
`random_seed_set`	Initializes a random seed for use in the IMSL random number generators.	850
`random_sphere`	Generates pseudorandom points on a unit circle or `K`-dimensional sphere.	823
`random_stable`	Sets up a table to generate pseudorandom numbers from a general discrete distribution.	800
`random_student_t`	Generates pseudorandom Student's *t*.	802
`random_substream_seed_get`	Retrieves a seed for the congruential generators that do not do shuffling that will generate random numbers beginning 100,000 numbers farther along.	848
`random_table_get`	Retrieves the current table used in the shuffled generator.	852
`random_table_set`	Sets the current table used in the shuffled generator.	851
`random_table_twoway`	Generates a pseudorandom two-way table.	825
`random_triangular`	Generates pseudorandom numbers from a triangular distribution.	803
`random_uniform`	Generates pseudorandom numbers from a uniform (0, 1) distribution.	804
`random_uniform_discrete`	Generates pseudorandom numbers from a discrete uniform distribution.	775
`random_von_mises`	Generates pseudorandom numbers from a von Mises distribution.	806

Function	Purpose Statement	Page
`random_weibull`	Generates pseudorandom numbers from a Weibull distribution.	808
`randomness_test`	Performs a test for randomness.	505
`ranks`	Computes the ranks, normal scores, or exponential scores for a vector of observations.	36
`rcbd_factorial`	Analyzes data from balanced and unbalanced randomized complete-block experiments.	279
`regression`	Fits a multiple linear regression model using least squares.	64
`regression_prediction`	Computes predicted values, confidence intervals, and diagnostics after fitting a regression model.	85
`regression_selection`	Selects the best multiple linear regression models.	112
`regression_stepwise`	Builds multiple linear regression models using forward selection, backward selection or stepwise selection.	123
`regression_summary`	Produces summary statistics for a regression model given the information from the fit.	77
`regressors_for_glm`	Generates regressors for a general linear model.	56
`robust_covariances`	Computes a robust estimate of a covariance matrix and mean vector.	204
`sign_test`	Performs a sign test.	442
`simple_statistics`	Computes basic univariate statistics.	2
`sort_data`	Sorts observations by specified keys, with option to tally cases into a multi-way frequency table.	27
`split_plot`	Analyzes a wide variety of split-plot experiments with fixed, mixed or random factors.	316
`split_split_plot`	Analyzes data from split-split-plot experiments.	329
`strip_plot`	Analyzes data from strip-plot experiments.	345
`strip_split_plot`	Analyzes data from strip-split-plot experiments.	355
`survival_estimates`	Estimates using various parametric models.	697
`survival_glm`	Analyzes survival data using a generalized linear model.	673
`t_cdf`	Evaluates the Student's *t* distribution function.	751
`t_inverse_cdf`	Evaluates the inverse of the Student's *t* distribution function.	753
`table_oneway`	Tallies observations into one-way frequency table.	18
`table_twoway`	Tallies observations into a two-way frequency table.	22

Function	Purpose Statement	Page
`tie_statistics`	Computes tie statistics for a sample of observations.	458
`version`	Returns integer information describing the version of the library, license number, operating system, and compiler.	878
`wilcoxon_rank_sum`	Performs a Wilcoxon rank sum test.	460
`wilcoxon_sign_rank`	Performs a Wilcoxon sign rank test.	445
`write_matrix`	Prints a rectangular matrix (or vector) stored in contiguous memory locations.	861
`write_options`	Sets or retrieves an option for printing a matrix.	868
`yates`	Estimates missing observations in designed experiments using Yate's method.	390

Index