Last update: May 19, 2016

ProbDist

A Software Library of Probability Distributions and Goodness-of-Fit Statistics in ANSI C

Pierre L'Ecuyer and Richard Simard

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

This document describes a set of basic software utilities, implemented in ANSI C, developed in our simulation laboratory. It is part of a larger set of tools used for stochastic simulation and for testing random number generators. It provides procedures to compute densities, mass functions, distribution functions and their inverses, and reliability functions, for various continuous and discrete probability laws. It also offers a mechanism for collecting observational data and computing elementary statistics on it, and tools for performing and reporting different types of univariate goodness-of-fit tests.

Copyright

Contents

fmass

This module provides functions for computing the probability terms (or mass function) for some standard discrete distributions.

For certain distributions (e.g., the Poisson, binomial, and negative binomial), one can either recompute a probability term each time it is needed, or precompute tables that contain the probability terms and the distribution function, and then use these tables whenever a value is needed. The latter trades memory for speed and is recommended especially if the distribution function has to be computed several times for the same parameter(s). We describe how this works for the Poisson distribution. Things work similarly for the other distributions.

To compute a single Poisson probability from scratch, simply use `fmass_PoissonTerm1`. To precompute tables, one must first call `fmass_CreatePoisson` with the desired parameter value λ of the Poisson distribution. This will precompute and store the non-negligible probability terms $f(s)$ (those that exceed `fmass_Epsilon`) in a table, and the cumulative distribution function

$$F(x) = \sum_{s=0}^{x} f(s)$$

for the corresponding values of x in a second table. In fact, that second table will contain $F(x)$ when $F(x) \leq 1/2$ and $1 - F(x)$ when $F(x) > 1/2$. These tables are kept in a structure of type `fmass_INFO` which can be deleted by calling `fmass_DeletePoisson`. Any value of the mass, distribution, complementary distribution, or inverse distribution function can be obtained from this structure by calling `fmass_Poisson2`, `fdist_Poisson2`, `fbar_Poisson2`, or `finv_Poisson2`, respectively. As a rule of thumb, creating tables and using `fdist_Poisson2` is faster than just using `fdist_Poisson1` as soon as two or three calls are made to this function, unless λ is large. (If λ is very large, the tables are not created because they would take too much space, and the functions with suffix `_Poisson2` automatically call those with suffix `_Poisson1` instead.)

Types

```
struct fmass_INFO_T;

typedef struct fmass_INFO_T *fmass_INFO;
```

Type of structure used to store precomputed discrete distributions.

The Poisson distribution

`double fmass_PoissonTerm1 (double lambda, long s);`

Computes and returns the value of the Poisson probability

$$f(s) = \frac{e^{-\lambda}\lambda^s}{s!} \tag{1}$$

for $\lambda =$ `lambda`. If one has to call this function several times with the same λ, where λ is not too large, then it is more efficient to use `fmass_PoissonTerm2`. Restriction: $\lambda > 0$.

`fmass_INFO fmass_CreatePoisson (double lambda);`

Creates and returns a structure that contains the mass and distribution functions for the Poisson distribution with parameter `lambda` $= \lambda$, which are computed and stored in dynamic arrays inside that structure. Such a structure is needed for calling `fmass_PoissonTerm2`, `fdist_Poisson2`, `fbar_Poisson2`, or `finv_Poisson2`. It can be deleted by calling the procedure `fmass_DeletePoisson`. Restriction: $\lambda > 0$.

`void fmass_DeletePoisson (fmass_INFO W);`

Deletes the structure `W` created previously by `fmass_CreatePoisson`.

`double fmass_PoissonTerm2 (fmass_INFO W, long s);`

Returns the Poisson probability (1) from the structure `W`, which must have been created previously by calling `fmass_CreatePoisson` with the desired λ.

The binomial distribution

`double fmass_BinomialTerm3 (long n, double p, long s);`

Computes and returns the binomial term

$$f(s) = \binom{n}{s} p^s (1-p)^{n-s} = \frac{n!}{s!(n-s)!}\, p^s (1-p)^{n-s}, \tag{2}$$

where p is an arbitrary real number. In the case where $0 \le p \le 1$, the returned value is a probability term for the binomial distribution. Restriction: $0 \le s \le n$.

`double fmass_BinomialTerm1 (long n, double p, double q, long s);`

Computes and returns the binomial term

$$f(s) = \binom{n}{s} p^s q^{n-s} = \frac{n!}{s!(n-s)!}\, p^s q^{n-s}, \tag{3}$$

where p and q are arbitrary real numbers. In the case where $0 \leq p \leq 1$ and $q = 1 - p$, the returned value is a probability term for the binomial distribution. Restriction: $0 \leq s \leq n$.

`double fmass_BinomialTerm4 (long n, double p, double p2, long s);`

Computes and returns the binomial term

$$f(s) = \binom{n}{s} p^s (1-p_2)^{n-s} = \frac{n!}{s!(n-s)!}\, p^s (1-p_2)^{n-s}, \tag{4}$$

where p and p_2 are real numbers in $[0, 1]$. In the case where $p_2 = p$, the returned value is a probability term for the binomial distribution. If p_2 is small, this function is more precise than `fmass_BinomialTerm1`. Restriction: $0 \leq s \leq n$.

`fmass_INFO fmass_CreateBinomial (long n, double p, double q);`

Creates and returns a structure that contains binomial terms (3) for $0 \leq s \leq n$, and the corresponding cumulative function. If $0 \leq p = 1 - q \leq 1$, these are the probabilities and the distribution function of a binomial random variable. The values are computed and stored in dynamic arrays. Such a structure is needed for calling `fmass_BinomialTerm2`, `fdist_Binomial2`, `fbar_Binomial2`, or `finv_Binomial2`. It can be deleted by calling `fmass_DeleteBinomial`. This function is more general than the binomial probability distribution as it computes the binomial terms when $p + q \neq 1$, and even when p or q are negative. However in this case, the cumulative terms will be meaningless and only the mass terms `fmass_BinomialTerm2` are computed.

`void fmass_DeleteBinomial (fmass_INFO W);`

Deletes the structure `W` created previously by `fmass_CreateBinomial`.

`double fmass_BinomialTerm2 (fmass_INFO W, long s);`

Returns the value of the binomial term (3) from the structure `W`, which must have been created previously by `fmass_CreateBinomial` with the desired parameters.

The negative binomial distribution

`double fmass_NegaBinTerm1 (long n, double p, long s);`

Computes and returns the value of the negative binomial probability term

$$f(s) = \binom{n+s-1}{s} p^n (1-p)^s, \tag{5}$$

which can be interpreted as the probability of having s failures before the nth success in a sequence of independent Bernoulli trials with success probability p. Restrictions: $n > 0$, $0 \leq p \leq 1$, and $s \geq 0$.

`fmass_INFO fmass_CreateNegaBin (long n, double p);`

Creates and returns a structure that contains the probability terms (5) and the distribution functions for the negative binomial distribution with parameter n and p. Such a structure is needed for calling `fmass_NegaBinTerm2`, `fdist_NegaBin2`, `fbar_NegaBin2`, or `finv_NegaBin2`. It can be deleted by calling `fmass_DeleteNegaBin`. Restrictions: $0 \leq p \leq 1$ and $n > 0$.

`void fmass_DeleteNegaBin (fmass_INFO W);`

Deletes the structure `W` created previously by `fmass_CreateNegaBin`.

`double fmass_NegaBinTerm2 (fmass_INFO W, long s);`

Returns the negative binomial probability (5) from the structure `W`, which must have been created previously by calling `fmass_CreateNegaBin` with the desired parameters.

fdist

This module provides procedures to compute (or approximate) the distribution functions of several standard types of random variables and of certain goodness-of-fit test statistics. Recall that the distribution function of a continuous random variable X with density f is

$$F(x) = P[X \leq x] = \int_{-\infty}^{x} f(x)dx \tag{6}$$

while that of a discrete random variable X with mass function f over the set of integers is

$$F(x) = P[X \leq x] = \sum_{s=-\infty}^{x} f(x). \tag{7}$$

All the procedures in this module return $F(x)$ for some probability distribution.

Most distributions are implemented only in standardized form here, i.e., with the location parameter set to 0 and the scale parameter set to 1. To shift the distribution by x_0 and rescale by c, it suffices to replace x by $(x - x_0)/c$ in the argument when calling the function.

For some of the discrete distributions, the value of $F(x)$ can be simply recovered from a table that would have been previously constructed; see the module `fmass` for the details. This permits one to avoid recomputing the sums.

```
#include <testu01/gdef.h>
#include <testu01/fmass.h>
```

Continuous distributions

`double fdist_Unif (double x);`

Returns x for $x \in [0, 1]$, returns 0 for $x < 0$, and returns 1 for $x > 1$. This is the uniform distribution function over $[0, 1]$.

`double fdist_Expon (double x);`

Returns

$$F(x) = 1 - e^{-x} \tag{8}$$

for $x > 0$, and 0 for $x < 0$. This is the standard exponential distribution [19] with mean 1.

`double fdist_Weibull (double alpha, double x);`

Returns

$$F(x) = 1 - e^{-x^{\alpha}}, \tag{9}$$

for $x > 0$, and 0 for $x \leq 0$. This is the standard Weibull distribution function [19] with shape parameter α. Restriction: $\alpha > 0$.

`double fdist_ExtremeValue (double x);`

Returns

$$F(x) = e^{-e^{-x}}, \tag{10}$$

the standard extreme value distribution function [20].

`double fdist_Logistic (double x);`

Returns

$$F(x) = \frac{1}{1+e^{-x}} = \frac{1}{2}\left(1 + \tanh\left(\frac{x}{2}\right)\right), \tag{11}$$

the standard logistic distribution function [20].

`double fdist_Pareto (double c, double x);`

Returns

$$F(x) = 1 - \frac{1}{x^c}, \tag{12}$$

for $x \geq 1$ and 0 for $x < 1$. This is the standard Pareto distribution function [19]. Restriction: $c > 0$.

`double fdist_Normal1 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the approximation given in [21, page 90]. This distribution is less precise than `fdist_Normal2` in the lower tail, as it will not compute probabilities smaller than `DBL_EPSILON`.

`double fdist_Normal2 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the Chebyshev approximation proposed in [36], which gives 15 decimals of precision nearly everywhere. This function is 1.5 times slower than `fdist_Normal1`.

```
#ifdef HAVE_ERF
   double fdist_Normal3 (double x);
#endif
```

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the `erf` function from the standard Unix C library. The macro `HAVE_ERF` from `mylib/gdef` must be defined. On some machines, this function is twice as fast as `fdist_Normal1`.

`double fdist_Normal4 (double x);`

Returns an approximation of $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses Marsaglia's et al [31] fast method with tables lookup. Returns 15 decimal digits of precision. This function is as fast as `fdist_Normal1` (no more no less).

`double fdist_BiNormal1 (double x, double y, double rho, int ndig);`

Returns the value u of the standard bivariate normal distribution, given by

$$\begin{aligned} u &= \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} e^{-T} dy\, dx \\ T &= \frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}, \end{aligned} \tag{13}$$

where $\rho =$ `rho` is the correlation between x and y, and `ndig` is the number of decimal digits of accuracy. The code was translated from the Fortran program written by T. G. Donnelly [11] and copyrighted by the ACM (see `http://www.acm.org/publications/policies/copyright_policy`). The absolute error is expected to be smaller than 10^{-d}, where $d =$ `ndig`. Restriction: `ndig` ≤ 15.

`double fdist_BiNormal2 (double x, double y, double rho);`

Returns the value of the standard bivariate normal distribution as defined in (13) above. It was translated directly from the Matlab code written by Alan Genz and available from his web page (the code is copyrighted by Alan Genz, and is included in this package with the kind permission of its author). The algorithm, described in [16], is a modified form of the algorithm proposed in [12]. The program's accuracy results in an absolute error less than $5 \cdot 10^{-16}$.

`double fdist_LogNormal (double mu, double sigma, double x);`

Returns the lognormal distribution function, defined by [19]

$$F(x) = \Phi\left(\frac{\ln(x) - \mu}{\sigma}\right) \tag{14}$$

for $x > 0$ and 0 for $x \leq 0$, where Φ is the standard normal distribution. Restriction: $\sigma > 0$.

`double fdist_JohnsonSB (double alpha, double beta, double a, double b, double x);`

Returns the Johnson JSB distribution function [25]:

$$F(x) = \Phi\left(\alpha + \beta \ln\left(\frac{x-a}{b-x}\right)\right), \tag{15}$$

where Φ is the standard normal distribution. Restrictions: $\beta > 0$, $a < b$, and $a \leq x \leq b$.

`double fdist_JohnsonSU (double alpha, double beta, double x);`

Returns the Johnson JSU distribution function [25]:

$$F(x) = \Phi\left(\alpha + \beta \ln\left(x + \sqrt{x^2+1}\right)\right) \tag{16}$$

where Φ is the standard normal distribution. Restriction: $\beta > 0$.

`double fdist_ChiSquare1 (long k, double x);`

Returns an approximation of the chi-square distribution function with k degrees of freedom, which is a special case of the gamma distribution, with shape parameter $k/2$ and scale parameter $1/2$. Uses the approximation given in [21, p.116] for $k \leq 1000$, and the normal approximation for $k > 1000$. Gives no more than 4 decimals of precision for $k > 1000$.

`double fdist_ChiSquare2 (long k, int d, double x);`

Returns an approximation of the chi-square distribution function with k degrees of freedom, by calling `fdist_Gamma (k/2, d, x/2)`. The function will do its best to return d decimals digits of precision (but there is no guarantee). For k not too large (e.g., $k \leq 1000$), d gives a good idea of the precision attained. Restrictions: $k > 0$ and $0 < d \leq 15$.

`double fdist_Student1 (long n, double x);`

Returns the approximation of [21, p.96] for the *Student-t* distribution function with n degrees of freedom, whose density is

$$f(x) = \frac{\Gamma\left((n+1)/2\right)}{\Gamma(n/2)\sqrt{\pi n}} \left[1 + \frac{x^2}{n}\right]^{-(n+1)/2}, \qquad -\infty < x < \infty. \tag{17}$$

Gives at least 12 decimals of precision for $n \leq 10^3$, and at least 10 decimals for $10^3 < n \leq 10^5$. Restriction: $n > 0$.

`double fdist_Student2 (long n, int d, double x);`

Returns an approximation of the *Student-t* distribution function with n degrees of freedom, with density (17). Uses the relationship (see [19])

$$2F(x) = \begin{cases} I_{n/2,1/2}(n/(n+x^2)) & \text{for } x < 0, \\ I_{1/2,n/2}(x^2/(n+x^2)) & \text{for } x \geq 0, \end{cases} \tag{18}$$

where $I_{p,q}$ is the *beta* distribution function with parameters p and q (also called the incomplete *beta* ratio) defined in (21), which is approximated by calling `fdist_Beta`. The function tries to return d decimals digits of precision (but there is no guarantee). Restrictions: $n > 0$ and $0 < d \leq 15$.

`double fdist_Gamma (double a, int d, double x);`

Returns an approximation, based on [4], of the *gamma* distribution function with parameter a, whose density is

$$f(x) = \frac{x^{a-1}e^{-x}}{\Gamma(a)}, \tag{19}$$

for $x \geq 0$, where Γ is the gamma function, defined by

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx. \tag{20}$$

The function tries to return d decimals digits of precision. For a not too large (e.g., $a \leq 1000$), d gives a good idea of the precision attained. For $a \geq 100000$, uses a normal approximation given in [32]. Restrictions: $a > 0$ and $0 < d \leq 15$.

`double fdist_Beta (double p, double q, int d, double x);`

Returns an approximation of

$$F(x) = I_{p,q}(x) = \int_0^x \frac{t^{p-1}(1-t)^{q-1}}{B(p,q)} dt, \tag{21}$$

the *beta* distribution function with parameters p and q, evaluated at $x \in [0,1]$, where $B(p,q)$ is the *beta* function defined by

$$B(p,q) = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)}, \tag{22}$$

where $\Gamma(x)$ is the Gamma function defined in (20). For $\max(p,q) \leq 1000$, use a recurrence relation in p and q for `fdist_Beta`, given in [14, 15]. Else, if $\min(p,q) \leq 30$, use an approximation due to Bol'shev [27]. Otherwise, use a normal approximation [32]. The function tries to return d decimals digits of precision. For $d \leq 13$, when the normal approximation is *not* used, d gives a good idea of the precision attained. Restrictions: $p > 0$, $q > 0$, $x \in [0,1]$ and $0 < d \leq 15$.

`double fdist_BetaSymmetric (double p, double x);`

Returns an approximation of the symmetrical *beta* distribution function $F(x)$ with parameters $p = q$ as defined in (21). Uses four different hypergeometric series (for the four cases x close to 0 and $p \leq 1$, x close to 0 and $p > 1$, x close to 1/2 and $p \leq 1$, and x close to 1/2 and $p > 1$) to compute $F(x)$. For $p > 100000$, uses a normal approximation given in [32]. Restrictions: $p > 0$ and $x \in [0,1]$.

`double fdist_KSPlus (long n, double x);`

Returns $p = P[D_n^+ \leq x]$, where

$$D_n^+ = \sup_{-\infty < s < \infty} [\hat{F}_n(s) - F(s)]^+ \tag{23}$$

is the positive Kolmogorov-Smirnov statistic for a sample of size n whose empirical distribution function is $\hat{F}_n$, under the hypothesis that the observations follow a continuous distribution function F. (Recall that x^+ represents $\max(0,x)$, the positive part of x.) The statistic

$$D_n^- = \sup_{-\infty < s < \infty} [F(s) - \hat{F}_n(s)]^+ \tag{24}$$

has the same distribution as D_n^+. Procedures for computing these statistics are availables in module `gofs`. The distribution function of D_n^+ can be approximated via the following expressions:

$$P[D_n^+ \le x] = 1 - x \sum_{i=0}^{\lfloor n(1-x) \rfloor} \binom{n}{i} \left(\frac{i}{n} + x\right)^{i-1} \left(1 - \frac{i}{n} - x\right)^{n-i} \tag{25}$$

$$= x \sum_{j=0}^{\lfloor nx \rfloor} \binom{n}{j} \left(\frac{j}{n} - x\right)^{j} \left(1 - \frac{j}{n} + x\right)^{n-j-1} \tag{26}$$

$$\approx 1 - e^{-2nx^2} \left[1 - \frac{2x}{3}\left(1 - x\left(1 - \frac{2nx^2}{3}\right)\right.\right.$$
$$\left.\left. - \frac{2}{3n}\left(\frac{1}{5} - \frac{19nx^2}{15} + \frac{2n^2x^4}{3}\right)\right) + O(n^{-2})\right]. \tag{27}$$

Formula (25) and (26) can be found in [13], equations (2.1.12) and (2.1.16), while (27) can be found in [9]. Formula (26) contains less terms than (25) when $x < 0.5$, but becomes numerically unstable as nx increases. The approximation (27) is simpler to compute and excellent when nx is large. Our implementation uses (26) when $nx < 6.5$, (25) when $nx \ge 6.5$ and $n \le 4000$, and (27) when $nx \ge 6.5$ and $n > 4000$.

`double fdist_KS1 (long n, double x);`

Returns $u = P[D_n \le x]$ where $D_n = \max\{D_n^+, D_n^-\}$ is the two-sided Kolmogorov-Smirnov statistic [7] for a sample of size n, and D_n^+ and D_n^- are defined in (23) and (24). This method uses Pomeranz's recursion formula [8, 34] for $n \le 400$, which return at least 13 decimal digits of precision. It uses the Pelz-Good asymptotic expansion [33] in the central part of the range for $n > 400$ and returns at least 6 decimal digits of precision everywhere for $400 < n \le 4000$. For $n > 4000$, it returns at least 2 decimal digits of precision for all $u > 10^{-22}$, and at least 5 decimal digits of precision for all $u > 10^{-7}$. For a given n, the precision increases as x increases. This method is much faster than `fdist_KS2` for moderate or large n.

`double fdist_KS2 (long n, double x);`

Another version of the Kolmogorov-Smirnov distribution $P[D_n \le x]$, using Durbin's matrix formula [13]. It is astronomically slow for large n. According to its authors [30], it should return at least 7 decimal digits of precision.

`double fdist_KSPlusJumpOne (long n, double a, double x);`

Similar to `fdist_KSPlus` but for the case where the distribution function F has a jump of size a at a given point x_0, is zero at the left of x_0, and is continuous at the right of x_0. The Kolmogorov-Smirnov statistic is defined in that case as

$$D_n^+(a) = \sup_{a \le u \le 1} \left(\hat{F}_n(F^{-1}(u)) - u\right) = \max_{\lfloor 1+an \rfloor \le j \le n} \left(j/n - F(V_{(j)})\right). \tag{28}$$

where $V_{(1)}, \ldots, V_{(n)}$ are the observations sorted by increasing order. The procedure returns an approximation of $P[D_n^+(a) \le x]$ computed via

$$P[D_n^+(a) \le x] = 1 - x \sum_{i=0}^{\lfloor n(1-a-x) \rfloor} \binom{n}{i} \left(\frac{i}{n} + x\right)^{i-1} \left(1 - \frac{i}{n} - x\right)^{n-i} \tag{29}$$

$$= x \sum_{j=0}^{\lfloor n(a+x) \rfloor} \binom{n}{j} \left(\frac{j}{n} - x\right)^{j} \left(1 - \frac{j}{n} + x\right)^{n-j-1}. \tag{30}$$

The current implementation uses formula (30) when $n(x+a) < 6.5$ and $x + a < 0.5$, and uses (29) when $nx \ge 6.5$ or $x + a \ge 0.5$. Restriction: $0 < a < 1$.

`double fdist_CramerMises (long n, double x);`

Returns an approximation of $P[W_n^2 \le x]$, where W_n^2 is the Cramér von Mises statistic (see [38, 39, 2, 22]) defined in (43), for a sample of independent uniforms over $(0, 1)$. The approximation is based on the distribution function of $W^2 = \lim_{n\to\infty} W_n^2$, which has the following series expansion derived by Anderson and Darling [2]:

$$P(W^2 \le x) = \frac{1}{\pi\sqrt{x}} \sum_{j=0}^{\infty} (-1)^j \binom{-1/2}{j} \sqrt{4j+1}\ \exp\left\{-\frac{(4j+1)^2}{16x}\right\} \mathrm{K}_{1/4}\left(\frac{(4j+1)^2}{16x}\right), \tag{31}$$

where K_ν is the modified Bessel function of the second kind. To correct for the deviation between $P(W_n^2 \le x)$ and $P(W^2 \le x)$, we add a correction in $1/n$, obtained empirically by simulation. For $n = 10$, 20, 40, the error is less than 0.002, 0.001, and 0.0005, respectively, while for $n \ge 100$ it is less than 0.0005. For $n \to \infty$, we estimate that the procedure returns at least 6 decimal digits of precision. For $n = 1$, the procedure computes the exact distribution: $P(W_1^2 \le x) = 2\sqrt{x - 1/12}$ for $1/12 \le x \le 1/3$.

`double fdist_WatsonG (long n, double x);`

Returns an approximation of $P[G_n \le x]$, where G_n is the Watson statistic defined in (44), for a sample of independent uniforms over $(0, 1)$. The approximation is computed in a similar way as for `fdist_CramerMises`. To implement this procedure, a table of the values of $g(x) = \lim_{n\to\infty} P[G_n \le x]$ and of its derivative was first computed by numerical integration. For $x \le 1.5$, the procedure uses this table with cubic spline interpolation. For $x > 1.5$, it uses the empirical curve $g(x) = 1 - e^{19-20x}$. A correction of order $1/\sqrt{n}$, obtained empirically from 10^7 simulation runs with $n = 256$ and also implemented as an interpolation table with an exponential tail, is then added. The absolute error is estimated to be less than 0.01, 0.005, 0.002, 0.0008, 0.0005, 0.0005, 0.0005 for $n = 16$, 32, 64, 128, 256, 512, 1024, respectively. For the trivial case $n = 1$, always returns 0.5.

`double fdist_WatsonU (long n, double x);`

Returns $P[U^2 \le x]$, where U^2 is the Watson statistic defined in (46) in the limit when $n \to \infty$, for a sample of independent uniforms over $(0, 1)$. Only this limiting distribution (when $n \to \infty$)

is implemented. It is given by

$$P(U^2 \le x) \;=\; 1 + 2\sum_{j=1}^{\infty}(-1)^j e^{-2j^2\pi^2 x} \tag{32}$$

This sum converges extremely fast except for small x, where alternating successive terms give rise to numerical instability. But with the Poisson summation formula [24], the sum can be transformed to

$$P(U^2 \le x) \;=\; \sqrt{\frac{2}{\pi x}}\;\sum_{j=0}^{\infty} e^{-(2j+1)^2/8x} \tag{33}$$

which can be used for small x. The current implementation uses (32) for $x > 0.15$, and (33) for $x \le 0.15$. The absolute difference between the returned value and $P[U_n^2 \le x]$ is estimated to be less than 0.01 for $n \ge 8$. For the trivial case $n = 1$, always returns 0.5.

`double fdist_AndersonDarling (long n, double x);`

Returns $F_n(x) = P[A_n^2 \le x]$, where A_n^2 is the Anderson-Darling statistic [2] defined in (47), for a sample of independent uniforms over $(0, 1)$. The approximation is computed similarly as for `fdist_CramerMises`. To implement this procedure, an interpolation table of the values of $F(x) = \lim_{n\to\infty} P[A_n^2 \le x]$ was first computed by numerical integration. Then a linear correction in $1/n$, obtained by simulation, was added. For $x \le 5$, the procedure approximates $F_n(x) = P[A_n^2 \le x]$ by interpolation. For $5 < x < 10$, it uses the empirical curve $F_n(x) \approx 1 - e^{-1.06x-0.56} - e^{-1.06x-1.03}/n$, which includes the empirical correction in $1/n$. The absolute error on $F_n(x)$ is estimated to be less than 0.001 for $n > 6$ except far in the tails. For $n = 2$, 3, 4, 6, it is estimated to be less than 0.04, 0.01, 0.005, 0.002, respectively. In the lower tail ($x < 0.2$), the approximation (3.6) of Sinclair and Spurr [37]

$$F(x) = 1 - \frac{1}{1 + \exp\left(1.784 + 0.9936x + \frac{0.03287}{x} - \frac{(2.018+0.2029/x)}{\sqrt{x}}\right)}$$

is used without correction for finite n. In the far upper tail ($x > 10$), the approximation (3.5) of Sinclair and Spurr [37]

$$F(x) = 1 - \frac{1.732\exp(-x)}{\sqrt{\pi x}}$$

is used without correction for finite n. For $n = 1$, the procedure returns the exact value, $F_1(x) = \sqrt{1 - 4e^{-x-1}}$ for $x \ge \ln(4) - 1$.

`double fdist_AndersonDarling2 (long n, double x);`

Returns the value of the Anderson-Darling distribution at x for a sample of n independent uniforms over $(0, 1)$ using Marsaglia's and al. algorithm [29]. First the limiting distribution for $n \to \infty$ is computed to within 6-digit accuracy according to the authors. Then an empirical correction obtained by simulation is added for finite n. For $n = 1$, the procedure returns the exact value, $F_1(x) = \sqrt{1 - 4e^{-x-1}}$ for $x \ge \ln(4) - 1$.

Discrete distributions

`double fdist_Geometric (double p, long s);`

Returns

$$F(s) = \sum_{j=0}^{s} p\,(1-p)^j \;=\; 1-(1-p)^{s+1}, \tag{34}$$

the distribution function of a geometric random variable with parameter p, evaluated at s. Restriction: $0 \le p \le 1$.

`double fdist_Poisson1 (double lambda, long s);`

Returns

$$F_\lambda(s) = e^{-\lambda} \sum_{j=0}^{s} \frac{\lambda^j}{j!}, \tag{35}$$

the Poisson distribution function with parameter $\lambda =$ `lambda`, evaluated at s. In the cases where the Poisson distribution must be computed more than once with the same λ, it is more efficient to use `fdist_Poisson2` instead of `fdist_Poisson1`. Restriction: $\lambda > 0$.

`double fdist_Poisson2 (fmass_INFO W, long s);`

Returns the Poisson distribution function (35) from the structure `W`, which must have been created previously by calling `fmass_CreatePoisson` with the desired λ.

`double fdist_Binomial1 (long n, double p, long s);`

Returns

$$F(s) = \sum_{j=0}^{s} \binom{n}{j} p^j (1-p)^{n-j}, \tag{36}$$

the distribution function of a binomial random variable with parameters n and p, evaluated at s. When the binomial distribution has to be computed more than once with the same parameters n and p, it is more efficient to use `fdist_Binomial2` instead of `fdist_Binomial1`, unless n is very large (e.g., $n > 10^5$). Restrictions: $0 \le p \le 1$ and $n \ge 0$.

`double fdist_Binomial2 (fmass_INFO W, long s);`

Returns the binomial distribution function (36) from the structure `W`, which must have been created previously by calling `fmass_CreateBinomial` with the desired values of n and p.

`double fdist_NegaBin1 (long n, double p, long s);`

Returns

$$F(s) = \sum_{j=0}^{s} \binom{n+j-1}{j} p^n (1-p)^j, \tag{37}$$

the distribution function of a negative binomial random variable with parameters n and p, evaluated at s. If this distribution has to be computed more than once with the same n and p,

it is more efficient to use `fdist_NegaBin2` instead of `fdist_NegaBin1`, unless n is very large. Restrictions: $n \geq 0$ and $0 \leq p \leq 1$.

```
double fdist_NegaBin2 (fmass_INFO W, long s);
```

Returns the negative binomial distribution function (37) from the structure `W`, which must have been created previously by calling `fmass_CreateBinomial` with the desired values of n and p.

```
double fdist_Scan (long N, double d, long m);
```

Returns $F(m)$, the distribution function of the scan statistic with parameters N and d, evaluated at m. For a description of this statistic and its distribution, see `fbar_Scan`, which computes its complementary distribution $\bar{F}(m) = 1 - F(m-1)$.

wdist

This module provides wrappers functions that are needed because the parameter of type `wdist_CFUNC` in `gofw_ActiveTests1` and in `gofs_ContUnifTransform`, for example, is not type-compatible with the distribution functions provided in `fdist`, since the different distributions take a different number of arguments.

```
#include <testu01/fmass.h>
```

Types

```
typedef double (*wdist_CFUNC) (double [], double);
```

A generic *continuous* distribution function with an arbitrary number of parameters given in the first argument. The second argument is the point x at which the function is evaluated.

```
typedef double (*wdist_DFUNC) (fmass_INFO, long);
```

A generic *discrete* distribution function over the set of integers. The first argument contains the parameters of the function and possibly precomputed tables of values of the function. The second argument is the point x at which the function is evaluated.

Wrap-up functions

```
double wdist_Normal (double Par[], double x);
```

Wrapper function for the standard normal distribution, needed for compatibility with the type `wdist_CFUNC` used as a parameter in certain functions such as `gofw_ActiveTests1`, etc. Returns $\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. `Par` is unused.

```
double wdist_ChiSquare (double Par[], double x);
```

Wrapper function for the chi-square distribution, similar to `wdist_Normal`. Returns $P[X \leq x]$, where X has the chi-square distribution with k degrees of freedom. The value of k must be in `Par[0]`.

```
double wdist_Unif (double Par[], double x);
```

Wrapper function for the uniform distribution. Returns x. `Par` is unused.

fbar

This module is similar to `fdist`, except that it provides procedures to compute or approximate the complementary distribution function of X, which we define as $\bar{F}(x) = P[X \geq x]$, instead of $F(x) = P[X \leq x]$. Note that with our definition of $\bar{F}$, one has $\bar{F}(x) = 1 - F(x)$ for continuous distributions and $\bar{F}(x) = 1 - F(x-1)$ for discrete distributions over the integers. This is non-standard but we find it convenient.

For more details about the specific distributions, see the module `fdist`. When $F(x)$ is very close to 1, these procedures generally provide much more precise values of $\bar{F}(x)$ than using $1 - F(x)$ where $F(x)$ is computed by a procedure from `fdist`.

```
#include <testu01/gdef.h>
#include <testu01/fmass.h>
```

Continuous distributions

`double fbar_Unif (double x);`

Returns $1 - x$ for $x \in [0, 1]$, 1 for $x < 0$, and 0 for $x > 1$. This is the complementary uniform distribution function over $[0, 1]$.

`double fbar_Expon (double x);`

Returns the complementary exponential distribution: $\bar{F}(x) = e^{-x}$ for $x > 0$, and $= 1$ for $x \leq 0$.

`double fbar_Weibull (double alpha, double x);`

Returns the complementary standard Weibull distribution function with shape parameter α [19], defined by $\bar{F}(x) = e^{-x^\alpha}$ for $x > 0$ and 1 for $x \leq 0$. Restriction: $\alpha > 0$.

`double fbar_Logistic (double x);`

Returns $\bar{F}(x) = 1/(1+e^x)$, the complementary standard logistic distribution function evaluated at x [20].

`double fbar_Pareto (double c, double x);`

Returns $\bar{F}(x) = 1/x^c$ for $x \geq 1$ and 1 for $x \leq 1$, which is the complementary standard Pareto distribution function [19]. Restriction: $c > 0$.

`double fbar_Normal1 (double x);`

Returns an approximation of $1 - \Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses a Chebyshev series giving 16 decimal digits of precision [36].

`double fbar_Normal2 (double x);`

Returns an approximation of $1-\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses Marsaglia's et al [31] fast method with tables lookup. Returns 15 decimal digits of precision. This function is approximately 1.3 times faster than `fbar_Normal1`.

```
#ifdef HAVE_ERF
   double fbar_Normal3 (double x);
#endif
```

Returns an approximation of $1-\Phi(x)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses the `erfc` function from the standard Unix C library. The macro `HAVE_ERF` from `mylib/gdef` must be defined. This function is twice as fast as `fbar_Normal2`.

`double fbar_BiNormal1 (double x, double y, double rho, int ndig);`

Returns the value u of the upper standard bivariate normal distribution, given by

$$u = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_x^\infty \int_y^\infty e^{-T} dy\, dx \tag{38}$$

$$T = \frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)},$$

where $\rho =$ `rho` is the correlation between x and y, and `ndig` is the number of decimal digits of accuracy. It calls the function `fdist_BiNormal1`. The absolute error is expected to be smaller than 10^{-d}, where $d =$ `ndig`. Restriction: `ndig` ≤ 15.

`double fbar_BiNormal2 (double x, double y, double rho);`

Returns the value of the upper standard bivariate normal distribution as defined in (38) above. It calls the function `fdist_BiNormal2` (see the description in module `fdist`). The function gives an absolute error less than $5 \cdot 10^{-16}$.

`double fbar_ChiSquare1 (long N, double x);`

Returns $\bar{F}(x)$, the complementary chi-square distribution function with N degrees of freedom. Uses the approximation given in [21, p.116] for $N \leq 1000$, and the normal approximation for $N > 1000$. Gives no more than 4 decimals of precision for $N > 1000$.

`double fbar_ChiSquare2 (long N, int d, double x);`

Returns $\bar{F}(x)$, the complementary chi-square distribution function with N degrees of freedom, by calling `fbar_Gamma (N/2, d, x/2)`. The function will do its best to return d decimals digits of precision (but there is no guarantee). Restrictions: $N > 0$ and $0 < d \leq 15$.

`double fbar_Gamma (double a, int d, double x);`

Returns an approximation [4] of the complementary *gamma* distribution function with parameter a. The function tries to return d decimals digits of precision. For a not too large (e.g., $a \leq 1000$), d gives a good idea of the precision attained. Restrictions: $a > 0$ and $0 < d \leq 15$.

`double fbar_KS1 (long n, double x);`

Returns the complementary Kolmogorov-Smirnov distribution $\bar{F}(x) = P[D_n \geq x]$ in a form that is more precise in the upper tail, using the program described in [**?**]. It returns at least 10 decimal digits of precision everywhere for all $n \leq 400$, at least 6 decimal digits of precision for $400 < n \leq 200000$, and a few correct digits (1 to 5) for $n > 200000$. Restrictions: $n \geq 1$ and $0 \leq x \leq 1$.

`double fbar_KSPlus (long n, double x);`

Returns the complementary Kolmogorov-Smirnov+ distribution $\bar{F}(x) = P[D_n^+ \geq x]$ in a form that is more precise in the upper tail. It should return at least 8 decimal digits of precision everywhere. Restrictions: $n > 0$ and $0 \leq x \leq 1$.

```
double fbar_LogNormal (double mu, double sigma, double x);

double fbar_JohnsonSB (double alpha, double beta, double a, double b,
                       double x);

double fbar_JohnsonSU (double alpha, double beta, double x);

double fbar_CramerMises (long n, double x);

double fbar_WatsonU (long n, double x);

double fbar_WatsonG (long n, double x);

double fbar_AndersonDarling (long n, double x);
```

Return the complementary distribution function $P[X \geq x]$. See the description of the respective functions in `fdist`.

Discrete distributions

`double fbar_Geometric (double p, long s);`

Returns the complementary distribution function of a geometric random variable X with parameter p, $\bar{F}(s) = P[X \geq s] = (1-p)^s$ for $s \geq 0$. Restriction: $0 \leq p \leq 1$.

`double fbar_Poisson1 (double lambda, long s);`

Returns the complementary distribution function $P[X \geq s]$ for a Poisson random variable X with parameter λ. Computes and adds the non-negligible terms in the tail. Restriction: $\lambda > 0$.

`double fbar_Poisson2 (fmass_INFO W, long s);`

Returns the complementary Poisson distribution function, using the structure `W` which must have been created previously by calling `fmass_CreatePoisson` with the desired λ.

`double fbar_Binomial2 (fmass_INFO W, long s);`

Returns the complementary distribution function $P[X \geq s]$ for a binomial random variable X, using the structure `W` which must have been created previously by calling `fmass_CreateBinomial` with the desired values of n and p.

`double fbar_NegaBin2 (fmass_INFO W, long s);`

Returns the complementary distribution function $P[X \geq s]$ for a negative binomial random variable X, using the structure `W` which must have been created previously by calling `fmass_CreateNegaBin` with the desired values of n and p.

`double fbar_Scan (long N, double d, long m);`

Return $P[S_N(d) \geq m]$, where $S_N(d)$ is the scan statistic (see [17] and `gofs_Scan`), defined as

$$S_N(d) = \sup_{0 \leq y \leq 1-d} \eta[y,\, y+d], \tag{39}$$

where d is a constant in $(0, 1)$, $\eta[y,\, y+d]$ is the number of observations falling inside the interval $[y, y+d]$, from a sample of N i.i.d. $U(0, 1)$ random variables. One has (see [1]),

$$P[S_N(d) \geq m] \;\approx\; \left(\frac{m}{d} - N - 1\right) b(m) + 2\sum_{i=m}^{N} b(i) \tag{40}$$

$$\approx\; 2(1 - \Phi(\theta\kappa)) + \theta\kappa \frac{\exp[-\theta^2\kappa^2/2]}{d\sqrt{2\pi}} \tag{41}$$

where Φ is the standard normal distribution function,

$$\begin{aligned} b(i) &= \binom{N}{i} d^i (1-d)^{N-i}, \\ \theta &= \sqrt{\frac{d}{1-d}}, \\ \kappa &= \frac{m}{d\sqrt{N}} - \sqrt{N}. \end{aligned}$$

For $d \leq 1/2$, (40) is exact for $m > N/2$, but only an approximation otherwise. The approximation (41) is good when Nd^2 is large or when $d > 0.3$ and $N > 50$. In other cases, this implementation sometimes use the approximation proposed by Glaz [17]. For more information, see [1, 17, 41]. The approximation returned by this function is generally good when it is close to 0, but is not very reliable when it exceeds, say, 0.4. Restrictions: $N \geq 2$ and $d \leq 1/2$.

finv

Here one finds procedures to compute or approximate the inverse of certain distribution functions. Each procedure computes $F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}$, where $0 \leq u \leq 1$ and F is the distribution function of a specific type of random variable. These procedures can be used, among other things, to generate the corresponding random variables by inversion, by passing a $U(0,1)$ random variate as the value of u.

Several distributions are only implemented in standardized form here, i.e., with the location parameter set to 0 and the scale parameter set to 1. To obtain the inverse for the distribution shifted by x_0 and rescaled by a factor c, it suffices to multiply the returned value by c and add x_0.

```
#include <testu01/gdef.h>     /* From the library mylib */
#include <testu01/fmass.h>
#include <testu01/fdist.h>
#include <testu01/wdist.h>
```

Continuous distributions

`double finv_Expon (double u);`

Returns the inverse of the standard exponential distribution,

$$F^{-1}(u) = -\ln(1-u), \qquad 0 \leq u \leq 1.$$

`double finv_Weibull (double alpha, double u);`

Returns the inverse of the standard Weibull distribution,

$$F^{-1}(u) = (-\ln(1-u))^{1/\alpha}, \qquad 0 \leq u \leq 1.$$

Restriction: $\alpha > 0$.

`double finv_ExtremeValue (double u);`

Returns the inverse of the standard extreme value distribution,

$$F^{-1}(u) = -\ln(-\ln(u)), \qquad 0 \leq u \leq 1.$$

`double finv_Logistic (double u);`

Returns the inverse of the standard logistic distribution,

$$F^{-1}(u) = \ln\left(\frac{u}{1-u}\right), \qquad 0 \leq u \leq 1.$$

`double finv_Pareto (double c, double u);`

Returns the inverse of the standard Pareto distribution,

$$F^{-1}(u) = \left(\frac{1}{1-u}\right)^{1/c}, \qquad 0 \le u \le 1.$$

Restriction: $c > 0$.

`double finv_Normal1 (double u);`

Returns an approximation of $\Phi^{-1}(u)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses rational Chebyshev approximations giving at least 15 decimal digits of precision over most of the range [5]. Far in the lower tail ($u < 10^{-122}$), the precision decreases slowly until for $u < 10^{-308}$, the function gives only 11 decimal digits of precision.

`double finv_Normal2 (double u);`

Returns an approximation of $\Phi^{-1}(u)$, where Φ is the standard normal distribution function, with mean 0 and variance 1. Uses Marsaglia's et al [31] method with tables lookup. The method works provided that the processor respects the IEEE-754 floating-point standard. Returns 6 decimal digits of precision. This function is twice as fast as `finv_Normal1`.

`double finv_LogNormal (double mu, double sigma, double u);`

Returns the inverse of the lognormal distribution,

$$F^{-1}(u) = e^{\mu + \sigma\Phi^{-1}(u)}, \qquad 0 \le u \le 1.$$

Restriction: $\sigma > 0$.

```
double finv_JohnsonSB (double alpha, double beta, double a, double b,
                       double u);
```

Returns the inverse of the Johnson JSB distribution,

$$F^{-1}(u) = \frac{a + b\,v}{1 + v},$$

where

$$v = \exp\left(\frac{\Phi^{-1}(u) - \alpha}{\beta}\right), \qquad 0 \le u \le 1.$$

and Φ^{-1} is the inverse of the standard normal distribution. Restrictions: $\beta > 0$ and $a < x < b$.

`double finv_JohnsonSU (double alpha, double beta, double u);`

Returns the inverse of the Johnson JSU distribution,

$$F^{-1}(u) = \frac{v - 1/v}{2}$$

where

$$v = \exp\left(\frac{\Phi^{-1}(u) - \alpha}{\beta}\right), \qquad 0 \le u \le 1.$$

and Φ^{-1} is the inverse of the standard normal distribution. Restriction: $\beta > 0$.

`double finv_ChiSquare1 (long k, double u);`

Returns a quick and dirty approximation of $F^{-1}(u)$, where F is the chi-square distribution with k degrees of freedom. Uses the approximation given in Figure L.24 of [6].

`double finv_ChiSquare2 (long k, double u);`

Returns an approximation of $F^{-1}(u)$, where F is the chi-square distribution with k degrees of freedom. Uses the approximation given in [3] and in Figure L.23 of [6]. This function is up to 20 times slower than `finv_ChiSquare1`.

`double finv_Student (long n, double u);`

Returns an approximation of $F^{-1}(u)$, where F is the *Student-t* distribution function with n degrees of freedom. Uses an approximation giving at least 5 decimal digits of precision when $n \ge 8$ or $n \le 2$, and 3 decimal digits of precision when $3 \le n \le 7$ (see [18] and Figure L.28 of [6]).

`double finv_BetaSymmetric (double p, double u);`

Returns a special approximation of $F^{-1}(u)$, where $F(x)$ is the symmetric *beta* distribution with shape parameter $p = q$ as defined in (21). Uses four different hypergeometric series (for the four cases x close to 0 and $p \le 1$, x close to 0 and $p > 1$, x close to 1/2 and $p \le 1$, and x close to 1/2 and $p > 1$) to compute the distribution $u = F(x)$, which are then solved by Newton's method for the solution of equations. For $p > 100000$, uses a normal approximation given in [32]. Restrictions: $p > 0$ and $0 \le u \le 1$.

```
double finv_GenericC (wdist_CFUNC F, double par[], double u, int d,
                      int detail);
```

Uses binary search to find the inverse of a generic continuous distribution function F, evaluated at u. The parameters of F (if any) are passed in the array `par`. The returned value has d decimal digits of precision. If `detail > 0`, the procedure will print detailed information about the inversion process. Restrictions: $0 \le u \le 1$ and $d > 0$.

`long finv_Geometric (double p, double u);`

Returns the inverse of the geometric distribution,

$$F^{-1}(u) = \left\lfloor \frac{\ln(1-u)}{\ln(1-p)} \right\rfloor, \qquad 0 \le u \le 1.$$

Restriction: $0 \le p \le 1$.

gofs

This module provides tools for computing goodness-of-fit test statistics for testing the hypothesis $\mathcal{H}_0$ that a sample of N observations $V_1, \ldots, V_N$ comes from a given univariate probability distribution F. These test statistics generally measure, in different ways, the distance between F and the *empirical distribution function* (EDF) $\hat{F}_N$ of $V_1, \ldots, V_N$. They are also called EDF test statistics. The observations V_i are usually transformed into $U_i = F(V_i)$, which always satisfy $0 \leq U_i \leq 1$, and which follow the $U(0,1)$ distribution under $\mathcal{H}_0$. These observations are also usually sorted. Here, $U_{(1)}, \ldots, U_{(N)}$ stand for N observations $U_1, \ldots, U_N$ sorted by increasing order, where $0 \leq U_i \leq 1$.

Procedures for applying certain types of transformations to the observations V_i or U_i are also provided. This includes the transformation $U_i = F(V_i)$, as well as the power ratio and iterated spacing transformations [40].

```
#include <testu01/bitset.h>       /* From the library mylib */
#include <testu01/fmass.h>
#include <testu01/fdist.h>
#include <testu01/wdist.h>
```

Environment variables

```
extern double gofs_MinExpected;
```

Used for the chi-square tests. When a chi-square test statistic is computed, the expected number of observations in each class should be large enough if we want the chi-square test statistic to follow approximately the chi-square distribution. Larger expected numbers are usually required when these numbers differ between classes [35]. The function `gofs_MergeClasses` can be used to regroup classes in order to make sure that the expected number in each class is at least `gofs_MinExpected`. The default value of this variable is 10.0.

Transforming the observations

```
void gofs_ContUnifTransform (double V[], long N, wdist_CFUNC F,
                             double par[], double U[]);
```

Applies the transformation $U_i = F(V_i)$ to the values in `V[1..N]`, where F is a *continuous* distribution function given by `F` and with parameters in `par`, and puts the result in `U[1..N]`. If `V` contains random variables from the distribution function `F`, then `U` will contain uniform random variables over $(0,1)$.

```
void gofs_DiscUnifTransform (double V[], long N, wdist_DFUNC F,
                             fmass_INFO W, double U[]);
```

Applies the transformation $U_i = F(V_i)$ to the values in `V[1..N]`, where F is a *discrete* distribution function specified by `F` and the previously-created structure `W`, and puts the result in

`U[1..N]`. Note: If `V[1..N]` are the values of random variables with distribution function `F`, then `U[1..N]` will contain the values of *discrete* random variables distributed over the set of values taken by `F`, not uniform random variables over $(0, 1)$.

```
void gofs_DiffD (double U[], double D[], long N1, long N2,
                 double a, double b);
```

Assumes that the real-valued observations `U[N1..N2]` are already sorted in increasing order and computes the differences between the successive observations. The difference `U[i+1] - U[i]` is put in `D[i]` for `N1 <= i < N2`, whereas `U[N1] - a` is put into `D[N1-1]` and `b - U[N2]` is put into `D[N2]`. The sizes of the arrays `U` and `D` must be at least `N2+1`.

```
void gofs_DiffL (long U[], long D[], long N1, long N2, long a, long b);

#ifdef USE_LONGLONG
void gofs_DiffLL (longlong U[], longlong D[], long N1, long N2,
                  longlong a, longlong b);
void gofs_DiffULL (ulonglong U[], ulonglong D[], long N1, long N2,
                   ulonglong a, ulonglong b);
#endif
```

Same as `gofs_DiffD`, but for integer-valued observations.

```
void gofs_IterateSpacings (double V[], double S[], long N);
```

Applies one iteration of the *iterated spacings* transformation [23, 40]. Assumes that `S[0...N]` contains the *spacings* between N real numbers $U_1, \ldots, U_N$ in the interval $[0, 1]$. These spacings are defined by

$$S_i = U_{(i+1)} - U_{(i)}, \qquad 0 \le i \le N,$$

where $U_{(0)} = 0$, $U_{(N+1)} = 1$, and $U_{(1)}, \ldots, U_{(N)}$, are the U_i sorted in increasing order. These spacings may have been obtained by calling `gofs_DiffD`. This procedure transforms the spacings into new spacings, by a variant of the method described in section 11 of [28] and also by Stephens [40]: it sorts $S_0, \ldots, S_N$ to obtain $S_{(0)} \le S_{(1)} \le S_{(2)} \le \cdots \le S_{(N)}$, computes the weighted differences

$$\begin{aligned} S_0 &= (N+1)S_{(0)}, \\ S_1 &= N(S_{(1)} - S_{(0)}), \\ S_2 &= (N-1)(S_{(2)} - S_{(1)}), \\ &\vdots \\ S_N &= S_{(N)} - S_{(N-1)}, \end{aligned}$$

and computes $V_i = S_0 + S_1 + \cdots + S_{i-1}$ for $1 \le i \le N$. It then returns $S_0, \ldots, S_N$ in `S[0..N]` and $V_1, \ldots, V_N$ in `V[1..N]`.

Under the assumption that the U_i are i.i.d. $U(0, 1)$, the new S_i can be considered as a new set of spacings having the same distribution as the original spacings, and the V_i are a new sample of i.i.d. $U(0, 1)$ random variables, sorted by increasing order.

This transformation is useful to detect *clustering* in a data set: A pair of observations that are close to each other is transformed into an observation close to zero. A data set with unusually clustered observations is thus transformed to a data set with an accumulation of observations near zero, which is easily detected by the Anderson-Darling GOF test.

```
void gofs_PowerRatios (double U[], long N);
```

Applies the *power ratios* transformation W described in section 8.4 of Stephens [40]. Assume that `U[1...N]` contains N real numbers $U_{(1)}, \ldots, U_{(N)}$ from the interval $[0, 1]$, already sorted in increasing order, and computes the transformations:

$$U'_i = (U_{(i)}/U_{(i+1)})^i, \qquad i = 1, \ldots, N,$$

with $U_{(N+1)} = 1$. These U'_i are sorted in increasing order and put back in `U[1...N]`. If the $U_{(i)}$ are i.i.d. $U(0, 1)$ sorted by increasing order, then the U'_i are also i.i.d. $U(0, 1)$.

This transformation is useful to detect clustering, as explained in `gofs_IterateSpacings`, except that here a pair of observations close to each other is transformed into an observation close to 1. An accumulation of observations near 1 is also easily detected by the Anderson-Darling GOF test.

```
void gofs_MergeClasses (double NbExp[], long Loc[],
                        long *smin, long *smax, long *NbClasses);
```

This function is convenient for regrouping classes before applying a chi-square test, in the case where the expected number of observations in some of the classes may be too small. It merges classes of observations so that the expected number of observations in each class is at least `gofs_MinExpected`. Initially, the expected numbers in each class are in `NbExp[*smin...*smax]`. When the function returns, if `Loc`$[s] = j$, this means that class s has been merged with class j. In this case, all observations that previously belonged to class s are redirected to class j, and `NbExp`$[s]$ has been added to `NbExp`$[j]$ and then set to zero. `NbClasses` gives the final number of classes, `smin` contains the new index of the lowest class, and `smax` the new index of the highest class.

```
void gofs_WriteClasses (double NbExp[], long Loc[],
                        long smin, long smax, long NbClasses);
```

Prints the classes before or after their regrouping by `gofs_MergeClasses`. The parameters are the same as for the latter function. If `NbClasses > 0`, assumes that `gofs_MergeClasses` has already been called to regroup classes and prints the classes after the regrouping. If `NbClasses <= 0`, prints only the classes before any regrouping.

Computing EDF test statistics

```
double gofs_Chi2 (double NbExp[], long Count[], long smin, long smax);
```

Computes and returns the chi-square statistic for the observations o_i in `Count[smin...smax]`, for which the corresponding expected values e_i are in `NbExp[smin...smax]`. Assuming that i

goes from 1 to k, where $k =$ `smax-smin+1` is the number of classes, the chi-square statistic is defined as

$$X^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}. \tag{42}$$

Under the hypothesis that the e_i are the correct expectations and if these e_i are large enough, X^2 follows approximately the chi-square distribution with $k-1$ degrees of freedom. If some of the e_i are too small, one can use `gofs_MergeClasses` to regroup classes.

`double gofs_Chi2Equal (double NbExp, long Count[], long smin, long smax);`

Similar to `gofs_Chi2`, except that the expected number of observations per class is assumed to be the same for all classes, and equal to `NbExp`.

`long gofs_Scan (double U[], long N, double d);`

Computes and returns the scan statistic $S_N(d)$, defined in (39). The N observations in the array `U[1..N]` must be real numbers in the interval $[0, 1]$, sorted in increasing order. (See `fbar_Scan` for the distribution function of $S_N(d)$).

`double gofs_CramerMises (double U[], long N);`

Computes and returns the Cramér-von Mises statistic W_N^2 (see [13, 38, 39]), defined by

$$W_N^2 = \frac{1}{12N} + \sum_{j=1}^{N} \left(U_{(j)} - \frac{(j-0.5)}{N} \right)^2, \tag{43}$$

assuming that `U[1...N]` contains $U_{(1)}, \ldots, U_{(N)}$ sorted in increasing order.

`double gofs_WatsonG (double U[], long N);`

Computes and returns the Watson statistic G_N (see [42, 10]), defined by

$$\begin{aligned} G_N &= \sqrt{N} \max_{1 \le j \le N} \left\{ j/N - U_{(j)} + \overline{U}_N - 1/2 \right\} \\ &= \sqrt{N} \left(D_N^+ + \overline{U}_N - 1/2 \right), \end{aligned} \tag{44}$$

where $\overline{U}_N$ is the average of the observations $U_{(j)}$, assuming that `U[1...N]` contains the sorted $U_{(1)}, \ldots, U_{(N)}$.

`double gofs_WatsonU (double U[], long N);`

Computes and returns the Watson statistic U_N^2 (see [13, 38, 39]), defined by

$$W_N^2 = \frac{1}{12N} + \sum_{j=1}^{N} \left\{ U_{(j)} - \frac{(j-0.5)}{N} \right\}^2, \tag{45}$$

$$U_N^2 = W_N^2 - N \left(\overline{U}_N - 1/2 \right)^2. \tag{46}$$

where $\overline{U}_N$ is the average of the observations $U_{(j)}$, assuming that `U[1...N]` contains the sorted $U_{(1)}, \ldots, U_{(N)}$.

`double gofs_AndersonDarling (double U[], long N);`

Computes and returns the Anderson-Darling statistic A_N^2 (see [26, 39, 2]), defined by

$$A_N^2 \;=\; -N - \frac{1}{N}\sum_{j=1}^{N}\left\{(2j-1)\ln(U_{(j)}) + (2N+1-2j)\ln(1-U_{(j)})\right\},$$

assuming that `U[1...N]` contains $U_{(1)}, \ldots, U_{(N)}$.

`void gofs_KS (double U[], long N, double *DP, double *DM, double *D);`

Computes the Kolmogorov-Smirnov (KS) test statistics D_N^+, D_N^-, and D_N, defined by

$$D_N^+ \;=\; \max_{1\le j\le N}\left(j/N - U_{(j)}\right), \tag{47}$$

$$D_N^- \;=\; \max_{1\le j\le N}\left(U_{(j)} - (j-1)/N\right), \tag{48}$$

$$D_N \;=\; \max\,(D_N^+, D_N^-). \tag{49}$$

and return their values in `DP`, `DM`, and `D`, respectively. These statistics compare the empirical distribution of $U_{(1)}, \ldots, U_{(N)}$, which are assumed to be in `U[1...N]`, with the uniform distribution.

`void gofs_KSJumpOne (double U[], long N, double a, double *DP, double *DM);`

Compute the KS statistics $D_N^+(a)$ and $D_N^-(a)$ defined in the description of the function `fdist_KSPlusJumpOne`, assuming that F is the uniform distribution over $[0, 1]$ and that $U_{(1)}, \ldots, U_{(N)}$ are in `U[1...N]`. Returns the values in `DP` and `DM`.

gofw

This module contains functions used to print results of GOF test statistics (see module `gofs`), or to apply a series of tests simultaneously and print the results. Strictly speaking, applying several tests simultaneously makes the p-values "invalid" in the sense that the probability of having *at least one* p-value less than 0.01, say, is larger than 0.01. One must therefore be careful with the interpretation of these p-values (one could use, e.g., the Bonferroni inequality [25]). Applying simultaneous tests is convenient in some situations, such as in screening experiments for detecting statistical deficiencies in random number generators. In that context, rejection of the null hypothesis typically occurs with extremely small p-values (e.g., less than 10^{-15}), and the interpretation is quite obvious in this case.

The module also provides tools to plot an empirical or theoretical distribution function, by creating a data file that contains a graphic plot in a format compatible with the software specified by the environment variable `gofw_GraphSoft`.

```
#include <testu01/gdef.h>           /* From the library mylib */
#include <testu01/bitset.h>         /* From the library mylib */
#include <testu01/fdist.h>
#include <testu01/wdist.h>
#include <stdio.h>
```

Plotting distribution functions

```
typedef enum {
   gofw_Gnuplot,
   gofw_Mathematica
   } gofw_GraphType;
```

Data file formats used for plotting functions or creating graphics.

```
extern gofw_GraphType gofw_GraphSoft;
```

Environment variable that selects the type of software to be used for plotting the graphs of functions. The data files produced by `gofw_GraphFunc` and `gofw_GraphDistUnif` will be in a format suitable for this selected software. The default value is `gofw_Gnuplot`. To display a graphic in file `f` using `gnuplot`, for example, one can use the command "`plot f with steps, x with lines`" in `gnuplot`.

```
void gofw_GraphFunc (FILE *f, wdist_CFUNC F, double par[], double a,
                     double b, int m, int mono, char Desc[]);
```

Prints data to plot the graph of function F over the interval $[a, b]$, in file `f`. It is assumed that the parameters of F are in `par`, so that `F(par, x)` returns the value of F at x, and that F is either non-decreasing or non-increasing. If `mono` $= 1$, the procedure will verify that F is non-decreasing; if `mono` $= -1$, it will verify that F is non-increasing. (This is useful to verify if

F is effectively a sensible approximation to a distribution function or its complementary in the given interval.) The string `Desc` should give a short caption for the graphic plot. The procedure computes the $m+1$ points $(x_i, F(x_i))$, where $x_i = a+i(b-a)/m$ for $i = 0, 1, \ldots, m$, and writes these points to file `f` in a format suitable for the software specified by `gofw_GraphSoft`. If `f = NULL`, the results are sent to the standard output.

```
void gofw_GraphDistUnif (FILE *f, double U[], long N, char Desc[]);
```

Prints data in file `f` to plot the empirical distribution of $U_{(1)}, \ldots, U_{(N)}$, which are assumed to be in `U[1...N]`, and to compare it with the uniform distribution. The two endpoints $(0, 0)$ and $(1, 1)$ are always printed. The string `Desc` should give a short caption for the graphic plot. The data is printed in a format suitable for the software specified by `gofw_GraphSoft`. If `f = NULL`, the results are sent to the standard output.

Computing and printing p-values for EDF test statistics

```
extern double gofw_Epsilonp;
extern double gofw_Epsilonp1;
```

Environment variables used in `gofw_Writep0` to determine which p-values are too close to 0 or 1 to be printed explicitly. If `gofw_Epsilonp` $= \epsilon$ and `gofw_Epsilonp1` $= \epsilon_1$, then any p-value (sometimes also called significance level) less than ϵ or larger than $1-\epsilon_1$ is *not* written explicitly; the program simply writes "`eps`" (p-values close to 0) or "`1 - eps1`" (p-values close to 1). The default values are `gofw_Epsilonp` $= 10^{-300}$ and `gofw_Epsilonp1` $= 10^{-15}$. The default value of `gofw_Epsilonp` is slightly bigger than the minimum normalized positive floating-point number `DBL_MIN` $= 2.2 * 10^{-308}$ given in the IEEE floating-point standard, while `gofw_Epsilonp1` is slightly bigger than `DBL_EPSILON` $= 2.2 * 10^{-16}$, the "machine ϵ" for type `double`.

```
extern double gofw_Suspectp;
```

Environment variable used in `gofw_Writep1` to determine which p-values should be marked as suspect when printing test results. If `gofw_Suspectp` $= \alpha$, then any p-value less than α or larger than $1-\alpha$ is considered suspect and is "singled out" by `gofw_Writep1`. The default value is 0.001.

```
double gofw_pDisc (double pL, double pR);
```

Computes a variant of the p-value p whenever a test statistic has a *discrete* probability distribution. This p-value is defined as follows:

$$
\begin{aligned}
p_L &= P[Y \le y] \\
p_R &= P[Y \ge y] \\
p &= \begin{cases} p_R, & \text{if } p_R < p_L \\ 1 - p_L, & \text{if } p_R \ge p_L \text{ and } p_L < 0.5 \\ 0.5 & \text{otherwise.} \end{cases}
\end{aligned}
$$

The function takes p_L and p_R as input and returns p.

`void gofw_Writep0 (double p);`

Prints the p-value p of a test, in the format "$1 - p$" if p is close to 1, and p otherwise.

`void gofw_Writep1 (double p);`

Prints the string "`"p-value of test :  "`", then calls `gofw_Writep0` to print p, and adds the marker "`****`" if p is considered suspect (uses the environment variable `gofw_Suspectp` for this).

`void gofw_Writep2 (double x, double p);`

Prints `x` on the current output line, then goes to the next line and calls `gofw_Writep1 (p)`.

`void gofw_WriteKS0 (long N, double DP, double DM, double D);`

Computes the p-values of the three Kolmogorov-Smirnov statistics D_N^+, D_N^-, and D_N, whose values are in `DP, DM, D`, respectively, assuming a sample of size N. Then prints these statistics and their p-values using `gofw_Writep2` for each one.

`void gofw_WriteKS1 (double V[], long N, wdist_CFUNC F, double par[]);`

Computes the KS test statistics to compare the empirical distribution of the observations in `V[1..N]` with the theoretical distribution `F`, with parameters in `par`, then calls `gofw_KS0` to compute and print the p-values. These tests are valid only if `F` is continuous.

`void gofw_WriteKSJumpOne0 (long N, double a, double DP);`

Similar to `gofw_KS0`, but for the KS statistic $D_N^+(a)$ defined in (28). Writes a header, computes the p-value and calls `gofw_Writep2`.

```
void gofw_WriteKSJumpOne1 (double V[], long N,
                           wdist_CFUNC F, double par[], double a);
```

Similar to `gofw_WriteKS1`, but for $D_N^+(a)$ defined in (28). Calls `gofw_WriteKSJumpOne0`.

Applying several tests at once and printing results

Higher-level tools for applying several EDF goodness-of-fit tests simultaneously are offered here. The test types available are listed in `gofw_TestType`. The environment variable `gofw_ActiveTests` specifies which tests in this list are to be performed when asking for several simultaneous tests via the functions `gofw_ActiveTests0`, `gofw_WriteActiveTests0`, etc.

```
typedef enum {
   gofw_KSP,                        /* Kolmogorov-Smirnov+        */
   gofw_KSM,                        /* Kolmogorov-Smirnov-        */
   gofw_KS,                         /* Kolmogorov-Smirnov         */
   gofw_AD,                         /* Anderson-Darling           */
   gofw_CM,                         /* Cramer-vonMises            */
   gofw_WG,                         /* Watson G                   */
   gofw_WU,                         /* Watson U                   */
   gofw_Mean,                       /* Mean                       */
   gofw_Var,                        /* Variance                   */
   gofw_Cor,                        /* Correlation                */
   gofw_Sum,                        /* Sum                        */
   gofw_NTestTypes                  /* Total number of test types */
   } gofw_TestType;
```

Types of EDF tests supported by the present modules. Here, `gofw_Sum`, `gofw_Mean`, `gofw_Var` and `gofw_Cor` usually represent tests based on the sum, the mean, the variance of the observations and on the correlation between pairs of successive observations.

```
typedef double gofw_TestArray [gofw_NTestTypes];
```

Array of values, one for each type of EDF test statistic. Can be used to store the values of these statistics or their p-values, for example.

```
extern char *gofw_TestNames [gofw_NTestTypes];
```

Name of each `gofw_TestType` test. Could be used for printing the test results, for example.

```
extern bitset_BitSet gofw_ActiveTests;
```

The set of EDF tests that are to be performed when calling the procedures `gofw_ActiveTests0`, `gofw_WriteActiveTests0`, etc. By default, this set contains `gofw_KSP`, `gofw_KSM`, and `gofw_AD`. Note: `gofw_Sum`, `gofw_Mean`, `gofw_Var` and `gofw_Cor` are *always excluded* from this set of active tests.

```
void gofw_InitTestArray (gofw_TestArray A, double x);
```

Sets all elements of array A to x.

```
void gofw_Tests0 (double U[], long N, gofw_TestArray sVal);
```

Computes all EDF test statistics in `gofw_TestType` (except `gofw_Mean`, `gofw_Var` `gofw_Cor` and `gofw_Sum`) to compare the empirical distribution of $U_{(1)}, \ldots, U_{(N)}$ with the uniform distribution, assuming that these sorted observations are in `U[1...N]`. If $N > 1$, returns in `sVal[0..7]` the values of the KS statistics D_N^+, D_N^- and D_N, of the Cramér-von Mises statistic W_N^2, Watson's G_N and U_N^2, Anderson-Darling's A_N^2, and the average of the U_i's, respectively. If $N = 1$, only puts `U[1]` in `sVal[gofw_Mean]` and `1 - U[1]` in `sVal[gofw_KSP]`. Calling this function is more efficient than computing these statistics separately by calling the corresponding procedures in `gofs`.

```
void gofw_Tests1 (double V[], long N, wdist_CFUNC F, double par[],
                  gofw_TestArray sVal);
```

Similar to `gofw_Test0`, except that the observations are in `V[1..N]`, not necessarily sorted, and that their empirical distribution is compared with the continuous distribution `F`, whose parameters (if any) are in `par`. If $N = 1$, only puts `V[1]` in `sVal[gofw_Mean]` and `1 - F(par, V[1])` in `sVal[gofw_KSP]`.

```
void gofw_ActiveTests0 (double U[], long N,
                        gofw_TestArray sVal, gofw_TestArray pVal);
```

Computes the EDF test statistics by calling `gofw_Tests0`, then computes the p-values of those that currently belong to `gofw_ActiveTests`, and return these quantities in `sVal` and `pVal`, respectively. Assumes that $U_{(1)}, \ldots, U_{(N)}$ are in `U[1...N]` and that we want to compare their empirical distribution with the uniform distribution. If $N = 1$, only puts `U[1]` in `sVal[gofw_Mean]`, and `1 - U[1]` in `sVal[gofw_KSP], pVal[gofw_KSP]`, and `pVal[gofw_Mean]`.

```
void gofw_ActiveTests1 (double V[], long N, wdist_CFUNC F, double par[],
                        gofw_TestArray sVal, gofw_TestArray pVal);
```

Similar to `gofw_ActiveTests0`, except that the observations are in `V[1..N]`, not necessarily sorted, and that we want to compare their empirical distribution with the distribution `F`, whose parameters (if any) are in `par`. The EDF tests are valid only if `F` is continuous. If $N = 1$, only puts `V[1]` in `sVal[gofw_Mean]`, and `1 - F(par, V[1])` in `sVal[gofw_KSP], pVal[gofw_KSP]`, and `pVal[gofw_Mean]`.

```
void gofw_ActiveTests2 (double V[], double U[], long N, wdist_CFUNC F,
                        double par[], gofw_TestArray sVal,
                        gofw_TestArray pVal);
```

Similar to `gofw_ActiveTests1`, but first sorts the `V` and then returns the U computed from

$$U[j] = F(\text{par}, V[j]), \qquad j = 1, \ldots, N$$

and sorted.

```
void gofw_WriteActiveTests0 (long N, gofw_TestArray sVal,
                                     gofw_TestArray pVal);
```

Writes the p-values of the *active* EDF test statistics, which are in `gofw_ActiveTests`. It is assumed that the values of these statistics and their p-values are *already computed*, in `sVal` and `pVal`, and that the sample size is N. These statistics and p-values are printed using `gofw_Writep2` for each one. If $N = 1$, prints only `pVal[gofw_KSP]` using `gofw_Writep1`.

```
void gofw_WriteActiveTests1 (double V[], long N,
                             wdist_CFUNC F, double par[]);
```

This is equivalent to calling `gofw_ActiveTests1 (V, N, F, par, sVal, pVal)` followed by `gofw_WriteActiveTests0 (N, sVal, pVal)`.

```
void gofw_WriteActiveTests2 (long N, gofw_TestArray sVal,
                             gofw_TestArray pVal, char Desc[]);
```

If $N = 1$, prints the string `Desc` followed by the elements `gofw_Mean` of `sVal` and `pVal`. Otherwise calls `gofw_WriteActiveTests0 (N, sVal, pVal)`.

```
void gofw_IterSpacingsTests0 (double U[], long N, int k,
                              lebool printval, lebool graph, FILE *f);
```

Repeats the following k times: Applies the `gofs_IterateSpacings` transformation to the $U_{(1)}, \ldots, U_{(N)}$, assuming that these observations are in `U[1...N]`, then computes the EDF test statistics and calls `gofw_ActiveTests0` after each transformation. The function returns the *original* array `U` (the transformations are applied on a copy of `U`). If `printval = TRUE`, prints all the values to the standard output after each iteration. If `graph = TRUE`, calls `gofw_GraphDistUnif` after each iteration to print to file `f` the data for plotting the distribution function of the U_i.

```
void gofw_IterPowRatioTests0 (double U[], long N, int k,
                              lebool printval, lebool graph, FILE *f);
```

Similar to `gofw_IterSpacingsTest0`, but with the `gofs_PowerRatios` transformation.

statcoll

This module contains some basic tools for collecting statistical observations and computing simple statistics on them.

Collector type

```
typedef struct {
   double *V;
   long Dim;
   long NObs;
   char *Desc;
} statcoll_Collector;
```

A collector of real-valued statistical observations. The array `V` has dimensions `Dim + 1` (i.e., elements `V[0]` to `V[Dim]`) and contains `NObs` observations in `V[1]` to `V[NbObs]`. The element `V[0]` can be used for special purposes. The character string `Desc` (max. 127 characters) contains the name of the collector (used for printing reports, etc.). A collector is created by calling `statcoll_Create` and destroyed by calling `statcoll_Delete`. Observations are added one at a time by calling `statcoll_AddObs`.

Prototypes

```
statcoll_Collector * statcoll_Create (long N, const char Desc[]);
```

Creates and returns a collector that can take up to `N` observations. Initializes its fields `Dim` to `N`, `NObs` to 0, `Desc` to `Desc`, and allocates `V[0..Dim]`. (If `Desc` is too long, the description will be truncated). This function must be called for each new collector `statcoll_Collector`. One may call `statcoll_Init` later to reinitialize a collector or to change its dimension.

```
statcoll_Collector * statcoll_Delete (statcoll_Collector *S);
```

Releases the space allocated for arrays `V` and `Desc` in this collector, then deletes the collector, and returns the `NULL` pointer.

```
void statcoll_Init (statcoll_Collector *S, long N);
```

Initializes the collector `S` by setting its observations counter `NObs` to 0. Then ensures that its dimension `Dim` is at least `N` (enlarges the array `V` if needed).

```
void statcoll_SetDesc (statcoll_Collector *S, const char Desc[]);
```

Set the `Desc` field of collector `S` to `Desc`.

`void statcoll_AddObs (statcoll_Collector *S, double x);`

Adds an observation of value `x` to the collector `S`. If the array `V` is already full (`NObs = Dim`), it will be automatically enlarged (`Dim` will be doubled) to accomodate the new observations.

`void statcoll_Write (statcoll_Collector *S, int k, int p1, int p2, int p3);`

Writes the observations currently in collector `S`, `k` values per line, with at least `p1` positions per value, `p2` digits after the decimal point, and `p3` significant digits.

`double statcoll_Average (statcoll_Collector *S);`

Returns the average of the observations currently in collector `S`.

`double statcoll_Variance (statcoll_Collector *S);`

Returns the sample variance of the observations currently in collector `S`, i.e.,

$$S_n^2 = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \bar{X}_N)^2,$$

where $X_1, \ldots, X_N$ are the N observations and $\bar{X}_N$ their average.

`double statcoll_AutoCovar (statcoll_Collector *S, int k);`

Returns the sample autocovariance of lag `k` for the observations currently in collector `S`, i.e.,

$$\hat{\sigma}_k = \frac{1}{N-k}\sum_{i=1}^{N-k}(X_i X_{i+k} - \bar{X}_N^2),$$

where $X_1, \ldots, X_N$ are the N observations and $\bar{X}_N$ their average.

`double statcoll_Covar (statcoll_Collector *S1, statcoll_Collector *S2);`

Returns the sample covariance between the observations in collector `S1` and those in collector `S2`, i.e.,

$$\frac{1}{N-1}\sum_{i=1}^{N}(X_i Y_i - \bar{X}_N \bar{Y}_N),$$

where $X_1, \ldots, X_N$ are the N observations in `S1`, $Y_1, \ldots, Y_N$ are the N observations in `S2`, and $\bar{X}_N$ and $\bar{Y}_N$ are their respective averages. The two collectors must contain the same number of observations.

References

[1] N. H. Anderson and D. M. Titterington. A comparison of two statistics for detecting clustering in one dimension. *Journal of Statistical Computation and Simulation*, 53:103–125, 1995.

[2] T. W. Anderson and D. A. Darling. Asymptotic theory of certain goodness of fit criteria based on stochastic processes. *Annals of Mathematical Statistics*, 23:193–212, 1952.

[3] D. J. Best and D. E. Roberts. Algorithm AS 91: The percentage points of the χ^2 distribution. *Applied Statistics*, 24:385–388, 1975.

[4] G. P. Bhattacharjee. The incomplete gamma integral. *Applied Statistics*, 19:285–287, 1970. AS32.

[5] J. M. Blair, C. A. Edwards, and J. H. Johnson. Rational Chebyshev approximations for the inverse of the error function. *Mathematics of Computation*, 30:827–830, 1976.

[6] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, NY, second edition, 1987.

[7] J. R. Brown and M. E. Harvey. Rational arithmetic MATHEMATICA functions to evaluate the one-sided one-sample K-S cumulative sample distribution. *Journal of Statistical Software*, 19(6):1–32, 2007.

[8] J. R. Brown and M. E. Harvey. Rational arithmetic MATHEMATICA functions to evaluate the two-sided one sample K-S cumulative sample distribution. *Journal of Statistical Software*, 26(2):1–40, 2008.

[9] D. A. Darling. On the theorems of Kolmogorov-Smirnov. *Theory of Probability and Its Applications*, V(4):356–360, 1960.

[10] D. A. Darling. On the asymptotic distribution of Watson's statistic. *The Annals of Statistics*, 11(4):1263–1266, 1983.

[11] T. G. Donnelly. Algorithm 462: Bivariate normal distribution. *Communications of the ACM*, 16(10):638, 1973.

[12] Z. Drezner and G. O. Wesolowsky. On the computation of the bivariate normal integral. *Journal of Statistical Computation and Simulation*, 35:101–107, 1989.

[13] J. Durbin. *Distribution Theory for Tests Based on the Sample Distribution Function*. SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, PA, 1973.

[14] W. Gautschi. Algorithm 222: Incomplete beta function ratios. *Communications of the ACM*, 7(3):143–144, 1964.

[15] W. Gautschi. Certification of algorithm 222: Incomplete beta function ratios. *Communications of the ACM*, 7(3):244, 1964.

[16] A. Genz. Numerical computation of rectangular bivariate and trivariate normal and t probabilities. *Statistics and Computing*, 14:151–160, 2004. See `http://www.math.wsu.edu/faculty/genz/homepage`.

[17] J. Glaz. Approximations and bounds for the distribution of the scan statistic. *Journal of the American Statistical Association*, 84:560–566, 1989.

[18] G. W. Hill. Algorithm 395: Student's t-distribution. *Communications of the ACM*, 13:617–619, 1970.

[19] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 1. Wiley, 2nd edition, 1994.

[20] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 2. Wiley, 2nd edition, 1995.

[21] W. J. Kennedy Jr. and J. E. Gentle. *Statistical Computing*. Dekker, New York, NY, 1980.

[22] M. Knott. The distribution of the Cramér-von Mises statistic for small sample sizes. *Journal of the Royal Statistical Society B*, 36:430–438, 1974.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1998.

[24] S. Lang. *Elliptic functions*. Addison-Wesley, Reading, Mass., 1973.

[25] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, third edition, 2000.

[26] P. A. W. Lewis. Distribution of the Anderson-Darling statistic. *Annals of Mathematical Statistics*, 32:1118–1124, 1961.

[27] K. V. Mardia and P. J. Zemroch. *Tables of the F and Related Distributions with Algorithms*. Academic Press, London, 1978.

[28] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, pages 3–10, North-Holland, Amsterdam, 1985. Elsevier Science Publishers.

[29] G. Marsaglia and J. Marsaglia. Evaluating the Anderson-Darling distribution. *Journal of Statistical Software*, 9(2):1–5, 2004. See `http://www.jstatsoft.org/v09/i02/`.

[30] G. Marsaglia, W. W. Tsang, and J. Wang. Evaluating Kolmogorov's distribution. *Journal of Statistical Software*, 8(18):1–4, 2003. See `http://www.jstatsoft.org/v08/i18/`.

[31] G. Marsaglia, A. Zaman, and J. C. W. Marsaglia. Rapid evaluation of the inverse normal distribution function. *Statistics and Probability Letters*, 19:259–266, 1994.

[32] D. B. Peizer and J. W. Pratt. A normal approximation for binomial, F, beta, and other common related tail probabilities. *Journal of the American Statistical Association*, 63:1416–1456, 1968.

[33] W. Pelz and I. J. Good. Approximating the lower tail-areas of the Kolmogorov-Smirnov one-sample statistic. *Journal of the Royal Statistical Society, Series B*, 38(2):152–156, 1976.

[34] J. Pomeranz. Exact cumulative distribution of the Kolmogorov-Smirnov statistic for small samples (algorithm 487). *Communications of the ACM*, 17(12):703–704, 1974.

[35] T. R. C. Read and N. A. C. Cressie. *Goodness-of-Fit Statistics for Discrete Multivariate Data*. Springer Series in Statistics. Springer-Verlag, New York, NY, 1988.

[36] J. L. Schonfelder. Chebyshev expansions for the error and related functions. *Mathematics of Computation*, 32:1232–1240, 1978.

[37] C. D. Sinclair and B. D. Spurr. Approximations to the distribution function of the Anderson-Darling test statistic. *Journal of the American Statistical Association*, 83(404):1190–1191, 1988.

[38] M. A. Stephens. Use of the Kolmogorov-Smirnov, Cramér-Von Mises and related statistics without extensive tables. *Journal of the Royal Statistical Society, Series B*, 33(1):115–122, 1970.

[39] M. S. Stephens. Tests based on EDF statistics. In R. B. D'Agostino and M. S. Stephens, editors, *Goodness-of-Fit Techniques*. Marcel Dekker, New York and Basel, 1986.

[40] M. S. Stephens. Tests for the uniform distribution. In R. B. D'Agostino and M. S. Stephens, editors, *Goodness-of-Fit Techniques*, pages 331–366. Marcel Dekker, New York and Basel, 1986.

[41] S. R. Wallenstein and N. Neff. An approximation for the distribution of the scan statistic. *Statistics in Medicine*, 6:197–207, 1987.

[42] G. S. Watson. Optimal invariant tests for uniformity. In *Studies in Probability and Statistics*, pages 121–127. North Holland, Amsterdam, 1976.