A Proposal to add Mathematical Functions ***for Statistics*** to the C++ Standard Library

Document number: JTC 1/SC22/WG14/N1069, WG21/N1668
Date:11 Aug 2004
Project: Languages C and C++
References: C ISO/IEC IS 9899:1999, C++ ISO/IEC IS 14882:1998(E),
Reply to: Paul A Bristow, pbristow@hetp.u-net.com, J16/04-0108

Contents

1 Background & motivation

Why is this important? What kinds of problems does it address, and what kinds of programmers is it intended to support? Is it based on existing practice?

C99[ISO:9899] provides an extended <math.h> header and library to provide additional mathematical so-called 'special' functions, in this case erf, erfc, tgamma and lgamma, by P J Plauger G21/N1372 = J16/02-0012 and has been added to the C++ Standard Library TR1.

Walter E Brown has recently made a proposal to add several further 'special functions' similar to the C99 items to C++ (document WG21/N1422=J16/03-0004, in html file n1422.htm and revised in document WG21/N1514 = J16/03-0097 M1514.htm 15 May 2003), now accepted in TR1.

These functions are a small subset of the so-called 'Special Functions', for which the National Bureau of Standards publication Handbook of Mathematical Functions by M Abramowitz and I Stegun in 1964 is of course legendary. D W Lozier and F W L Olver at NIST have (December 2000) re-reviewed the field in their paper "Numerical Evaluation of Special Functions" available at http://math.nist.gov/nesf/ with other reports of an on-going project to update and modernize this guide. They also review the many software packages which provide some, or all of these functions.

C99 math function additions erf, erfc, tgamma and lgamma are useful, but have probably been selected because they are easy to evaluate. The other functions chosen by Walter Brown are also widely used in some mathematics and science areas, but do not include several functions which have very much wider utility in **statistical problems, even the most elementary**. For example, ISO 5725 on “Accuracy Trueness, and Precision of Measurement methods and results”, uses statistical tests which require the functions proposed: this Standard has extremely wide application to almost all measurements of physical, physiological and chemical properties. Similar statistical methods are used for economic, sociological, psychological and other applications.

Basic statistical functions such as Student's t , Chi-squared and Fisher F tests require the incomplete beta function: this is by far the most widely (and implicitly) used in calculating statistical probabilities. For example, to calculate the probability of means (from two groups of observations) being different requires the incomplete beta function (to calculate the probability of Student's t). Without this function, one has to fall back on rather inaccurate lookup tables for an arbitrary and limited set of probabilities.

Sadly, the incomplete beta function is also the most difficult to implement accurately, especially over the full range of arguments: fortunately, even modest accuracy is often all that is needed in practice, in

contrast to other functions, like log and exp, where full machine accuracy is highly desirable. Other functions are generally quite straightforward, often derived simply from the incomplete beta function, and from the erf and gamma functions already proposed (at least for moderate, but acceptable, accuracy). Although the list of functions proposed below may appear long, many functions, for example, complements and inverses, are trivially derived from others.

Many implementations of these functions are found in most statistical and mathematical packages, for example, Matlab, MathCAD, Maple, Mathematica, Visual BASIC and GNU GSL. Lacking these functions, C or C++ and the Standard Library cannot conveniently be used when a statistical calculation is required, often at the end of some prior preprocessing of measured data. As a result, the only expedient solution is to use other tools instead, or some ugly hybrid of a statistical package and C/C++ user-written subroutine. It is believed that the lack of these functions is significantly reducing the application of C and C++ to processing all types of measured data **by programmers whether novice or guru**.

The Cephes package by S Moshier is a good example of an example of **existing practice in C** with various precisions on several platforms (although it is strictly a C implementation and would require repackaging for C++ use). The implementation of Moshier caters for single, double 64-bit, 80-bit and 128-bit functions, showing that these functions are practicable for various C/C++ floating-point representations, and provide useful accuracies. The NAG, IMSL, SGI and GNU Scientific Library also provide C libraries, but none a true C++ version. The recently released Numerical Recipes in C++ by Press et al does not really provide a full C++ implementation as described below because it is a barely modified transfer from the previous book Numerical Recipes in C. But it does provide an example of 'existing practice', and offers some discussion of the implementation options and, most usefully and popularly, practical guidance on how to use the functions: Perhaps a quarter of the book is devoted to functions proposed here. Of the above implementations, only Cephes has an entirely free license, but together they prove feasibility.

2 Impact on the C and C++ Standards

This proposal is for pure additions to existing functions. It does not require any language changes. All the functions are well understood and both C, C++ (and FORTRAN and other language) implementations of many functions exist.

3 Design Decisions

3.1 Implementation

Although the full list of functions reviewed by D W Lozier and F W L Olver would be desirable in the long term, as Walter E Brown remarks in his proposal for some additional C++ headers for 'special functions', the implementation of all is a major task, and might seem daunting to implementers. But a well-established public domain C implementation by Moshier exists which could be used, without licensing difficulties, to allow commercial vendors to easy produce a working library. Other vendors may invest in providing more refined versions, improving accuracy, speed, validation and documentation.

Even modest implementations would still be extremely useful. Computing solely with 64-bit doubles would, at best, probably limit most results to much less than 16 decimal digit accuracy. For a simple implementation, the float and the long double overloads could just return this double result as far as possible. Other implementations might use long double to compute results which might be fully accurate at double precision. Where speed and space are critical, for example embedded systems, an

implementation might compute using only 32-bit floats, and simply return these results for the other double and long double overloads, if required, accepting the limited accuracy.

3.2 Accuracy

One of the major problems facing implementers is accuracy, especially in the asymptotic regions. Unlike some functions like cos and sin, it will be difficult to achieve accuracy close to a few ‘epsilon’ or ‘ulp’ for double, and especially for long double types. In general, it may be necessary to accept with the loss of several decimal digits accuracy, especially if it is necessary to balance computation speed with accuracy. But even 6 decimal digit accuracy would often be much better than current tables, often the accuracy of the ‘reference’ A&S publication.

Validation is also remains a significant problem. Testing with random values, as provided by Moshier, is informative but may be misleading too, for example failing to detect non-monotonicity when the algorithm changes method of calculation. There is a lack of sufficiently highly accurate (40 decimal digits), published and verified test values (even a few spot test values) particularly for more than 64-bit floating-point formats. So I believe it would be premature for the Standard to try to mandate accuracy (or speed) requirements in the Standard (especially when this has not yet been done for pow, exp and log!). But as observed by Kevin Lynch, **documentation on accuracy and speed** is indispensable, if a 'Quality of Implementation' matter.

The other design decisions are broadly the same as those discussed by Walter E Brown in his proposal for other mathematical functions.

3.3 Choice of functions

The functions below are selected because they clearly have very wide utility. This collection would complete the collection of all the so-called ‘special functions’ which are of general utility. The name is misleading - many are of wider practical use than geometric functions like sin and cos (although of course many of the functions do, in turn, use these geometric functions).

Many of the functions are easily derived from, or closely related to those already proposed by Walter E Brown and C99 tgmath functions. It is not proposed that implementation of ALL functions would be mandatory. It is not proposed to mandate how unimplemented functions should be handled: for example, they could just be ignored, return and/or set an error code, or throw an exception.

3.4 Real only - Not complex?

As with Walter Brown's proposal, complex arguments are much less important, much more problematic, and much more difficult to implement, and therefore are not proposed yet. All functions which have real or (preferably unsigned) integer parameter(s) and (corresponding type of) real result. On the other hand, there might be merit in defining the signatures of complex versions, even if they are not yet implemented.

3.5 Overload rather than template

I support (reluctantly) and adopt Brown’s rationale and proposals for choice of overloaded functions rather than function templates and for error codes rather than exceptions. This is a pragmatic decision and accepts that the advantages of a 'proper C++ solution' using exceptions etc. are outweighed by the costs of change and incompatibility with C. Exception throwing versions are not precluded.

3.6 Names

The choice of names is necessarily contentious. The Standard Library conventions of all lower case and underscore as word separator is maintained. Where a C99 name exists, I have maintained it, despite what I feel are some most unfortunate choices. Long names are generally chosen, preferring clarity to curtness. Descriptive names are generally preferred over traditional cryptic single Greek-letter 'mathemations' names : with the rationale that these are most helpful to the more numerous less-specialist users, but will still be familiar to more mathematical users (who might otherwise prefer traditional abbreviated names). The name of function is first (to keep variants together in alphabetical lists), and variants like complement and inverse are differentiated by adding suffixes _c or _inv or _c_inv. The suffix '_distribution' is be added where appropriate for a distribution function, but a preferred naming reflects the major statistical usage as probabilities and quantiles (previously, and perhaps still better known, as percentiles or percentage points). For example, the normal distribution function and its inverse are more helpfully (for statistics users) named as normal_probability and normal_quantile (implicitly with zero mean and unit standard deviation because this is simple and usually implicitly understood).

For example:

```
double normal_distribution (double z); // Normal distribution function.
double normal_probability(double z); // Normal distribution probability.
double normal_distribution_inv (double p); // Normal distribution function inverse.
double normal_quantile(double p); // Normal distribution quantile or percentile.
```

For the convenience of statistical users, some functions will have more than one name, for example the C99 function erf is equivalent to normal_distribution. Although this risks 'polluting' or at least 'cluttering' the C++ std namespace, in practice the compromise seems helpful to the many users to whom the erf name is unfamiliar.

3.7 unsigned integer parameters

Where integer parameters are used, unsigned int is preferred for C++ (unlike existing C versions which use int). This will avoid spurious warnings about conversions between signed and unsigned. In some cases checks for negative parameters like degrees of freedom may be avoided (but sadly not checks that parameters are > 0).
(If using unsigned integers precludes adoption in C libraries, then perhaps signed integers may be preferred, but for C++ unsigned integers do seem to be the 'Right Type').

Degrees of freedom are specified as double because non-integral degrees of freedom can have real meaning. Overloads for unsigned int are proposed as well, for example:

```
double students_t (double df, double t); // Student's t with degrees of freedom as double.
double students_t (unsigned int df, double t); // with degrees of freedom as integer.
```

Of course, checks for negative degrees of freedom are required in the double case, but may not be necessary for unsigned integers.

3.8 Order of parameters

The order of arguments is chosen as suggested by P J Plauger (N1502=03-0085 Proposed signature changes for special math functions in TR-1) placing the independent variable first permits C++ (only) to define sensible defaults for the remaining parameters.

3.9 Non-Central versions

Non-central or asymmetric versions of Student's t and Chi-squared functions are not yet proposed because non-central functions are not widely used and much more difficult to implement. An additional parameter might be added with a default of zero, for example:

```
double students_t (double df, double t, double noncentrality = 0); // Student's t with degrees of freedom as double, and optional non-centrality.
```

But a default zero requirement would preclude adoption into C99, so a function with a different name may be a better choice?

```
double students_t_a(double df, double t, double noncentrality); // Asymmetric Student's t with degrees of freedom as double, and non-centrality.
```

But definition of their signature might still be sensible.

3.10 C compatibility

As suggested by P J Plauger (N1502, for C99 compatibility are approved for TR-1), it would make sense to define the float and long double *f and *l versions of all these functions in C++). And, of course, we should also add the overloads for these functions needed to match the argument promotion rules of the C99 generics. For example, riemann_zeta(2) should call the double version of this function, not cause a compile-time ambiguity.

3.11 Extension to User defined Types

Although this proposal only provides signatures for built-in floating point types, float, double and long double, there should be no difficulty in extending the signatures to cover User Defined Types(UDT). For example, higher accuracy versions might use some 128-bit software floating-point type or an arbitrary precision floating-point type. Or more complicated UDTs might provide more than different precision, for example: intervals (for example, Boost Interval library) or uncertain types, holding a confidence interval, degrees of freedom and distribution information about a measured value.

4 Functions

List of Mathematical Functions considered essential for Statistics.

(To be inserted into Table 80 (clause 26)).

26.x.1 Synopsis

Mathematical 'special' functions

(only double versions are shown, overloads for float and long double will also be provided).

```
double beta_distribution(double a, double b, float x)); // Beta distribution function.
double beta_incomplete (double a, double b, double x); // Incomplete beta integral.
double beta_incomplete_inv (double a, double b, double y); // Inverse of incomplete beta
integral.
double binomial (unsigned int k, unsigned int n, double p); // Binomial distribution function.
double binomial_c (unsigned int k, unsigned int n, double p); // Binomial distribution
function complemented.
double binomial_distribution_inv(unsigned int k, unsigned int n, double y); // Binomial
distribution function inverse.
double binomial_neg_distribution (unsigned int k, unsigned int n, double p); // Negative
binomial distribution .
double binomial_neg_distribution_c (unsigned int k, unsigned int n, double p); // Negative
binomial distribution complement.
double binomial_neg_distribution_inv (unsigned int k, unsigned int n, double p); // Inverse of
negative binomial distribution.
double chebyshev_poly(double x, double* coefficient, unsigned int n); // Evaluate Chebeshev
polynomial.
double chi_sqr_distribution(double df, double x);  // Chi-squared distribution function.
double chi_sqr_distribution_c(double df, double x);  // Chi-squared distribution function
complemented.
double chi_sqr_distribution_c_inv(double df, double p); // Inverse of Chi-squared distribution
function complemented.
double digamma(double x); // psi or digamma function.
double fisher_distribution(unsigned int ia, unsigned int ib, double c); // Fisher F
distribution.
double fisher_distribution_c(unsigned int ia, unsigned int ib, double c); // Fisher F
distribution complemented.
double fisher_distribution_c_inv(double dfn, double dfd, double y); // Inverse of complemented
Fisher F distribution.
double gamma_distribution (double a, double b, double x); // Gamma probability distribution
function.
double gamma_distribution_c (double a, double b, double x); // Gamma probability distribution
function complemented.
double gamma_incomplete (double a, double x); // Incomplete gamma function.
double gamma_incomplete_c (double a, double x); // Incomplete gamma function  complemented.
double gamma_incomplete_inv (double a, double y0); // Inverse of incomplete gamma integral.
double gamma_incomplete_c_inv (double a, double y0); // Inverse of complemented incomplete
gamma integral. double gamma (double x); // gamma function (or tgamma as in C99 math.h?)
double lgamma (double x); // log gamma function name as C99.
double normal_distribution (double a); // Normal distribution function.
double normal_distribution_inv (double a); // Inverse of normal distribution function.
double poisson_distribution (unsigned int k, double m); // Poisson distribution.
double poisson_distribution_c(unsigned int k, double m); // Complemented Poisson distribution.
double poisson_distribution_inv(unsigned int k, double y); // Inverse Poisson distribution.
double students_t (double df, double t); // Student's t.
double students_t_inv (double df, double p); // Inverse of Student's t.
double students_t (unsigned int df, double t); // Student's t.
double students_t_inv(unsigned int df, double p); // Inverse of Student's t.
```

Distribution function probabilities and quantiles

```
double normal_probability(double z); // Probability of quantile z.
double normal_quantile(double p); // Quantile of probability p.
double students_t_probability(double t, double df, double ncp);// Probability of quantile.
double students_t_quantile(double p, double df, double ncp); // Quantile of probability p.
double chi_sqr_probability(double x, double df, double ncp); // Probability of quantile.
double chi_sqr_quantile(double p, double df, double ncp); // Quantile of probability p.
double beta_probability(double x, double a, double b); // Probability of x, a, b.
double beta_quantile(double p, double a, double b); // Quantile of
double fisher_probability(double f, double dfn, double dfd, double ncp); // Probability of
quantile.
double fisher_quantile(double p, double dfn, double dfd, double ncp); // Quantile of
probability p.
double binomial_probability(double x, double n, double pr); // Probability of x.
```

```
unsigned int binomial_first(double p, unsigned int n, double r); // 1st k for probability >= p
double neg_binomial_probability(double x, double n, double pr); // Probability of quantile.
double poisson_probability(double x, double lambda); // Probability of quantile.
double poisson_quantile(double p, double lambda); // Quantile of probability p.
double gamma_probability(double x, double shape, double scale); // Probability of x.
double gamma_quantile(double p, double shape, double scale); // Quantile of probability p.
double smirnov_inv(int n, double p); // Exact Smirnov statistic.

double kolmogorov ( double ); // Kolmogorov statistic.
double kolmogorov_inv (double p); // Kolmogorov statistic inverse.
```

Several functions included in C99 will be required (in overloaded C++ versions for float, double and long double precisions) as proposed by P J Plauger WG21/N1372 2002):

```
double beta(double x, double y); // beta function in C99.
double cbrt (double x); // Cube root in C99.
double log1p (double x); // log1p(x) = log(1+x) in C99.
double expm1 (double x); // expm1(x) = exp(x) - 1 in C99.
double cosm1 (double x); // cosm1(x) = cos(x) - 1 in C99.
double pow(double, int); // integral power in C99.
```

Since polynomial expansion is almost certain to be used in all implementations, it may be logical to require these (in overloaded versions for float, double and long double precisions) as provided by Walter Brown's proposal 26.x.7 Legendre polynomials legendre_Pl and associated legendre_Plm, and also perhaps similar Chebyshev polynomials.

For example:

Cephes C style Descriptions and names are given, abstracted from Stephen Moshier's Cephes 28 documentation http://www.moshier.net/ as an example of an existing well-established implementation.

Draft of Proposed Text

(To be inserted into Table 80 (clause 26) after removal of references to Cephes which are temporarily included for convenience). Section numbers will also be required of course.

List of Mathematical Functions essential for Statistics.

Beta function

Provided by Walter Brown Proposal 26.x19 as:

```
float beta(float x, float y):
double beta(double x, double y);
long double beta(long double x, long double y)
```

Cephes `double a, b, y, beta(); y = beta( a, b )`

Description:

$$B(a,b) = \int_0^1 t^{a-1}(1-t)^b\,dt$$

$$= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

For large arguments, the logarithm of the function is evaluated using log gamma(), then exponentiated.

Beta distribution

```
float beta_distribution(float a, float b, float x); // Beta distribution function.
double beta_distribution(double a, double b, float x)); // Beta distribution function.
long double beta_distribution(long double a, long double b, float x)); // Beta distribution
function.
```

Cephes `double a, b, x, y, btdtr();  y = btdtr( a, b, x );`

Description:
Returns the area from zero to x under the beta density function which is given by

$$p_x(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

The area from zero to x under the beta density is the integral:

$$p(x(= \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1}(1-t)^{b-1}\,dt$$

The beta distribution function is identical to the incomplete beta integral function beta_incomplete (a, b, x).

$$P(x) = I_x(a,b)$$

The complemented function gives the area under the right tail of the distribution function:

$$1 - P(1-x) = 1 - I_x(a,) = I_{1-x}(b,a)$$

= beta_incomplete (b, a, x);

```
float beta_distribution_c(float a, float b, float x); // Beta distribution function
complement.
double beta_distribution_c(double a, double b, float x)); // Beta distribution function
complement.
long double beta_distribution_c(long double a, long double b, float x)); // Beta distribution
function complement.
```

Incomplete beta integral

```
double beta_incomplete (double a, double b, double x); // Incomplete beta integral.
```
Cephes `double a, b, x, y, incbet(); y = incbet( a, b, x );`

Description:
Returns incomplete beta integral of the arguments, evaluated from zero to x.
The function is defined as:

$$I_x(a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

The domain of definition is 0 <= x <= 1.
(In the Cephes implementation, a and b are restricted to positive values).

The integral from x to 1 may be obtained by the symmetry relation:

1 - incbet(a, b, x) = incbet(b, a, 1-x).

The integral is evaluated by a continued fraction expansion or, when b*x is small, by a power series.

Inverse of incomplete beta integral

```
double beta_incomplete_inv (double a, double b, double y); // Inverse of incomplete beta
integral.
```
Cephes `double a, b, x, y, incbi(); x = incbi( a, b, y );`

Description:
Given y, the function finds x such that
beta_incomplete_inv (a, b, x) = y .
Performs interval halving or Newton iterations to find the root of beta_incomplete_inv (a,b,x) - y = 0.

Binomial distribution

```
float binomial (unsigned int k, unsigned int n, float p); // Binomial distribution function.
double binomial (unsigned int k, unsigned int n, double p); // Binomial distribution function.
long double binomial (unsigned int k, unsigned int n, long double p); // Binomial distribution
function.
```

Cephes `int k, n; double p, y, bdtr(); y = bdtr( k, n, p );`

Description:
Returns the sum of the terms 0 through k of the Binomial probability density:

$$bdtr(k,n,p) = \sum_{j=0}^{k} \binom{n}{j} p^j (1-p)^{n-j}$$

$$= I_{1-p}(n-k, k+1)$$

The terms are not summed directly; instead the incomplete beta integral is employed, according to the formula:

y = bdtr(k, n, p) = incbet(n-k, k+1, 1-p).

The arguments must be positive, with p ranging from 0 to 1.

Complemented binomial distribution

```
float binomial_c (unsigned int k, unsigned int n, float p); // Binomial distribution function
complement.
double binomial_c (unsigned int k, unsigned int n, double p); // Binomial distribution
function complement.
long double binomial_c (unsigned int k, unsigned int n, long double p); // Binomial
distribution function complement.
```

Cephes `int k, n; double p, y, bdtrc(); y = bdtrc( k, n, p );`

Description:
The complementary function returns the sum of the terms k+1 through n of the Binomial probability density:

$$bdtrc(k,n,p) = \sum_{j=k+1}^{n} \binom{j}{j} p^{j(1-p)^{n-j}}$$

The terms are not summed directly; instead the incomplete beta integral is employed, according to the formula

y = bdtrc(k, n, p) = incbet(k+1, n-k, p).

The arguments must be positive, with p ranging from 0 to 1.

Inverse binomial distribution

```
float binomial_distribution_inv(unsigned int k, unsigned int n, float y); // Binomial
distribution function inverse.
double binomial_distribution_inv(unsigned int k, unsigned int n, double y); // Binomial
distribution function inverse.
long double binomial_distribution_inv(unsigned int k, unsigned int n, long double y); //
Binomial distribution function inverse.
```

Cephes `int k, n; double p, y, bdtri(); p = bdtr( k, n, y );`

Description:
Finds the event probability p such that the sum of the terms 0 through k of the Binomial probability density is equal to the given cumulative probability y.

This is accomplished using the inverse beta integral function and the relation

1 - p = incbi(n-k, k+1, y).

Chi-square distribution

```
double chi_sqr_distribution(double df, double x); // Chi-squared distribution function.
```
Cephes `double df, x, y, chdtr(); y = chdtr( df, x);`

Description:
Returns the area under the left hand tail (from 0 to x) of the Chi squared probability density function with v degrees of freedom.

$$chdtr(df, x) = \frac{1}{2^{df/2}\Gamma(df/2)}\int_0^x t^{df/2-1}e^{-t/2}dt$$

$$= P(df/2.x/2)$$

where x is the Chi-square variable.

The incomplete gamma integral is used, according to the formula:

y = chdtr(v, x) = igam(v/2.0, x/2.0).

The arguments must both be positive.

Complemented Chi-square distribution

```
double chi_sqr_distribution_c(double df, double x); // Chi-squared distribution function
complemented.
```
Cephes `double v, x, y, chdtrc(); y = chdtrc( v, x );`

Description:
Returns the area under the right hand tail (from x to infinity) of the Chi squared probability density function with v degrees of freedom:

$$chdtr(df, x) = \frac{1}{2^{df/2}\Gamma(df/2)} \int_x^\infty t^{df/2-1} e^{-t/2} dt$$

where x is the Chi-squared variable.

The incomplete gamma integral is used, according to the formula:

y = chdtr(v, x) = igamc(v/2.0, x/2.0)

The arguments must both be positive.

Inverse of complemented Chi-squared distribution

```
double chi_sqr_distribution_c_inv(double df, double p); // Inverse of Chi-squared distribution function complemented.
```
Cephes `double df, x, y, chdtri(); x = chdtri( df, y );`

Description:
Finds the Chi-square argument x such that the integral from x to infinity of the Chi-squared density is equal to the given cumulative probability y.

This is accomplished using the inverse gamma integral function and the relation

x/2 = igami(df/2, y);

F distribution

```
double fisher_distribution(unsigned int ia, unsigned int ib, double c); // Fisher F distribution.
```
Cephes `int df1, df2; double x, y, fdtr(); y = fdtr( df1, df2, x );`

Description:

Returns the area from zero to x under the Fisher F density function (also known as Snedcor's density or the variance ratio density). This is the density of

$$x = \frac{u_1/df1}{u_2/df2}$$

where u and u2 are random variables having Chi square distributions with df1 and df2 degrees of freedom, respectively.

The left tail area from 0 to x under the F density is:

$$fdtr(df1, df2, F) = I_w(df1/2, df2/2)$$

where

$$w = \frac{df1 F}{df2 + df1 F}$$

The incomplete beta integral is used, according to the formula

$$P(x) = incbet(df_1/2, df_2/2, df_1 \cdot x/(df_2 + df_1 \cdot x))$$

The arguments a and b are greater than zero, and x is non-negative.

Complemented F distribution

```
double fisher_distribution_c(unsigned int ia, unsigned int ib, double c); // Fisher F distribution complemented.
```

Cephes `int df1, df2; double x, y, fdtrc(); y = fdtrc( df1, df2, x );`

Description:
Returns the right tail area from x to infinity under the F density function
(also known as Snedecor's density or the variance ratio density).

$$1 - P(x) = 1/B(a,b)\int_x^\infty t^{(a-1)}(1-t)^{b-1}.dt$$

The incomplete beta integral is used, according to the formula:

$$P(x) = incbet(df_2/2, df_1/2, (df_2/(df_2 + df_1 \cdot x))$$

$$fdtrc(d_1, d_2, F) = I_\nu(d_1/2, d_2/2)$$

where

$$\nu = \frac{d_2}{d_2 + d_1 F}$$

Inverse of F distribution

```
double fisher_distribution_inv(double dfn, double dfd, double y); // Inverse of Fisher F distribution.
```

Description:
Finds the F density argument x such that the integral from zero to x (left tail) of the F density is equal to the given probability p.

This is accomplished using the inverse beta integral function and the relations

$$z = incbi(df_1/2, df_2/2, p)$$
$$x = df_2 \cdot z/(df_1 \cdot (1-z))$$

Probability p must be > zero and p <=unity.

Inverse of complemented F distribution

```
double fisher_distribution_c_inv(double dfn, double dfd, double y); // Inverse of complemented Fisher F distribution.
```
Cephes `int df1, df2; double x, p, fdtrci(); x = fdtrci( df1, df2, p );`

Description:
Finds the F density argument x such that the integral from x to infinity (right tail) of the F density is equal to the given probability p.

This is accomplished using the inverse beta integral function and the relations

$$z = incbi(df_2/2, df_1/2, p)$$
$$x = df_2(1-z)/(df_1 \cdot z)$$

Gamma function

```
double gamma (double x); // gamma function (or tgamma in C99 math.h)
```
Cephes `double x, y, gamma(); y = gamma( x );`

Description:

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

Returns gamma function of the argument. The result is correctly signed, and the sign (+1 or -1)

Arguments |x| <= 34 are reduced by recurrence and the function approximated by a rational function of degree 6/7 in the interval (2,3). Large arguments are handled by Stirling's formula. Large negative arguments are made positive using a reflection formula. See also lgamma.

Natural logarithm of gamma function

```
double lgamma (double x); // log gamma function as C99.
```
Cephes `double x, y, lgam(); extern int sgngam; y = lgam( x );`

Description:
Returns the base e (2.718...) logarithm of the absolute value of the gamma function of the argument. Cephes returns a sign (+1 or -1) of the gamma function in a global (extern) variable named sgngam, but this is not provided by the C99 or C++equivalents.

For arguments greater than 13, the logarithm of the gamma function is approximated by the logarithmic version of Stirling's formula using a polynomial approximation of degree 4. Arguments between -33 and +33 are reduced by recurrence to the interval [2,3] of a rational approximation. The cosecant reflection formula is employed for arguments less than -33.

Gamma distribution function

```
double gamma_distribution (double a, double b, double x); // Gamma probability distribution function.
```
Cephes `double a, b, x, y, gdtr(); y = gdtr(a, b, x );`

Description:
Returns the integral from zero to x of the gamma probability density function:

$$gdtr(a,b,x) = \frac{a^b}{\Gamma(b)} \int_0^x t^{b-1} e^{-at} dt$$

The incomplete gamma integral is used, according to the relation

$$y = igam(b, ax)$$

Complemented gamma distribution function

```
double gamma_distribution_c (double a, double b, double x); // Gamma probability distribution function complemented.
```
Cephes `double a, b, x, y, gdtrc(); y = gdtrc( a, b, x );`

Description:
Returns the integral from x to infinity of the gamma probability density function:

$$gdtr(a,b,x) = \frac{a^b}{\Gamma(b)} \int_x^\infty t^{b-1} e^{-at} dt$$

The incomplete gamma integral is used, according to the relation:

$$y = igamc(b, a \cdot x)$$

Incomplete gamma integral

```
double gamma_incomplete (double a, double x); // Incomplete gamma function.
```
Cephes `double a, x, y, igam(); y = igam( a, x );`

Description:
The function is defined by

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

$$= \frac{x^a e^{-x}}{\Gamma(a+1)} F_1(1, a+1, x)$$

Where F_1 is the confluent hypergeometric function.

Both arguments a and x must be positive. The integral is evaluated by either a power series or continued fraction expansion, depending on the relative values of a and x.

Complemented incomplete gamma integral

```
double gamma_incomplete_c (double a, double x); // Incomplete gamma function complemented.
```
Cephes `double a, x, y, igamc(); y = igamc( a, x );`

Description:
The function is defined by

$$igamc(a,x) = 1 - igam(a,x)$$

$$P^*(a,x) = 1 - P(a,x)$$

Both arguments a and x must be positive. The integral is evaluated by either a power series or continued fraction expansion, depending on the relative values of a and x.

Inverse of complemented incomplete gamma integral

```
double gamma_incomplete_c_inv (double a, double y0); // Inverse of complemented incomplete
gamma integral.
```
Cephes `double a, x, p, igami(); x = igami( a, p );`

Description:
Given p, the function finds x such that

$$igamc(a,x) = p$$

This can be accomplished starting with the approximate value 3

$$x = a \cdot t$$

where

$$t = 1 - d - ndtri(p) \cdot \sqrt{d}$$

and

$$d = 1/9 \cdot a$$

and performing Newton iterations to find the root of igamc(a,x) - p = 0.

Negative binomial distribution

```
double binomial_neg_distribution (unsigned int k, unsigned int n, double p); // Negative
binomial distribution .
```
Cephes `int k, n; double p, y, nbdtr(); y = nbdtr( k, n, p );`

Description:
Returns the sum of the terms 0 through k of the negative binomial distribution:

$$nbdtr(k,n,p) = \sum_{j=0}^{k} \binom{n+j-1}{j} p^n (1-p)^j$$

In a sequence of Bernoulli trials, this is the probability that k or fewer failures precede the nth success.

The terms are not computed individually; instead the incomplete beta integral is employed, according to the formula

$$y = nbdtr(k,n,p) = incbet(n,k+1,p)$$

The arguments k and n must be positive, with p ranging from 0 to 1.

Complemented negative binomial distribution

```
double binomial_neg_distribution_c (unsigned int k, unsigned int n, double p); // Negative binomial distribution complement.
```

Cephes `int k, n; double p, y, nbdtrc(); y = nbdtrc( k, n, p );`

Description:
Returns the sum of the terms k+1 to infinity of the negative binomial distribution:

$$nbdtrc(k,n,p) = \sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^n (1-p)^j$$

$$= 1 - I_p(n,k+1)$$
$$= I_{1-p}(k+1,n)$$

The terms are not computed individually; instead the incomplete beta integral is employed, according to the formula:

$$y = nbdtrc(k,n,p) = incbet(K=1,n,1-p)$$

The arguments k and n must be positive, with p ranging from 0 to 1.

Functional inverse of negative binomial distribution

```
double binomial_neg_distribution_inv (unsigned int k, unsigned int n, double p); // Inverse of negative binomial distribution.
```

Cephes `int k, n; double p, y, nbdtri(); p = nbdtri( k, n, y );`

Description:
Finds the argument p such that nbdtr(k, n, p) is equal to y.

Normal Gaussian distribution function

```
double normal_distribution (double a); // Normal distribution function.
double normal_probability (double a); // Normal distribution function.
```
Cephes `double x, y, ndtr(); y = ndtr( x );`

Description:
Returns the area under the Gaussian probability density function, integrated from minus infinity to x:

$$erf(x) = \frac{2}{\sqrt{\pi}} 0\int_{0}^{x} e^{-t^2} dt$$

The Gaussian or normal probability distribution function is

$$N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{t^2/2} dt$$

$$= \frac{1 + erf(x/\sqrt{2})}{2}$$

$$= 1 - \frac{erfc(x/\sqrt{2})}{2}$$

$$= (1 + erf(z))/2 = erfc(z)/2$$

where

$$z = x/\sqrt{2}$$

Computation is via the functions erf and erfc.

Although this function may be calculated using erf and/or erfc, it is considered better to define it separately:

1. To facilitate the most accurate implementation.
2. For clarity and convenience of users.

Inverse of Normal distribution function

```
double normal_distribution_inv (double a); // Inverse of normal distribution function.
```
Cephes `double x, y, ndtri(); x = ndtri( y );`

Description:
Returns the argument, x, for which the area under the Gaussian probability density function (integrated from minus infinity to x) is equal to y.

For small arguments 0 < y < exp(-2), the program computes
z = sqrt(-2.0 * log(y)); then the approximation is
x = z - log(z)/z - (1/z) P(1/z) / Q(1/z).

There are two rational functions P/Q, one for 0 < y < exp(-32)
and the other for y up to exp(-2). For larger arguments,
w = y - 0.5, and x/sqrt(2pi) = w + w**3 R(w**2)/S(w**2)).

Poisson distribution

```
double poisson_distribution (unsigned int k, double m); // Poisson distribution.
```
Cephes `int k; double m, y, pdtr(); y = pdtr( k, m );`

Description:
Returns the sum of the first k terms (area under the left tail) of the Poisson distribution:

$$pdtr(k,\lambda) = \sum_{j=0}^{k} \frac{e^{-\lambda}\lambda^{j}}{j!}$$

The terms are not summed directly; instead the incomplete gamma integral is employed, according to the relation:

$$pdtr(k,m) = igamc(k+1,m)$$

The arguments must both be positive.

Complemented Poisson distribution

```
double poisson_distribution_c(unsigned int k, double m); // Complemented Poisson distribution.
```
Cephes `int k; double m, y, pdtrc(); y = pdtrc( k, m );`

Description:
Returns the sum of the terms k+1 to infinity (area under the right tail) of the Poisson distribution:

$$pdtrc(k,\lambda) - \sum_{j=k+1}^{\infty} \frac{e^{-\gamma}\lambda^{j}}{j!} = P(k+1,\lambda)$$

The terms are not summed directly; instead the incomplete gamma integral is employed, according to the formula

$$y = pdtrc(k,m) = igam(k+1,m)$$

The arguments k and m must both be positive.

Inverse Poisson distribution

```
double poisson_distribution_inv(unsigned int k, double y); // Inverse Poisson distribution.
```
Cephes `int k; double m, y, pdtr(); m = pdtri( k, y );`

Description:
Finds the Poisson variable x such that the integral from 0 to x of the Poisson density is equal to the given probability y.

This can be found using the inverse gamma integral function and the relation

$$m - igami(k+1, y)$$

Psi (digamma) function

```
double digamma(double x);
```
Cephes `double x, psi(); y = psi( x );`

Description:

$$psi(x) = \frac{d}{dx} \ln \Gamma(x)$$

digamma is the logarithmic derivative of the gamma function.
For integer x,

$$psi(n) = -EUL + \sum_{k=1}^{n-1} \frac{1}{k}$$

This formula is used for 0 < n <= 10. If x is negative, it is transformed to a positive argument by the reflection formula:

$$psi(1-x) = psi(x) + \pi \cdot \cot(\pi \cdot x)$$

For general positive x, the argument is made greater than 10 using the recurrence:

$$psi(x+1) = psi(x) + 1/x$$

Then the following asymptotic expansion is applied:

$$psi(x) = \log(x) - 1/2x - \sum_{k=1}^{\infty} B_{2k} / (2k \cdot x^{2k})$$

where the B2k are Bernoulli numbers.

Student's t distribution

```
double students_t _probability(double df, double t); // Student's t.
double students_t _probability(unsigned int df, double t); // Student's t.
double students_t (double df, double t); // Student's t.
double students_t (unsigned int df, double t); // Student's t.
```
Cephes `double t, stdtr(); short k; y = stdtr( k, t );`

Description:
Computes the integral from minus infinity to t of the Student's t distribution with integer k > 0 degrees of freedom:

$$stdtr(k,t) = \frac{\Gamma((k+1)/2)}{\sqrt{k\pi}\cdot\Gamma(k/2)}\int_{-\infty}^{t}(1+x^2/k)^{-(k+1)/2}dx$$

For t < -2, this may be computed using the relationship to the incomplete beta function:

$$1 - stdtr(k,t) = \frac{1}{2}\cdot incbet(\frac{k}{2},\frac{1}{2},z)$$

where

$$z = k/(k+t^2)$$

For higher t, a direct method is derived from integration by parts.

Since the function is symmetric about t=0, the area under the right tail of the density is found by calling the function with -t instead of t.

Functional inverse of Student's t distribution

```
double students_t_inv (double df, double p); // Inverse of Student's t.
double students_t_inv(unsigned int df, double p); // Inverse of Student's t.
double students_t_quantile(double df, double p); // Inverse of Student's t.
double students_t_quantile (unsigned int df, double p); // Inverse of Student's t.
```

Cephes `double p, t, stdtri(); int k; t = stdtri(k, p );`

Description:
Given probability p, finds the argument t such that stdtr(k,t) is equal to p.

Math Constants required (Boost proposal in progress)

(but 40 decimal digit accuracy values will be required to include 128-bit floating point types.)

PI = 3.14159265358979323846 pi
PIO2 = 1.57079632679489661923 pi/2
PIO4 = 7.85398163397448309616E-1 pi/4
SQRT2 = 1.41421356237309504880 sqrt(2)
SQRTH = 7.07106781186547524401E-1 sqrt(2)/2
LOG2E = 1.44269504088896340735992 1/log(2)
SQ2OPI = 7.97884560802865355879892E-1 sqrt(2/pi)
LOGE2 = 6.93147180559945309417E-1 log(2)
LOGSQ2 = 3.46573590279972654709E-1 log(2)/2
THPIO4 = 2.35619449019234492885 3*pi/4
TWOOPI = 6.36619772367581343075535E-1 2/pi

5 Acknowledgements

In the Boost community forum, expert views on the presentation format of mathematical (and other) functions were expressed by:

Kevin Lynch
Physics Department, Boston University Boston, MA 02215 USA http://physics.bu.edu/~krlynch
and
Eric Ford.

6 References

Walter E Brown, "Proposal to add Mathematical Special Functions to the C++ Standard Library.", WG21/N1422,
WG21/N1514 = J16/03-0097 M1514.htm 15 May 2003, and many references therein.

P J Plauger, WG21/N1502=03-0085 Proposed signature changes for special math functions in TR-1.

Numerical Evaluation of Special Functions, December 2000
D. W. Lozier & F. W. J. Olver, in Walter Gautschi (ed)
Mathematics of computation 1943-1993,
Proceedings of Symposia in Applied Mathematics,
American Mathematical Society, Providence RI, USA 02940

"Numerical Evaluation of Special Functions" available at http://math.nist.gov/nesf/

and on-line information on the DMLF project led by NIST.

D. W. Lozier
Mathematical and Computational Sciences Division, National Institute of Standards and Technology
Gaithersburg, Md 20899-8910
dlozier@nist.gov

and F. W. J. Olver
Institute for Physical Science and Technology
University of Maryland
College Park, MD 20742
olver@bessel.umd.edu

S L B Moshier, Methods and Programs for Mathematical Functions, Ellis Horwood Ltd, (1989). ISBN 0135789982 (and 013578980X paperback).

S Moshier, http://www.moshier.net

Milton Abramowitz & Irene Stegun (eds), Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables, volume 55 of National Bureau of Standards Applied Mathematics Series. Reprinted with corrections by Dover 1972, http://members.fortunecity.com/aands

W H Press, W T Vetterling, Saul A Teukolsky, Brian P Flannery, Numerical Recipes in C++, 2nd ed, (2003) ISBN 0521 750332 4.