A Proposal to Add Mathematical *Special Functions* to the C++ Standard Library (version 3)

Document number:	WG21/N1542 = J16/03-0125
Date:	October 28, 2003
Revises:	WG21/N1514 = J16/03-0097
Project:	Programming Language C++
Reference:	ISO/IEC IS 14882:2003(E)

Reply to:	Walter E. Brown <wb@fnal.gov>
	CEPA Dept., Computing Division
	Fermi National Accelerator Laboratory
	Batavia, IL 60510-0500

Contents

---

1. Background and 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?

A. Introduction

Compared to C++ [ISO:14882], C99 [ISO:9899] provides an extended `<math.h>` header and library. Among the additions introduced by C99 are selected mathematical functions from categories of particular interest to the numerical computing communities: *exponential and logarithmic functions*, *circular and hyperbolic functions*, and *special functions*.

In particular, C99 specifies the following traditional and extended functions of numerical interest in `<math.h>`, each with variants to accomodate arguments of types `float`, `double`, and `long double`:

- *circular*, a.k.a. *trigonometric* (§7.12.4): `sin`, `cos`, `tan`, `asin`, `acos`, `atan` and `atan2`;
- *hyperbolic* (§7.12.5): `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`;
- *exponential* (§7.12.6): `exp`, `exp2`, `frexp`, `ldexp`, `expm1`;
- *logarithmic* (§7.12.6): `log10`, `log2`, `logb`, `ilogb`, `log1p`;
- *power* (§7.12.7): `pow`, `sqrt`, `cbrt`, `hypot`; and
- *special* (§7.12.8): `erf`, `erfc`, `tgamma`, `lgamma`.

All these functions either (a) are already part of the C++ standard library, or (b) have already been proposed [Plauger2002] and discussed [Dawes, Item 1] for incorporation into the forthcoming C++ Library Technical Report [Josuttis]. **We here propose to augment the C++ standard library with additional functions from the above *Special Functions* category**.

B. Prior Art and Suitability for Technical Report

Mathematical *Special Functions* have been extensively studied for well over a century. They and their applications are routinely taught as parts of required courses of study in scientific, engineering, and mathematical disciplines. Even a cursory bibliography includes such respected works as [Abramowitz], [Hildebrand], [Jackson], [Lebedev], [Spanier], and [Whittaker].

There is also a significant history of implementation experience with these functions, as evidenced, for example, by section C of the Fortran-based SLATEC Common Mathematical Library [SLATEC]. Today, *Special Functions* constitute important subsets of such well-respected add-on libraries as the *NAG C Library* [NAG], the *IMSL C Numerical Library* [IMSL], and the *GNU Scientific Library* [Galassi]. Further, some standard C libraries such as the SGI C library [SGI] and the GNU C library [GNU C] also provide, as extensions, a few of the proposed *Special Functions*.

These functions are appropriate for this TR because they plug an obvious hole ("Filling Gaps," as [Austern] phrases it) in the existing standard library. While these functions are clearly numerical in nature and will likely be most heavily used by the scientific and engineering communities, other communities of programmers also have needs, ranging from frequent to intermittent, for these functions.

This *Special Functions* proposal additionally falls into the "Standards Coordination and Infrastructure" categories identified in [Austern] as targets for the TR, for this proposal is based on an existing standard, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology* [ISO:31-11]. We draw particular attention to the tables constituting paragraphs 8 ("Exponential and logarithmic functions"), 9 ("Circular and hyperbolic functions"), 10 ("Complex numbers"), and 14 ("Special functions").

C. The Prospective User Community

Quantifying the scientific and mathematical programming community of users is a difficult task. While some feel that the size of this specialized group is relatively small compared to the size of the programming community as a whole, we would respectfully point out that this user community is demonstrably sufficiently large to support at least two commercial vendors (IMSL, NAG), a major public-domain project (GSL), as well as large, domain-specific libraries (*e.g.*, kernlib) and vendor-specific libraries, all of which incorporate significant *Special Functions* components.

Support for *Special Functions* has not waned in over 35 years, across a broad spectrum of significant numeric programming languages. Further, continuing interest in this field is demonstrated by an ongoing publication stream on these and related topics in journals (such as those sponsored by ACM, IEEE, MAA, and SIAM) devoted to numeric computing.

D. Why Should *Special Functions* be Standardized?

The mathematics portion of the standard library (`<math.h>`) has been hardly touched since C's earliest days, over 30 years ago. It is arguably well past the time that enhancements in this area ought be considered, as C99 has done. Mathematical *Special Functions* provide a very natural route to such extension.

While a number of libraries (see above) do incorporate *Special Functions*, no one library's coverage appears complete. Combining libraries is generally infeasible; there is a lack of inter-library consistency in naming conventions, calling sequences, and the like. In consequence, users must often (re-)invent missing functionality. The result of such an *ad hoc* approach is often characterized by a lack of generality in the context of a user's specific problem to be solved, as well as by insufficient attention to such details as corner cases, treatement of errors, and the like.

The benefits of incorporating *Special Functions* into the C++ Standard Library include predictability of interface and behavior across a broad spectrum of implementations, leading to improved portability and interoperability for applications that make use of these functions. In addition, users obtain professional attention to issues affecting quality and reliability, important details often overlooked by typical application programmers. This allows users to focus on their problems rather than on issues related to infrastructure or platform dependency.

Finally, we believe that adoption of this proposal would send a clear message to the various numeric computing communities that, contrary to significant popular belief within these communities, C++ is an eminently suitable programming language for their problem domain, too. General availability of the functions herein proposed would greatly enhance and promote C++ usage among computing communities in the scientific, engineering, and mathematical disciplines.

2. Impact On the C++ Standard

> *What does it depend on, and what depends on it? Is it a pure extension, or does it require changes to standard components? Does it require core language changes?*

This proposal is a pure extension. It does not require any changes in the core language. It does not require changes to any standard classes or functions. It does not require changes to any of the standard requirement tables. All the proposed functions are mathematically well-understood, all have proven their utility in practice over a considerable period of time, and all have been previously implemented in C and C++.

This proposal does not depend on any other C++ library extensions. This proposal potentially overlaps slightly with another proposal [Plauger2002] that would incorporate the bulk of C99's library additions into C++. However, the potential commonality between the two proposals is limited to a rather tiny part of `<math.h>` that is essentially identical in the two proposals; see §5. for details.

3. Design Decisions

> *Why did you choose the specific design that you did? What alternatives did you consider, and what are the tradeoffs? What are the consequences of your choice, for users and implementors? What decisions are left up to implementors? If there are any similar libraries in use, how do their design decisions compare to yours?*

A. How to Package the Additional Declarations?

Following the precedent set by C99, this proposal recommends that declarations for all the proposed *Special Functions* be incorporated into `<math.h>` and thence extended into `<cmath>` in the obvious way.

An alternative design would present these additional declarations in a new header. Obvious names for this header, *e.g.*, `<special_functions.h>`, seem unwieldy, and no suitably descriptive shorter names have come to mind. Further, it seems likely that implemention of some of the *Special Functions* can make advantageous use of extant functionality in `<math.h>` and so it seemed reasonable to avoid the separation.

Function name	[ISO:31-11]	[Abramowitz]	[ISO:9899]
`assoc_laguerre`	14.13	§13.6.9, *etc.*	
`assoc_legendre`	14.9	§8.6.6, *etc.*	
`beta`	14.20	§6.2	
`comp_ellint_1`	14.16	§17.3.1, *etc.*	
`comp_ellint_2`	14.17	§17.2.9, *etc.*	
`comp_ellint_3`	14.18	§17.7.2, *etc.*	
`conf_hyperg`	14.15	§13.1.2, *etc.*	
`cyl_bessel_i`	14.4	§9.6.3, *etc.*	
`cyl_bessel_j`	14.1	§9.1.10, *etc.*	
`cyl_bessel_k`	14.4	§9.6.4, *etc.*	
`cyl_neumann`	14.2	§9.1.2, *etc.*	
`ellint_1`	14.16	§17.2.6, *etc.*, with $sin^2\alpha = k^2$	
`ellint_2`	14.17	§17.2.9, *etc.*	
`ellint_e`	14.18	§17.7.1, *etc.*	
`erfc`	14.22	§7.1.2, *etc.*	§7.12.8.2
`erf`	14.22	§7.1.1, *etc.*	§7.12.8.1
`expint`	14.21	§5.1.2, *etc.*	
`hermite`	14.11	§13.6.17 and §13.6.18, *etc.*	
`hyperg`	14.14	§15.1.1, *etc.*	
`laguerre`	14.12	§13.6.9, *etc.*	
`legendre`	14.8	§8.6.18, *etc.*	
`riemann_zeta`	14.23	§23.2.1 and §23.2.6, *etc.*	
`sph_bessel`	14.5	§10.1.1, *etc.*	
`sph_legendre`	14.10	§8.1.1 footnote 2, *etc.*	
`sph_neumann`	14.6	§10.1.1, *etc.*	
`tgamma`	14.19	§6.1.1, *etc.*	§7.12.8.4

Table 1: Alphabetized summary, with references, of proposed *Special Functions*

B. Which *Special Functions* to Incorporate?

Because the set of mathematical functions that can be considered *Special Functions* is potentially unbounded, we considered several options in selecting our list of candidates (see Table 1) for standardization.

This proposal recommends adoption of the list of *Special Functions* specified in [ISO:31-11, paragraph 14]. This list has already received international scrutiny and endorsement via the standardization process. Further consultations with respected scientific colleagues have confirmed that these *Special Functions* would, if incorporated into the C++ standard library, constitute a significant contribution to the numeric community.

A second possibility was the adoption of all the *Special Functions* listed in the (exhaustive!) *Handbook of Mathematical Functions* [Abramowitz], the generally-accepted standard reference for this domain. However, a careful inspection of this work's extensive contents strongly suggests that its scope may be overly broad. More importantly, many of the listed functions appear to be very difficult to implement. (Indeed, a colleague suggested, not entirely in jest, that several Ph.D. dissertations could result from such efforts!)

We considered, third, adopting a list taken from an existing library in this domain. For example, the *Special Functions* portion of the GNU Scientific Library [Galassi] seemed to present a reasonable set for consideration. Indeed, these functions largely constitute a superset of the list recommended above, and are all clearly implementable. However, we felt it advantageous to require only the functions in the above list, in order to allow implementers the freedom to add value by providing additional functions.

A final possibility was the construction of an *ad hoc* list of *Special Functions*. We rejected this as the least defensible of the alternatives since, other than a feel for general utility, there are no obvious criteria for accepting some functions and rejecting others,

C. Function Templates or Overloaded Functions?

It is possible to declare the desired additional `<cmath>` functionality in either of two ways: as (specialized) function templates or as families of overloaded functions. We recommend overloading.

This recommendation is based in significant part on arguments favoring consistency of form with the existing contents of the affected headers: There are no templates today in `<math.h>` or in `<cmath>`. Further, we are unaware of any current *Special Functions* implementation that is based on template technology.

Additional reasons take into account the possible future extension of the new functions to additional headers (such as, for example, `<complex>`). To do so in the presence of function templates would raise such issues as the location of the primary template and the concomitant need to coordinate multiple cooperating headers. We prefer to avoid such entanglements.

D. Exceptions or Error Codes?

Most of the proposed functions must advise their callers of domain and/or range errors. In the context of C++, this would arguably be best done by throwing appropriate exceptions.

However, no standard functions in `<cmath>` today throw exceptions. Rather, mimicking the behavior of the functions in `<math.h>`, each sets a global `errno` variable to a suitable code (*i.e.*, `EDOM` or `ERANGE`) defined in `<cerrno>`.

We recommend that this existing behavior be preserved with respect to the proposed new functions. Not only is this a matter of consistency, but it preserves the possibility that a compatible version of this proposal might be incorporated into a future revision of C99.

As a consequence of this recommendation, we have assumed that general provisions (such as those provided in C99's §7.12.1) would be available to govern overall treatment of domain and range errors. In addition, we have carefully identified, for each proposed function, the conditions, if any, under which argument values give rise to domain errors. However, we have not provided similar specifications for range errors, since we have found that a general statement (*e.g.*, "a range error occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude" [ISO:9899, §7.12.1/3]) will suffice to cover our situations. Such a general statement can be incorporated by reference, and we believe that [Plauger2002] is very likely to do so.

E. Traditional or Extended Error Codes?

Having recommended, in the previous subsection, the continued use of error codes, a further question arises: Are the existing `EDOM`, `ERANGE` error codes sufficient to the needs of the proposed *Special Functions*?

We note that several current implementations of some of the proposed *Special Functions* do define their own codes to supplement the codes mandated by the standard. They incorporate codes for such situations as overflow, underflow, loss of precision, singularities, and the like, and make these codes accessible via additional global variables analogous to `errno`, or via other means.

Nothing in this proposal should be construed as preventing implementors from such optional extensions. However, this proposal recommends against requiring such behavior. We base this recommendation on consistency with current standards. In particular, we again call attention to [ISO:9899, §7.12.1].

F. Traditional or Descriptive Function Names?

In selecting names for the proposed new functions, we were moved, in previous versions of this proposal, to retain their traditional (mathematical) names. For example, we had kept the names of a few functions which are customarily denoted using Greek letters (spelled out, of course:

beta, zeta [but tgamma for compatibility with C99]). We also kept the function names erf and erfc because they are both traditional and descriptive, as well as for compatibility with C99.

In the remaining (majority of) cases, the traditional mathematical names are mostly single letters (*e.g.*, J and N). We judged such names to be too brief and insufficiently descriptive for programming purposes. Also, such one-character names are frequently reserved by coding standards to signify local variables. For these reasons, we recommend against use of the traditional names. Instead, we had recommended such names as bessel_J and neumann_N, combining a descriptive prefix with a traditional suffix.

We noted in passing that this policy resulted in a few pairs of our names that differed only in the case of a single character (*e.g.*, bessel_J and bessel_j). In each instance, this was an artifact of the traditional mathematical naming convention that we preserved on the basis of prior art as the accepted canonical mathematical nomenclature. While some coding standards recommend or require avoidance of multiple identifiers that differ only in case, we believed it appropriate in the present context to embrace case-sensitivity in distinguishing otherwise-identical names.

However, a recent paper [Plauger2003] suggests an alternate naming scheme. We have herein adopted the proposed scheme, and refer the reader to that paper for the rationale. (The other proposals made in [Plauger2003] would change other aspects of the special functions' signatures; by common agreement, those additional signature-related proposals have been withdrawn and hence not incorporated herein.)

G. Real- or Complex-valued Domains and Results?

Many of the proposed *Special Functions* have definitions over some or all of the complex plane as well as over some or all of the real numbers. Further, some of these functions can produce complex results, even over real-valued arguments. The present proposal restricts itself by considering only real-valued arguments and (correspondingly) real-valued results.

Our investigation of the alternative led us to realize that the complex landscape for the *Special Functions* is figuratively dotted with land mines. In coming to our recommendation, we gave weight to the statement from a respected colleague that "Several Ph.D. dissertations would [or could] result from efforts to implement this set of functions over the complex domain." This led us to take the position that there is insufficient prior art in this area to serve as a basis for standardization, and that such standardization would be therefore premature. While we could perhaps consider standardizing some subset of the *Special Functions* over the complex domain, we far prefer to treat this set of *Special Functions* as a unit.

We further ruled out (via domain errors, for example) the possibility that these *Special Functions* could return complex results. In making this recommendation, we followed the past practice of C++ and of C99: functions taking real arguments always return real results; only functions taking complex arguments return complex results. Perhaps the best example of this is the sqrt function: it always returns a value whose type is identical to its parameter's type. To do otherwise opens the door to a number of small, but bothersome technical issues. As one example, which header (<cmath> or <complex>) would declare such a function whose domain and range types are different?

Finally, none of our colleagues or reviewers has presented any compelling need or rationale for the extension to the complex domain or range. While there would certainly be some segments of the user community that could take advantage of such functionality (and we certainly don't mean to prohibit vendors from providing such as extensions), there seems to be insufficient demand to require such at present.

Because we have thus restricted ourselves to functions taking real-valued arguments and producing real-valued results, this proposal will make special provision for three of the *Special Functions* in the ISO standard's list: the cylindrical Hankel functions (also known as the cylindrical Bessel functions of the third kind), the spherical Hankel functions (also known as the spherical Bessel functions of the third kind), and the spherical harmonics functions. Such special treatment is needed because these functions are inherently complex-valued, even for real-valued arguments:

- We have entirely omitted both the cylindrical and the spherical Hankel functions from our list of candidates for standardization. Other functions in the list can be used to obtain the real and imaginary parts of the Hankel functions' results without loss of either precision or performance. Since these can be trivially composed, the Hankel functions' omission poses no significant burden on a user.

- Instead of the spherical harmonics, we opted to provide the closely-related and real-valued spherical associated Legendre functions. We chose to provide these because they can trivially be used to produce the real and imaginary parts of the spherical harmonics, and because a user could not otherwise obtain the spherical harmonics without loss of precision.

H. Which Conventions?

Among the functions in this proposal, there are several for which multiple sign and normalization conventions exist. As examples, we note that [ISO:31-11] and [Abramowitz] differ:

- In sign convention for associated Legendre functions, and
- In normalization convention for Laguerre polynomials.

Because the choices do not easily co-exist, we have opted to resolve any such ambiguities by appealing to a common source, the aforementioned standard *Handbook of Mathematical Functions* [Abramowitz].

I. Relationship to LIA?

It has been suggested that the functions constituting the subject of this proposal be reviewed within the framework of one or more parts of the International Standard for Language Independent Arithmetic [ISO:10967-1, ISO:10967-2, ISO:10967-3]. While we applaud the motivation underlying the suggestion, we recommend against such a perspective, believing it to be premature and not yet feasible.

As of this writing, only LIA Part 1 has been formally adopted as an International Standard: LIA Part 2 is a Final Draft, while LIA Part 3 is only a Committee Draft. Further, LIA Part 1 does not speak to the subject of the present proposal, while Parts 2 and 3 restrict themselves to coverage of "elementary" numerical functions such as those already part of the C++ standard library. Thus, none of the LIA documents addresses (or even mentions) any of the functions comprising the present proposal.

Finally, we note that the C++ standards body has to date not adopted any "conformity statement" regarding the LIA "binding standard" to be used by a conforming C++ implementation. In the absence of such guidance, it is unclear how to apply to C++ the principles (let alone the details) of the LIA documents.

In our view, the present situation is best summarized as constituting a lack of prior art: C++ has not determined how LIA is to apply to an implementation and, in any event, none of the functions comprising the present proposal are in the scope of the LIA documents as currently drafted. For these reasons, we believe there is today an insufficient basis on which to evaluate the present proposal with respect to LIA.

4. Proposed Text

Note: the wording presented in these subsections describes additions to the `<cmath>` and `<math.h>` headers.

A. To be inserted into Table 80 (Clause 26)

assoc_laguerre	conf_hyperg	ellint_2	legendre
assoc_legendre	cyl_bessel_i	ellint_e	riemann_zeta
beta	cyl_bessel_j	expint	sph_bessel
comp_ellint_1	cyl_bessel_k	hermite	sph_legendre
comp_ellint_2	cyl_neumann	hyperg	
comp_ellint_3	ellint_1	laguerre	

B. New section to be added to Clause 26

26.x.1 Synopsis

```
// (26.x.2) cylindrical Bessel functions (of the first kind):
double       cyl_bessel_j( double nu, double x );
float        cyl_bessel_jf( float nu, float  x );
long double  cyl_bessel_jl( long double nu, long double x );

// (26.x.3) cylindrical Neumann functions;
// cylindrical Bessel functions of the second kind:
double       cyl_neumann( double nu, double x );
float        cyl_neumannf( float nu, float  x );
long double  cyl_neumannl( long double nu, long double x );

// (26.x.4.1) regular modified cylindrical Bessel functions:
double       cyl_bessel_i( double nu, double x );
float        cyl_bessel_if( float nu, float  x );
long double  cyl_bessel_il( long double nu, long double x );

// (26.x.4.2) irregular modified cylindrical Bessel functions:
double       cyl_bessel_k( double nu, double x );
float        cyl_bessel_kf( float nu, float  x );
long double  cyl_bessel_kl( long double nu, long double x );

// (26.x.5) spherical Bessel functions (of the first kind):
double       sph_bessel( unsigned n, double x );
float        sph_besself( unsigned n, float  x );
long double  sph_bessell( unsigned n, long double x );

// (26.x.6) spherical Neumann functions;
// spherical Bessel functions of the second kind:
double       sph_neumann( unsigned n, double x );
float        sph_neumannf( unsigned n, float  x );
long double  sph_neumannl( unsigned n, long double x );

// (26.x.7) Legendre polynomials:
double       legendre( unsigned l, double x );
float        legendref( unsigned l, float x );
long double  legendrel( unsigned l, long double x );

// (26.x.8) associated Legendre functions:
double       assoc_legendre( unsigned l, unsigned m, double x );
float        assoc_legendref( unsigned l, unsigned m, float x );
long double  assoc_legendrel( unsigned l, unsigned m, long double x );
```

```
// (26.x.9) spherical associated Legendre functions
double       sph_legendre( unsigned l, unsigned m, double theta );
float        sph_legendref( unsigned l, unsigned m, float theta );
long double  sph_legendrel( unsigned l, unsigned m, long double theta );

// (26.x.10) Hermite polynomials:
double       hermite( unsigned n, double x );
float        hermitef( unsigned n, float x );
long double  hermitel( unsigned n, long double x );

// (26.x.11) Laguerre polynomials:
double       laguerre( unsigned n, double x );
float        laguerref( unsigned n, float x );
long double  laguerrel( unsigned n, long double x );

// (26.x.12) associated Laguerre polynomials:
double       assoc_laguerre( unsigned n, unsigned m, double x );
float        assoc_laguerref( unsigned n, unsigned m, float x );
long double  assoc_laguerrel( unsigned n, unsigned m, long double x );

// (26.x.13) hypergeometric functions:
double       hyperg( double a, double b, double c, double x );
float        hypergf( float a, float b, float c, float x );
long double  hypergl( long double a, long double b, long double c, long double x );

// (26.x.14) confluent hypergeometric functions:
double       conf_hyperg( double a, double c, double x );
float        conf_hypergf( float a, float c, float x );
long double  conf_hypergl( long double a, long double c, long double x );

// (26.x.15.1) (incomplete) elliptic integral of the first kind:
double       ellint_1( double k, double phi );
float        ellint_1f( float k, float phi );
long double  ellint_1l( long double k, long double phi );

// (26.x.15.2) (complete) elliptic integral of the first kind:
double       comp_ellint_1( double k );
float        comp_ellint_1f( float k );
long double  comp_ellint_1l( long double k );

// (26.x.16.1) (incomplete) elliptic integral of the second kind:
double       ellint_2( double k, double phi );
float        ellint_2f( float k, float phi );
long double  ellint_2l( long double k, long double phi );

// (26.x.16.2) (complete) elliptic integral of the second kind:
double       comp_ellint_2( double k );
float        comp_ellint_2f( float k );
long double  comp_ellint_2l( long double k );

// (26.x.17.1) (incomplete) elliptic integral of the third kind:
double       ellint_e( double k, double n, double phi );
float        ellint_ef( float k, float n, float phi );
long double  ellint_el( long double k, long double n, long double phi );
```

```
// (26.x.17.2) (complete) elliptic integral of the third kind:
double       comp_ellint_3( double k, double n );
float        comp_ellint_3f( float k, float n );
long double  comp_ellint_3l( long double k, long double n );

// (26.x.19) beta function:
double       beta( double x, double y );
float        betaf( float x, float y );
long double  betal( long double x, long double y );

// (26.x.20) exponential integral:
double       expint( double );
float        expintf( float );
long double  expintl( long double );

// (26.x.22) Riemann zeta function:
double       riemann_zeta( double );
float        riemann_zetaf( float );
long double  riemann_zetal( long double );
```

26.x.2 cylindrical Bessel functions (of the first kind)

```
double       cyl_bessel_j( double nu, double x );
float        cyl_bessel_jf( float nu, float  x );
long double  cyl_bessel_jl( long double nu, long double x );
```

1. Effects: The `cyl_bessel_j` functions compute the cylindrical Bessel functions of the first kind of their respective arguments `nu` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `cyl_bessel_j` functions return

$$\mathsf{J}_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)} \,.$$

26.x.3 cylindrical Neumann functions; cylindrical Bessel functions of the second kind

```
double       cyl_neumann( double nu, double x );
float        cyl_neumannf( float nu, float  x );
long double  cyl_neumannl( long double nu, long double x );
```

1. Effects: The `cyl_neumann` functions compute the cylindrical Neumann functions, also known as the cylindrical Bessel functions of the second kind, of their respective arguments `nu` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `cyl_neumann` functions return

$$\mathsf{N}_\nu(x) = \begin{cases} \dfrac{\mathsf{J}_\nu(x)\cos\nu\pi - \mathsf{J}_{-\nu}(x)}{\sin\nu\pi} & \text{for non-integral } \nu \\[2ex] \lim\limits_{\mu\to\nu} \dfrac{\mathsf{J}_\mu(x)\cos\mu\pi - \mathsf{J}_{-\mu}(x)}{\sin\mu\pi} & \text{for integral } \nu \end{cases} \,.$$

26.x.4.1 regular modified cylindrical Bessel functions

```
double       cyl_bessel_i( double nu, double x );
float        cyl_bessel_if( float nu, float  x );
long double  cyl_bessel_il( long double nu, long double x );
```

1. Effects: The `cyl_bessel_i` functions compute the regular modified cylindrical Bessel functions of their respective arguments `nu` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `cyl_bessel_i` functions return

$$\mathsf{I}_\nu(x) = \mathrm{i}^{-\nu}\mathsf{J}_\nu(\mathrm{i}x) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)} \ .$$

26.x.4.2 irregular modified cylindrical Bessel functions

```
double       cyl_bessel_k( double nu, double x );
float        cyl_bessel_kf( float nu, float  x );
long double  cyl_bessel_kl( long double nu, long double x );
```

1. Effects: The `cyl_bessel_k` functions compute the irregular modified cylindrical Bessel functions of their respective arguments `nu` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `cyl_bessel_k` functions return

$$\mathsf{K}_\nu(x) = (\pi/2)\mathrm{i}^{\nu+1}(\mathsf{J}_\nu(\mathrm{i}x) + \mathrm{i}\mathsf{N}_\nu(\mathrm{i}x)) = \begin{cases} \dfrac{\pi}{2}\dfrac{\mathsf{I}_{-\nu}(x) - \mathsf{I}_\nu(x)}{\sin\nu\pi} & \text{for non-integral } \nu \\[2ex] \dfrac{\pi}{2}\lim\limits_{\mu\to\nu}\dfrac{\mathsf{I}_{-\mu}(x) - \mathsf{I}_\mu(x)}{\sin\mu\pi} & \text{for integral } \nu \end{cases} \ .$$

26.x.5 spherical Bessel functions (of the first kind)

```
double       sph_bessel( unsigned n, double x );
float        sph_besself( unsigned n, float  x );
long double  sph_bessell( unsigned n, long double x );
```

1. Effects: The `sph_bessel` functions compute the spherical Bessel functions of the first kind of their respective arguments `n` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `sph_bessel` functions return

$$\mathsf{j}_n(x) = (\pi/2x)^{1/2}\mathsf{J}_{n+1/2}(x) \ .$$

26.x.6 spherical Neumann functions; spherical Bessel functions of the second kind

```
double       sph_neumann( unsigned n, double x );
float        sph_neumannf( unsigned n, float  x );
long double  sph_neumannl( unsigned n, long double x );
```

1. Effects: The `sph_neumann` functions compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments `n` and `x`. A domain error may occur if `x` is less than zero.

2. Returns: The `sph_neumann` functions return

$$\mathsf{n}_n(x) = (\pi/2x)^{1/2} \mathsf{N}_{n+1/2}(x) \ .$$

26.x.7 Legendre polynomials

```
double       legendre( unsigned l, double x );
float        legendref( unsigned l, float x );
long double  legendrel( unsigned l, long double x );
```

1. Effects: The `legendre` functions compute the Legendre polynomials of their respective arguments `l` and `x`. A domain error may occur if the magnitude of `x` is greater than one.

2. Returns: The `legendre` functions return

$$\mathsf{P}_l(x) = \frac{1}{2^l \, l!} \frac{\mathsf{d}^l}{\mathsf{d}x^l} (x^2 - 1)^l \ .$$

26.x.8 associated Legendre functions

```
double       assoc_legendre( unsigned l, unsigned m, double x );
float        assoc_legendref( unsigned l, unsigned m, float x );
long double  assoc_legendrel( unsigned l, unsigned m, long double x );
```

1. Effects: The `assoc_legendre` functions compute the associated Legendre functions of their respective arguments `l`, `m`, and `x`. A domain error occurs if `m` is greater than `l`. A domain error may occur if if the magnitude of `x` is greater than one.

2. Returns: The `assoc_legendre` functions return

$$\mathsf{P}_l^m(x) = (1 - x^2)^{m/2} \, \frac{\mathsf{d}^m}{\mathsf{d}x^m} \, \mathsf{P}_l(x) \ .$$

26.x.9 spherical associated Legendre functions

```
double       sph_legendre( unsigned l, unsigned m, double theta );
float        sph_legendref( unsigned l, unsigned m, float theta );
long double  sph_legendrel( unsigned l, unsigned m, long double theta );
```

1. Effects: The `sph_legendre` functions compute spherical associated Legendre functions of their respective arguments `l`, `m`, and `theta`. A domain error occurs if `m` is greater than `l`.

2. Returns: The `sph_legendre` functions return

$$(-1)^m \left[\frac{(2l+1)}{4\pi} \frac{(l-m)!}{(l+m)!} \right]^{1/2} \mathsf{P}_l^m(\cos\theta) \ .$$

26.x.10 Hermite polynomials

```
double       hermite( unsigned n, double x );
float        hermitef( unsigned n, float x );
long double  hermitel( unsigned n, long double x );
```

1. Effects: The `hermite` functions compute the Hermite polynomials of their respective arguments `n` and `x`.

2. Returns: The `hermite` functions return

$$\mathsf{H}_n(x) = (-1)^n e^{x^2} \frac{\mathsf{d}^n}{\mathsf{d}x^n} e^{-x^2} .$$

26.x.11 Laguerre polynomials

```
double       laguerre( unsigned n, double x );
float        laguerref( unsigned n, float x );
long double  laguerrel( unsigned n, long double x );
```

1. Effects: The `laguerre` functions compute the Laguerre polynomials of their respective arguments `n` and `x`.

2. Returns: The `laguerre` functions return

$$\mathsf{L}_n(x) = e^x \frac{\mathsf{d}^n}{\mathsf{d}x^n} \left(x^n e^{-x}\right) .$$

26.x.12 associated Laguerre polynomials

```
double       assoc_laguerre( unsigned n, unsigned m, double x );
float        assoc_laguerref( unsigned n, unsigned m, float x );
long double  assoc_laguerrel( unsigned n, unsigned m, long double x );
```

1. Effects: The `assoc_laguerre` functions compute the associated Laguerre polynomials of their respective arguments `n`, `m`, and `x`.

2. Returns: The `assoc_laguerre` functions return

$$\mathsf{L}_n^m(x) = e^x \frac{\mathsf{d}^m}{\mathsf{d}x^m} \mathsf{L}_n(x) .$$

26.x.13 hypergeometric functions

```
double       hyperg( double a, double b, double c, double x );
float        hypergf( float a, float b, float c, float x );
long double  hypergl( long double a, long double b, long double c, long double x );
```

1. Effects: The `hyperg` compute the hypergeometric functions of their respective arguments `a`, `b`, `c`, and `x`. A domain error may occur if `x` is greater than or equal to one.

2. Returns: The `hyperg` functions return

$$\mathsf{F}(a,b;c;x) = \frac{\Gamma(c)}{\Gamma(a)\Gamma(b)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)\,\Gamma(b+n)}{\Gamma(c+n)} \frac{x^n}{n!} .$$

26.x.14 confluent hypergeometric functions

```
double       conf_hyperg( double a, double c, double x );
float        conf_hypergf( float a, float c, float x );
long double  conf_hypergl( long double a, long double c, long double x );
```

1. Effects: The `conf_hyperg` functions compute the confluent hypergeometric functions of their respective arguments `a`, `c`, and `x`. A domain error occurs (a) if `c` is a negative integer, or (b) if `c` is zero.

2. Returns: The `conf_hyperg` functions return

$$\mathsf{F}(a; c; x) = \frac{\Gamma(c)}{\Gamma(a)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)}{\Gamma(c+n)} \frac{x^n}{n!} .$$

26.x.15.1 (incomplete) elliptic integral of the first kind

```
double       ellint_1( double k, double phi );
float        ellint_1f( float k, float phi );
long double  ellint_1l( long double k, long double phi );
```

1. Effects: The `ellint_1` functions compute the incomplete elliptic integral of the first kind of their respective arguments `k` and `phi`. A domain error may occur if the magnitude of `k` is greater than one.

2. Returns: The `ellint_1` functions return

$$\mathsf{F}(k, \phi) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2 sin^2\theta}} .$$

26.x.15.2 (complete) elliptic integral of the first kind

```
double       comp_ellint_1( double k );
float        comp_ellint_1f( float k );
long double  comp_ellint_1l( long double k );
```

1. Effects: The `comp_ellint_1` functions compute the complete elliptic integral of the first kind of their respective arguments `k`. A domain error occurs if the magnitude of `k` is greater than one.

2. Returns: The `comp_ellint_1` functions return

$$\mathsf{K}(k) = \int_0^{\pi/2} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2 sin^2\theta}} .$$

26.x.16.1 (incomplete) elliptic integral of the second kind

```
double       ellint_2( double k, double phi );
float        ellint_2f( float k, float phi );
long double  ellint_2l( long double k, long double phi );
```

1. Effects: The `ellint_2` functions compute the incomplete elliptic integral of the second kind of their respective arguments `k` and `phi`. A domain error may occur if the magnitude of `k` is greater than one.

2. Returns: The `ellint_2` functions return

$$\mathsf{E}(k, \phi) = \int_0^{\phi} \sqrt{1 - k^2 \sin^2\theta}\, \mathrm{d}\theta .$$

26.x.16.2 (complete) elliptic integral of the second kind

```
double       comp_ellint_2( double k );
float        comp_ellint_2f( float k );
long double  comp_ellint_2l( long double k );
```

1. Effects: The `comp_ellint_2` functions compute the complete elliptic integral of the second kind of their respective arguments `k`. A domain error occurs if the magnitude of `k` is greater than one.

2. Returns: The `comp_ellint_2` functions return

$$\mathsf{E}(k, \pi/2) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 \theta}\, \mathsf{d}\theta \ .$$

26.x.17.1 (incomplete) elliptic integral of the third kind

```
double       ellint_e( double k, double n, double phi );
float        ellint_ef( float k, float n, float phi );
long double  ellint_el( long double k, long double n, long double phi );
```

1. Effects: The `ellint_e` functions compute the incomplete elliptic integral of the third kind of their respective arguments `k`, `n`, and `phi`. A domain error may occur if the magnitude of `k` is greater than one.

2. Returns: The `ellint_e` functions return

$$\mathsf{\Pi}(n, k, \phi) = \int_0^{\phi} \frac{\mathsf{d}\theta}{(1 - n\, sin^2\theta)\sqrt{1 - k^2 sin^2\theta}} \ .$$

26.x.17.2 (complete) elliptic integral of the third kind

```
double       comp_ellint_3( double k, double n );
float        comp_ellint_3f( float k, float n );
long double  comp_ellint_3l( long double k, long double n );
```

1. Effects: The `comp_ellint_3` functions compute the complete elliptic integral of the third kind of their respective arguments `k` and `n`. A domain error occurs if the magnitude of `k` is greater than one.

2. Returns: The `comp_ellint_3` functions return

$$\mathsf{\Pi}(n, k, \pi/2) = \int_0^{\pi/2} \frac{\mathsf{d}\theta}{(1 - n\, sin^2\theta)\sqrt{1 - k^2 sin^2\theta}} \ .$$

paragraph**26.x.19 beta function**

```
double       beta( double x, double y );
float        betaf( float x, float y );
long double  betal( long double x, long double y );
```

1. Effects: The `beta` functions compute the beta function of their respective arguments `x` and `y`. A domain error may occur (a) if either `x` or `y` is a negative integer, or (b) if either `x` or `y` is zero.

2. Returns: The `beta` functions return

$$\mathsf{B}(x,y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)}\ .$$

26.x.20 exponential integral

```
double       expint( double x );
float        expintf( float x );
long double  expintl( long double x );
```

1. Effects: The `expint` functions compute the exponential integral of their respective arguments `x`.

2. Returns: The `expint` functions return

$$\mathsf{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,\mathsf{d}t\ .$$

26.x.22 Riemann zeta function

```
double       riemann_zeta( double x );
float        riemann_zetaf( float x );
long double  riemann_zetal( long double x );
```

1. Effects: The `riemann_zeta` functions compute the Riemann zeta function of their respective arguments `x`. A domain error occurs if `x` is equal to one.

2. Returns: The `riemann_zeta` functions return

$$\zeta(x) = \begin{cases} \displaystyle\sum_{k=1}^{\infty} k^{-x} & \text{for } x > 1 \\[2ex] 2^x \pi^{x-1} \sin(\frac{\pi x}{2})\,\Gamma(1-x)\,\zeta(1-x) & \text{for } x < 1 \end{cases}\ .$$

26.x.23 Additional versions

Each of the functions declared above that has one or more `double` parameters (the `double` version) shall have two additional overloads:

1. A version with each `double` parameter replaced with a `float` parameter (the `float` version), and
2. A version with each `double` parameter replaced with a `long double` parameter (the `long double` version).

The return type of each such `float` version shall be `float`, and the return type of each such `long double` version shall be `long double`.

Moreover, each `double` version shall have sufficient additional overloads to determine which of the above three versions to actually call, by the following ordered set of rules:

- First, if any argument corresponding to a `double` parameter in the `double` version has type `long double`, the `long double` version is called.

- Otherwise, if any argument corresponding to a `double` parameter in the `double` version has type `double` or has an integer type, the `double` version is called.
- Otherwise, the `float` version is called.

5. Relationship to Earlier Proposal

We noted in §2. that there is a small potential overlap between this proposal (which is based on [ISO:31-11]) and an earlier proposal (which is based on [ISO:9899]. However, no competition is intended between the two proposals. Indeed, we have coordinated efforts to ensure that no incompatibilities result.

In particular, this section describes three functions (`erf`, `erfc`, and `tgamma`) originating in the C99 library and previously proposed for C++ standardization as part of [Plauger2002]. They are mentioned in the present proposal for two reasons, however: (1) these functions are traditionally characterized as mathematical *Special Functions*, and (2) they form part of [ISO:31-11], on which our proposal is based.

We emphasize that these functions are here described solely in the interest of completeness so that the full spectrum of envisioned mathematical *Special Functions* may be viewed together. For purposes of the C++ standardization effort, these functions should be considered part of the [Plauger2002] proposal. The language in the remainder of this section should therefore be considered solely for informative purposes.

A. To be inserted into Table 80 (Clause 26)

```
erf
erfc
tgamma
```

B. New section to be added to Clause 26

26.x.1 Synopsis

```
// (26.x.18) gamma function:
double       tgamma( double );
float        tgammaf( float );
long double  tgammal( long double );

// (26.x.21.1) error function:
double       erf( double );
float        erff( float );
long double  erfl( long double );

// (26.x.21.2) complementary error function:
double       erfc( double );
float        erfcf( float );
long double  erfcl( long double );
```

26.x.18 gamma function

```
double       tgamma( double x );
float        tgammaf( float x );
long double  tgammal( long double x );
```

1. Effects: The `tgamma` functions compute the gamma function of their respective arguments `x`. A domain error occurs (a) if `x` is a negative integer, or (b) if `x` is zero.

2. Returns: The `tgamma` functions return

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t \, .$$

26.x.21.1 error function

```
double       erf( double x );
float        erff( float x );
long double  erfl( long double x );
```

1. Effects: The `erf` functions compute the error function of their respective arguments `x`.

2. Returns: The `erf` functions return

$$\mathrm{erf}\ x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t \, .$$

26.x.21.2 complementary error function

```
double       erfc( double x );
float        erfcf( float x );
long double  erfcl( long double x );
```

1. Effects: The `erfc` functions compute the complementary error function of their respective arguments `x`.

2. Returns: The `erfc` functions return

$$\mathrm{erfc}\ x = 1 - \mathrm{erf}\ x = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} \mathrm{d}t \, .$$

6. Acknowledgments

It is a pleasure to acknowledge the significant contributions provided by a number of colleagues at Fermilab: James Amundson, Mark Fischler, Jeffrey Kallenbach, Leo Michelotti, and Marc Paterno. Their active participation has materially improved this proposal, and their inspiration and support are deeply appreciated.

We extend special thanks to our Fermilab colleague John Marraffino for his extensive involvement with all aspects of this proposal's development and refinement. We are likewise especially grateful to Marc Paterno for his significant contributions to version 2.

We have also received valuable input and feedback from Matt Austern, Beman Dawes, Gabriel Dos Reis, J. Michael Gibbs, P. J. Plauger, and Fred J. Tydeman. We are grateful to them and to our outside reviewers for their careful consideration of earlier versions of this document: Their thoughtful comments and suggestions inspired several refinements and enhancements to this proposal.

Finally, we appreciate the support of the Fermi National Accelerator Laboratory's Computing Division, sponsors of our participation in the C++ standards effort. Thank you, one and all.

7. References

[Abramowitz] Abramowitz, Milton, and Irene A. Stegun (eds.): *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*, volume 55 of National Bureau of Standards Applied Mathematics Series. U. S. Government Printing Office, Washington, DC: 1964. Reprinted with corrections, Dover Publications: 1972. `http://members.fortunecity.com/aands`.

[Austern] Austern, Matt: *Notes on Standard Library Extensions*. WG21/N1314 (same as J16/01-0028): 17 May 2001.

[Dawes] Dawes, Beman: *Library Technical Report Proposals and Issues List (Revision 4)*. WG21/N1397 (same as J16/02-0055): 2002.

[Galassi] Galassi, Mark, *et al.*: *The GNU Scientific Library*. 2002. `http://www.gnu.org/software/gsl/gsl.html`.

[GNU C] GNU C Library Steering Committee *The GNU C Library*. `http://www.gnu.org/manual/glibc-2.0.6/html_node/libc_toc.html`.

[Hildebrand] Hildebrand, Francis B.: *Advanced Calculus for Applications, Second Edition*. Prentice-Hall, Inc., 1976. ISBN 0-13-011189-9.

[IMSL] Visual Numerics, Inc.: *IMSL C Numerical Library Version 5.0*. Unpublished. `http://www.vni.com/products/imsl/docs/cmath.pdf`.

[ISO:31-11] International Standards Organization: *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology*. International Standard ISO 31-11:1992. In *Quantities and units, Third edition*. ISBN 92-67-10185-4.

[ISO:9899] International Standards Organization: *Programming Languages — C, Second edition*. International Standard ISO/IEC 9899:1999.

[ISO:10967-1] International Standards Organization: *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*. International Standard ISO/IEC 10967-1:1994.

[ISO:10967-2] International Standards Organization: *Information technology — Language independent arithmetic — Part 2: Elementary numerical functions*. Draft International Standard ISO/IEC FDIS 10967-2:2000.

[ISO:10967-3] International Standards Organization: *Information technology — Language independent arithmetic — Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions*. Draft International Standard ISO/IEC CD 10967-3.1:2002.

[ISO:14882] International Standards Organization: *Programming Languages — C++* . International Standard ISO/IEC 14882:1998.

[Jackson] Jackson, John David: *Classical Electrodynamics, Second Edition*. John Wiley & Sons, 1975. ISBN 0-471-43132-X.

[Josuttis] Josuttis, Nicolai: *A New Work Item Proposal: Technical Report for Library Issues*. WG21/N1283 (same as J16/00-0060): 26 October 2000.

[Lebedev] Lebedev, N. N., and Richard A. Silverman (trans.): *Special Functions & Their Applications, Revised English Edition*. Dover Publications, Inc., 1972. ISBN 0-486-60624-4.

[NAG] The Numerical Algorithms Group: *The NAG C Library Manual, Mark 7*. The Numerical Algorithms Group, Ltd., Oxford, UK: 2002. `http://www.nag.com/numeric/CL/manual/html/CLlibrarymanual.asp`.

[Plauger2002] Plauger, P. J.: *Proposed C99 Library Additions to C++ (Revised)*. WG21/N1372 (same as J16/02-0030): 2002. `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2002/n1372.txt`.

[Plauger2003] Plauger, P. J.: *Proposed Signature Changes for Special Math Functions in TR-1*. WG21/N1502 (same as J16/03-0085): 2003 `http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1502.txt`.

[SGI] Silicon Graphics, Inc.: *Introduction to mathematical library functions*. 2002. `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/p_man/cat3/standard/math.z&srch=math`.

[SLATEC] Fong, Kirby W., *et al.*: *Guide to the SLATEC Common Mathematical Library*. July 1993. `http://www.netlib.org/slatec/guide`.

[Spanier] Spanier, Jerome and Keith B. Oldham: *An Atlas of Functions*. Hemisphere Publishing Corp., 1987. ISBN 0-89116-573-8.

[Whittaker] Whittaker, E. T. and G. N. Watson: *A Course of Modern Analysis, Fourth Edition Reprinted*. Cambridge University Press, 1958.

Summary of changes in version 2

In addition to numerous minor editorial and stylistic improvements, changes due to the following actions distinguish the present version 2 of this proposal from its predecessor document, WG21/N1422 = J16/03-0004:

- Add discussion re error handling in §3.D.
- Add discussion re function naming in §3.F.
- Consolidate function references via new table in §3.B.
- Improve normative text describing conditions leading to domain errors.
- Adjust `bessel_j` and `neumann_n` function signatures to correspond better with current practice.
- Replace `sph_Y` function with `sph_legendre_Plm`, and add rationale for this in §3.G.
- Rename `zeta` function as `riemann_zeta`.
- Remove from normative text all mention of range errors, and add rationale for this in §3.D.
- Number the **Effects** and **Returns** paragraphs in §4.B. and in §5.B.
- Extend acknowledgments in §6.
- Cite additional sources, and provide the new references in §7.

Summary of changes in version 3

In addition to a few minor editorial and stylistic improvements, changes due to the following actions distinguish the present version 3 of this proposal from its predecessor document, WG21/N1514 = J16/03-0097:

- Rename the *Special Functions* (per [Plauger2003]) throughout normative text in §4., and update Table 1, §3.F., and §7. accordingly
- Specify all `*f` and `*l` versions for the *Special Functions* in §4. and §5.
- Add new **26.x.23 Additional versions** paragraphs