Math Toolkit 2.10.0

Nikhar Agrawal
Anton Bikineev
Paul A. Bristow
Hubert Holin
Marco Guazzone
Christopher Kormanyos
Bruno Lalande
John Maddock
Jeremy W. Murphy
Matthew Pulver
Johan Råde
Benjamin Sobotta
Gautam Sewani
Nicholas Thompson
Thijs van den Berg
Daryle Walker
Xiaogang Zhang

Math Toolkit 2.10.0

by Nikhar Agrawal, Anton Bikineev, Paul A. Bristow, Hubert Holin, Marco Guazzone, Christopher Kormanyos, Bruno Lalande, John Maddock, Jeremy W. Murphy, Matthew Pulver, Johan Råde, Benjamin Sobotta, Gautam Sewani, Nicholas Thompson, Thijs van den Berg, Daryle Walker, and Xiaogang Zhang

This manual is also available in printer friendly PDF format, and as a CD ISBN 0-9504833-2-X 978-0-9504833-2-0, Classification 519.2-dc22.

Table of Contents

List of Tables

Overview

About the Math Toolkit

This library is divided into several interconnected parts:

Floating Point Utilities

Utility functions for dealing with floating-point arithmetic, includes functions for floating point classification (fpclassify, isnan, isinf etc), sign manipulation, rounding, comparison, and computing the distance between floating point numbers.

Specific Width Floating-Point Types

A set of typedefs similar to those provided by `<cstdint>` but for floating-point types.

Mathematical Constants

A wide range of constants ranging from various multiples of π, fractions, through to euler's constant etc.

These are of course usable from template code, or as non-templates with a simplified interface if that is more appropriate.

Statistical Distributions

Provides a reasonably comprehensive set of statistical distributions, upon which higher level statistical tests can be built.

The initial focus is on the central univariate distributions. Both continuous (like normal & Fisher) and discrete (like binomial & Poisson) distributions are provided.

A comprehensive tutorial is provided, along with a series of worked examples illustrating how the library is used to conduct statistical tests.

Mathematical Special Functions

Provides a small number of high quality special functions, initially these were concentrated on functions used in statistical applications along with those in the Technical Report on C++ Library Extensions.

The function families currently implemented are the gamma, beta & erf functions along with the incomplete gamma and beta functions (four variants of each) and all the possible inverses of these, plus digamma, various factorial functions, Bessel functions, elliptic integrals, sinus cardinals (along with their hyperbolic variants), inverse hyperbolic functions, Legrendre/Laguerre/Hermite polynomials and various special power and logarithmic functions.

All the implementations are fully generic and support the use of arbitrary "real-number" types, including Boost.Multiprecision, although they are optimised for use with types with known-about significand (or mantissa) sizes: typically `float`, `double` or `long double`.

These functions also provide the basis of support for the TR1 special functions.

Root Finding and Function Minimisation

A comprehensive set of root finding algorithms over the real-line, both with and without derivative support.

Also function minimisation via Brent's Method.

Polynomials and Rational Functions

Tools for manipulating polynomials and for efficient evaluation of rationals or polynomials.

Interpolation

Function interpolation via Barycentric or cubic B_spline approximations.

Numerical Integration and Differentiation

A reasonably comprehensive set of routines for integration (trapezoidal, Gauss-Legendre, Gauss-Kronrod and double-exponential) and differentiation.

The integration routines are all usable for functions returning complex results - and as a result for contour integrals as well.

Quaternions and Octonions

Quaternion and Octonians as class templates similar to `std::complex`.

Navigation

Boost.Math documentation is provided in both HTML and PDF formats.

- **Tutorials** are listed in the *Table of Contents* and include many examples that should help you get started quickly.
- **Source code** of the many *Examples* will often be your quickest starting point.
- **Index** (general) includes all entries.
- **Specific Indexes** list only functions, class signatures, macros and typedefs.

Using the Indexes

The main index will usually help, especially if you know a word describing what it does, without needing to know the exact name chosen for the function.

Tip

When using the index, keep in mind that clicking on an index term only takes you to the **section** containing the index entry. This section may be several pages long, so you may need to use the **find** facility of your browser or PDF reader to get to the index term itself.

Tip

A PDF reader usually allows a **global** find; this can be really useful if the term you expect to be indexed is not the one chosen by the authors. You might find searching the PDF version and viewing the HTML version will locate an elusive item.

Document Conventions

This documentation aims to use of the following naming and formatting conventions.

- C++ Code is in `fixed width font` and is syntax-highlighted in color, for example `double`.
- Other code is in block `teletype fixed-width font`.
- Replaceable text that **you** will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in *`code font`* and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense (like template parameter names) are specified in CamelCase.

Other Hints and tips

- If you have a feature request, or if it appears that the implementation is in error, please search first in the Boost Trac.
- Trac entries may indicate that updates or corrections that solve your problem are in Boost-trunk where changes are being assembled and tested ready for the next release. You may, at your own risk, download new versions from there.
- If you do not understand why things work the way they do, see the *rationale* section.
- If you do not find your idea/feature/complaint, please reach the author preferably through the Boost development list, or email the author(s) direct.

Admonishments

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Tip

These blocks contain information that you may find helpful while coding.

Important

These contain information that is imperative to understanding a concept. Failure to follow suggestions in these blocks will probably result in undesired behavior. Read all of these you find.

Warning

Failure to heed this will lead to incorrect, and very likely undesired, results.

Directory and File Structure

boost/math

/concepts/	Prototype defining the **essential** features of a RealType class (see real_concept.hpp). Most applications will use `double` as the RealType (and short `typedef` names of distributions are reserved for this type where possible), a few will use `float` or `long double`, but it is also possible to use higher precision types like NTL::RR, GNU Multiple Precision Arithmetic Library, GNU MPFR library that conform to the requirements specified by real_concept.
/constants/	Templated definition of some highly accurate math constants (in constants.hpp).
/distributions/	Distributions used in mathematics and, especially, statistics: Gaussian, Students-t, Fisher, Binomial etc
/policies/	Policy framework, for handling user requested behaviour modifications.
/special_functions/	Math functions generally regarded as 'special', like beta, cbrt, erf, gamma, lgamma, tgamma ... (Some of these are specified in C++, and C99/TR1, and perhaps TR2).
/tools/	Tools used by functions, like evaluating polynomials, continued fractions, root finding, precision and limits, and by tests. Some will find application outside this package.

boost/libs

/doc/	Documentation source files in Quickbook format processed into html and pdf formats.
/examples/	Examples and demos of using math functions and distributions.
/performance/	Performance testing and tuning program.
/test/	Test files, in many .cpp files, most using Boost.Test (some with test data as .ipp files, usually generated using NTL RR type with ample precision for the type, often for precisions suitable for up to 256-bit significand real types).
/tools/	Programs used to generate test data. Also changes to the NTL released package to provide a few additional (and vital) extra features.

Namespaces

All math functions and distributions are in `namespace boost::math`

So, for example, the Students-t distribution template in `namespace boost::math` is

```
template <class RealType> class students_t_distribution
```

and can be instantiated with the help of the reserved name `students_t`(for `RealType double`)

```
typedef students_t_distribution<double> students_t;

student_t mydist(10);
```

Warning

Some distribution names are also used in std random library, so to avoid the risk of ambiguity it is better to make explicit using declarations, for example: `using boost::math::students_t_distribution`

Functions not intended for use by applications are in `boost::math::detail`.

Functions that may have more general use, like `digits` (significand), `max_value`, `min_value` and `epsilon` are in `boost::math::tools`.

Policy and configuration information is in namespace `boost::math::policies`.

Tip

Many code snippets assume implicit namespace(s), for example, `std::` or `boost::math`.

Calculation of the Type of the Result

The functions in this library are all overloaded to accept mixed floating point (or mixed integer and floating point type) arguments. So for example:

```
foo(1.0, 2.0);
foo(1.0f, 2);
foo(1.0, 2L);
```

etc, are all valid calls, as long as "foo" is a function taking two floating-point arguments. But that leaves the question:

> *"Given a special function with N arguments of types T1, T2, T3 ... TN, then what type is the result?"*

If all the arguments are of the same (floating point) type then the result is the same type as the arguments.

Otherwise, the type of the result is computed using the following logic:

1. Any arguments that are not template arguments are disregarded from further analysis.
2. For each type in the argument list, if that type is an integer type then it is treated as if it were of type `double` for the purposes of further analysis.
3. If any of the arguments is a user-defined class type, then the result type is the first such class type that is constructible from all of the other argument types.
4. If any of the arguments is of type `long double`, then the result is of type `long double`.
5. If any of the arguments is of type `double`, then the result is of type `double`.
6. Otherwise the result is of type `float`.

For example:

```
cyl_bessel(2, 3.0);
```

Returns a `double` result, as does:

```
cyl_bessel(2, 3.0f);
```

as in this case the integer first argument is treated as a `double` and takes precedence over the `float` second argument. To get a `float` result we would need all the arguments to be of type float:

```
cyl_bessel_j(2.0f, 3.0f);
```

When one or more of the arguments is not a template argument then it doesn't effect the return type at all, for example:

```
sph_bessel(2, 3.0f);
```

returns a `float`, since the first argument is not a template argument and so doesn't effect the result: without this rule functions that take explicitly integer arguments could never return `float`.

And for user-defined types, all of the following return an `NTL::RR` result:

```
cyl_bessel_j(0, NTL::RR(2));

cyl_bessel_j(NTL::RR(2), 3);

cyl_bessel_j(NTL::quad_float(2), NTL::RR(3));
```

In the last case, `quad_float` is convertible to `RR`, but not vice-versa, so the result will be an `NTL::RR`. Note that this assumes that you are using a patched NTL library.

These rules are chosen to be compatible with the behaviour of *ISO/IEC 9899:1999 Programming languages - C* and with the Draft Technical Report on C++ Library Extensions, 2005-06-24, section 5.2.1, paragraph 5.

Error Handling

Quick Reference

Handling of errors by this library is split into two orthogonal parts:

- What kind of error has been raised?
- What should be done when the error is raised?

Warning

The default error actions are to throw an exception with an informative error message. If you do not try to catch the exception, you will not see the message!

The kinds of errors that can be raised are:

Domain Error	Occurs when one or more arguments to a function are out of range.
Pole Error	Occurs when the particular arguments cause the function to be evaluated at a pole with no well defined residual value. For example if tgamma is evaluated at exactly -2, the function approaches different limiting values depending upon whether you approach from just above or just below -2. Hence the function has no well defined value at this point and a Pole Error will be raised.
Overflow Error	Occurs when the result is either infinite, or too large to represent in the numeric type being returned by the function.
Underflow Error	Occurs when the result is not zero, but is too small to be represented by any other value in the type being returned by the function.
Denormalisation Error	Occurs when the returned result would be a denormalised value.
Rounding Error	Occurs when the argument to one of the rounding functions trunc, round and modf can not be represented as an integer type, is outside the range of the result type.
Evaluation Error	Occurs if no method of evaluation is known, or when an internal error occurred that prevented the result from being evaluated: this should never occur, but if it does, then it's likely to be due to an iterative method not converging fast enough.
Indeterminate Result Error	Occurs when the result of a function is not defined for the values that were passed to it.

The action undertaken by each error condition is determined by the current Policy in effect. This can be changed program-wide by setting some configuration macros, or at namespace scope, or at the call site (by specifying a specific policy in the function call).

The available actions are:

throw_on_error	Throws the exception most appropriate to the error condition.
errno_on_error	Sets ::errno to an appropriate value, and then returns the most appropriate result
ignore_error	Ignores the error and simply the returns the most appropriate result.
user_error	Calls a user-supplied error handler.

The following tables show all the permutations of errors and actions, with the **default action for each error shown in bold**:

Table 1. Possible Actions for Domain Errors

Action	Behaviour
throw_on_error	**Throws `std::domain_error`**
errno_on_error	Sets `::errno` to `EDOM` and returns `std::numeric_limits<T>::quiet_NaN()`
ignore_error	Returns `std::numeric_limits<T>::quiet_NaN()`
user_error	Returns the result of `boost::math::policies::user_domain_error`: this function must be defined by the user.

Table 2. Possible Actions for Pole Errors

Action	Behaviour
throw_on_error	**Throws `std::domain_error`**
errno_on_error	Sets `::errno` to `EDOM` and returns `std::numeric_limits<T>::quiet_NaN()`
ignore_error	Returns `std::numeric_limits<T>::quiet_NaN()`
user_error	Returns the result of `boost::math::policies::user_pole_error`: this function must be defined by the user.

Table 3. Possible Actions for Overflow Errors

Action	Behaviour
throw_on_error	**Throws `std::overflow_error`**
errno_on_error	Sets `::errno` to `ERANGE` and returns `std::numeric_limits<T>::infinity()`
ignore_error	Returns `std::numeric_limits<T>::infinity()`
user_error	Returns the result of `boost::math::policies::user_overflow_error`: this function must be defined by the user.

Table 4. Possible Actions for Underflow Errors

Action	Behaviour
throw_on_error	Throws `std::underflow_error`
errno_on_error	Sets `::errno` to `ERANGE` and returns 0.
ignore_error	**Returns 0**
user_error	Returns the result of `boost::math::policies::user_underflow_error`: this function must be defined by the user.

Table 5. Possible Actions for Denorm Errors

Action	Behaviour
throw_on_error	Throws `std::underflow_error`
errno_on_error	Sets `::errno` to `ERANGE` and returns the denormalised value.
ignore_error	**Returns the denormalised value.**
user_error	Returns the result of `boost::math::policies::user_denorm_error`: this function must be defined by the user.

Table 6. Possible Actions for Rounding Errors

Action	Behaviour
throw_on_error	Throws `boost::math::rounding_error`
errno_on_error	Sets `::errno` to `ERANGE` and returns the largest representable value of the target integer type (or the most negative value if the argument to the function was less than zero).
ignore_error	**Returns the largest representable value of the target integer type (or the most negative value if the argument to the function was less than zero).**
user_error	Returns the result of `boost::math::policies::user_rounding_error`: this function must be defined by the user.

Table 7. Possible Actions for Internal Evaluation Errors

Action	Behaviour
throw_on_error	**Throws `boost::math::evaluation_error`**
errno_on_error	Sets `::errno` to `EDOM` and returns the closest approximation found.
ignore_error	Returns the closest approximation found.
user_error	Returns the result of `boost::math::policies::user_evaluation_error`: this function must be defined by the user.

Table 8. Possible Actions for Indeterminate Result Errors

Action	Behaviour
throw_on_error	Throws `std::domain_error`
errno_on_error	Sets `::errno` to `EDOM` and returns the same value as `ignore_error`.
ignore_error	**Returns a default result that depends on the function where the error occurred.**
user_error	Returns the result of `boost::math::policies::user_indeterminate_result_error`: this function must be defined by the user.

All these error conditions are in namespace boost::math::policies, made available, for example, a by namespace declaration using `namespace boost::math::policies;` or individual using declarations `using boost::math::policies::overflow_error;`.

Rationale

The flexibility of the current implementation should be reasonably obvious: the default behaviours were chosen based on feedback during the formal review of this library. It was felt that:

- Genuine errors should be flagged with exceptions rather than following C-compatible behaviour and setting `::errno`.
- Numeric underflow and denormalised results were not considered to be fatal errors in most cases, so it was felt that these should be ignored.
- If there is more than one error, only the first detected will be reported in the throw message.

Finding More Information

There are some pre-processor macro defines that can be used to change the policy defaults. See also the policy section.

An example is at the Policy tutorial in Changing the Policy Defaults.

Full source code of this typical example of passing a 'bad' argument (negative degrees of freedom) to Student's t distribution is in the error handling example.

The various kind of errors are described in more detail below.

Domain Errors

When a special function is passed an argument that is outside the range of values for which that function is defined, then the function returns the result of:

```
boost::math::policies::raise_domain_error<T>(FunctionName, Message, Val, Policy);
```

Where `T` is the floating-point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, Val is the value that was out of range, and Policy is the current policy in use for the function that was called.

The default policy behaviour of this function is to throw a std::domain_error C++ exception. But if the Policy is to ignore the error, or set global `::errno`, then a NaN will be returned.

This behaviour is chosen to assist compatibility with the behaviour of *ISO/IEC 9899:1999 Programming languages - C* and with the Draft Technical Report on C++ Library Extensions, 2005-06-24, section 5.2.1, paragraph 6:

> *"Each of the functions declared above shall return a NaN (Not a Number) if any argument value is a NaN, but it shall not report a domain error. Otherwise, each of the functions declared above shall report a domain error for just those argument values for which:*
>
> *"the function description's Returns clause explicitly specifies a domain, and those arguments fall outside the specified domain; or*
>
> *"the corresponding mathematical function value has a non-zero imaginary component; or*
>
> *"the corresponding mathematical function is not mathematically defined.*
>
> *"Note 2: A mathematical function is mathematically defined for a given set of argument values if it is explicitly defined for that set of argument values or if its limiting value exists and does not depend on the direction of approach."*

Note that in order to support information-rich error messages when throwing exceptions, `Message` must contain a Boost.Format recognised format specifier: the argument `Val` is inserted into the error message according to the specifier used.

For example if `Message` contains a "%1%" then it is replaced by the value of `Val` to the full precision of T, where as "%.3g" would contain the value of `Val` to 3 digits. See the Boost.Format documentation for more details.

Evaluation at a pole

When a special function is passed an argument that is at a pole without a well defined residual value, then the function returns the result of:

```
boost::math::policies::raise_pole_error<T>(FunctionName, Message, Val, Policy);
```

Where `T` is the floating point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, `Val` is the value of the argument that is at a pole, and Policy is the current policy in use for the function that was called.

The default behaviour of this function is to throw a std::domain_error exception. But error handling policies can be used to change this, for example to `ignore_error` and return NaN.

Note that in order to support information-rich error messages when throwing exceptions, `Message` must contain a Boost.Format recognised format specifier: the argument `val` is inserted into the error message according to the specifier used.

For example if `Message` contains a "%1%" then it is replaced by the value of `val` to the full precision of T, where as "%.3g" would contain the value of `val` to 3 digits. See the Boost.Format documentation for more details.

Numeric Overflow

When the result of a special function is too large to fit in the argument floating-point type, then the function returns the result of:

```
boost::math::policies::raise_overflow_error<T>(FunctionName, Message, Policy);
```

Where `T` is the floating-point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, and Policy is the current policy in use for the function that was called.

The default policy for this function is that `std::overflow_error` C++ exception is thrown. But if, for example, an `ignore_error` policy is used, then returns `std::numeric_limits<T>::infinity()`. In this situation if the type `T` doesn't support infinities, the maximum value for the type is returned.

Numeric Underflow

If the result of a special function is known to be non-zero, but the calculated result underflows to zero, then the function returns the result of:

```
boost::math::policies::raise_underflow_error<T>(FunctionName, Message, Policy);
```

Where `T` is the floating point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, and Policy is the current policy in use for the called function.

The default version of this function returns zero. But with another policy, like `throw_on_error`, throws an `std::underflow_error` C++ exception.

Denormalisation Errors

If the result of a special function is a denormalised value *z* then the function returns the result of:

```
boost::math::policies::raise_denorm_error<T>(z, FunctionName, Message, Policy);
```

Where `T` is the floating point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, and Policy is the current policy in use for the called function.

The default version of this function returns *z*. But with another policy, like `throw_on_error` throws an `std::underflow_error` C++ exception.

Evaluation Errors

When a special function calculates a result that is known to be erroneous, or where the result is incalculable then it calls:

```
boost::math::policies::raise_evaluation_error<T>(FunctionName, Message, Val, Policy);
```

Where `T` is the floating point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, `Val` is the erroneous value, and Policy is the current policy in use for the called function.

The default behaviour of this function is to throw a `boost::math::evaluation_error`.

Note that in order to support information rich error messages when throwing exceptions, `Message` must contain a Boost.Format recognised format specifier: the argument `val` is inserted into the error message according to the specifier used.

For example if `Message` contains a "%1%" then it is replaced by the value of `val` to the full precision of T, where as "%.3g" would contain the value of `val` to 3 digits. See the Boost.Format documentation for more details.

Indeterminate Result Errors

When the result of a special function is indeterminate for the value that was passed to it, then the function returns the result of:

```
boost::math::policies::raise_overflow_error<T>(FunctionName, Message, Val, Default, Policy);
```

Where `T` is the floating-point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, Val is the value for which the result is indeterminate, Default is an alternative default result that must be returned for `ignore_error` and `errno_on_erro` policies, and Policy is the current policy in use for the function that was called.

The default policy for this function is `ignore_error`: note that this error type is reserved for situations where the result is mathematically undefined or indeterminate, but there is none the less a convention for what the result should be: for example the C99 standard specifies that the result of 0^0 is 1, even though the result is actually mathematically indeterminate.

Rounding Errors

When one of the rounding functions round, trunc or modf is called with an argument that has no integer representation, or is too large to be represented in the result type then the value returned is the result of a call to:

```
boost::math::policies::raise_rounding_error<T>(FunctionName, Message, Val, Policy);
```

Where `T` is the floating point type passed to the function, `FunctionName` is the name of the function, `Message` is an error message describing the problem, `Val` is the erroneous argument, and Policy is the current policy in use for the called function.

The default behaviour of this function is to throw a `boost::math::rounding_error`.

Note that in order to support information rich error messages when throwing exceptions, `Message` must contain a Boost.Format recognised format specifier: the argument `val` is inserted into the error message according to the specifier used.

For example if `Message` contains a "%1%" then it is replaced by the value of `val` to the full precision of T, where as "%.3g" would contain the value of `val` to 3 digits. See the Boost.Format documentation for more details.

Errors from typecasts

Many special functions evaluate their results at a higher precision than their arguments in order to ensure full machine precision in the result: for example, a function passed a float argument may evaluate its result using double precision internally. Many of the errors listed above may therefore occur not during evaluation, but when converting the result to the narrower result type. The function:

```
template <class T, class Policy, class U>
T checked_narrowing_cast(U const& val, const char* function);
```

Is used to perform these conversions, and will call the error handlers listed above on overflow, underflow or denormalisation.

Compilers

This section contains some information about how various compilers work with this library. It is not comprehensive and updated experiences are always welcome. Some effort has been made to suppress unhelpful warnings but it is difficult to achieve this on all systems.

Table 9. Supported/Tested Compilers

Platform	Compiler	Has long double support	Notes
Windows	MSVC 7.1 and later	Yes	All tests OK. We aim to keep our headers warning free at level 4 with this compiler.
Windows	Intel 8.1 and later	Yes	All tests OK. We aim to keep our headers warning free at level 4 with this compiler. However, The tests cases tend to generate a lot of warnings relating to numeric underflow of the test data: these are harmless.
Windows	GNU Mingw32 C++	Yes	All tests OK. We aim to keep our headers warning free with -Wall with this compiler.
Windows	GNU Cygwin C++	No	All tests OK. We aim to keep our headers warning free with -Wall with this compiler. Long double support has been disabled because there are no native long double C std library functions available.
Windows	Borland C++ 5.8.2 (Developer studio 2006)	No	We have only partial compatibility with this compiler: Long double support has been disabled because the native long double C standard library functions really only forward to the double versions. This can result in unpredictable behaviour when using the long double overloads: for example `sqrtl` applied to a finite value, can result in an infinite result. Some functions still fail to compile, there are no known workarounds at present.
Windows 7/Netbeans 7.2	Clang 3.1	Yes	Spot examples OK. Expect all tests to compile and run OK.

Platform	Compiler	Has long double support	Notes
Linux	GNU C++ 3.4 and later	Yes	All tests OK. We aim to keep our headers warning free with -Wall with this compiler.
Linux	Clang 3.2	Yes	All tests OK.
Linux	Intel C++ 10.0 and later	Yes	All tests OK. We aim to keep our headers warning free with -Wall with this compiler. However, The tests cases tend to generate a lot of warnings relating to numeric underflow of the test data: these are harmless.
Linux	Intel C++ 8.1 and 9.1	No	All tests OK. Long double support has been disabled with these compiler releases because calling the standard library long double math functions can result in a segfault. The issue is Linux distribution and glibc version specific and is Intel bug report #409291. Fully up to date releases of Intel 9.1 (post version l_cc_c_9.1.046) shouldn't have this problem. If you need long double support with this compiler, then comment out the define of BOOST_MATH_NO_LONG_DOUBLE_MATH_FUNCTIONS at line 55 of boost/math/tools/config.hpp. We aim to keep our headers warning free with -Wall with this compiler. However, The tests cases tend to generate a lot of warnings relating to numeric underflow of the test data: these are harmless.
Linux	QLogic PathScale 3.0	Yes	Some tests involving conceptual checks fail to build, otherwise there appear to be no issues.
Linux	Sun Studio 12	Yes	Some tests involving function overload resolution fail to build, these issues should be rarely encountered in practice.

Platform	Compiler	Has long double support	Notes
Solaris	Sun Studio 12	Yes	Some tests involving function overload resolution fail to build, these issues should be rarely encountered in practice.
Solaris	GNU C++ 4.x	Yes	All tests OK. We aim to keep our headers warning free with -Wall with this compiler.
HP Tru64	Compaq C++ 7.1	Yes	All tests OK.
HP-UX Itanium	HP aCC 6.x	Yes	All tests OK. Unfortunately this compiler emits quite a few warnings from libraries upon which we depend (TR1, Array etc).
HP-UX PA-RISC	GNU C++ 3.4	No	All tests OK.
Apple Mac OS X, Intel	Darwin/GNU C++ 4.x	Yes	All tests OK.
Apple Mac OS X, PowerPC	Darwin/GNU C++ 4.x	No	All tests OK. Long double support has been disabled on this platform due to the rather strange nature of Darwin's 106-bit long double implementation. It should be possible to make this work if someone is prepared to offer assistance.
Apple Mac OS X,	Clang 3.2	Yes	All tests expected to be OK.
IBM AIX	IBM xlc 5.3	Yes	All tests pass except for our fpclassify tests which fail due to a bug in `std::numeric_limits`, the bug effects the test code, not fpclassify itself. The IBM compiler group are aware of the problem.

Table 10. Unsupported Compilers

Platform	Compiler
Windows	Borland C++ 5.9.2 (Borland Developer Studio 2007)
Windows	MSVC 6 and 7

If your compiler or platform is not listed above, please try running the regression tests: cd into boost-root/libs/math/test and do a:

```
bjam mytoolset
```

where "mytoolset" is the name of the Boost.Build toolset used for your compiler. The chances are that **many of the accuracy tests will fail at this stage** - don't panic - the default acceptable error tolerances are quite tight, especially for long double types with an extended exponent range (these cause more extreme test cases to be executed for some functions). You will need to cast an eye over the output from the failing tests and make a judgement as to whether the error rates are acceptable or not.

Configuration Macros

Almost all configuration details are set up automatically by `<boost\math\tools\config.hpp>`.

In normal use, only policy configuration macros are likely to be used. See policy reference.

For reference, information on Boost.Math macros used internally are described briefly below.

Table 11. Boost.Math Macros

MACRO	Notes
`BOOST_MATH_NO_LONG_DOUBLE_MATH_FUNCTIONS`	Do not produce or use long double functions: this macro gets set when the platform's long double or standard library long double support is absent or buggy.
`BOOST_MATH_USE_FLOAT128`	When set the numeric constants support the `__float128` data type with constants having the `Q` suffix.
`BOOST_MATH_DISABLE_FLOAT128`	When set the numeric constants do not use the `__float128` data type even if the compiler appears to support it.
`BOOST_MATH_NO_REAL_CONCEPT_TESTS`	Do not try to use real concept tests (hardware or software does not support real_concept type).
`BOOST_MATH_CONTROL_FP`	Controls FP hardware exceptions - our tests don't support hardware exceptions on MSVC. May get set to something like: `_control87(MCW_EM,MCW_EM)`.
`BOOST_MATH_NO_DEDUCED_FUNCTION_POINTERS`	This macro is used by our test cases, it is set when an assignment of a function template to a function pointer requires explicit template arguments to be provided on the function name.
`BOOST_MATH_USE_C99`	Use C99 math functions.
`BOOST_NO_NATIVE_LONG_DOUBLE_FP_CLASSIFY`	define if no native (or buggy) `fpclassify(long double)` even though the other C99 functions are present.
`BOOST_MATH_SMALL_CONSTANT(x)`	Helper macro used in our test cases to set underflowing constants set to zero if this would cause compiler issues.
`BOOST_MATH_BUGGY_LARGE_FLOAT_CONSTANTS`	Set if constants too large for a float, will cause "bad" values to be stored in the data, rather than infinity or a suitably large value.
`BOOST_MATH_STD_USING`	Provides `using` statements for many std:: (abs to sqrt) and boost::math (rounds, modf) functions. This allows these functions to be called unqualified so that if argument-dependent Argument Dependent Lookup fails to find a suitable overload, then the std:: versions will also be considered.
`BOOST_FPU_EXCEPTION_GUARD`	Used at the entrypoint to each special function to reset all FPU exception flags prior to internal calculations, and then merge the old and new exception flags on function exit. Used as a workaround on platforms or hardware that behave strangely if any FPU exception flags are set when calling standard library functions.
`BOOST_MATH_INSTRUMENT`	Define to output diagnostics for math functions. This is rather 'global' to Boost.Math and so coarse-grained that it will probably produce copious output! (Especially because full precision values are output). Designed primarily for internal use and development.

MACRO	Notes
`BOOST_MATH_INSTRUMENT_CODE(x)`	Output selected named variable, for example `BOOST_MATH_INSTRUMENT_CODE("guess = " << guess);` Used by `BOOST_MATH_INSTRUMENT`
`BOOST_MATH_INSTRUMENT_VARIABLE(name)`	Output selected variable, for example `BOOST_MATH_INSTRUMENT_VARIABLE(result);` Used by `BOOST_MATH_INSTRUMENT`
`BOOST_MATH_INSTRUMENT_FPU`	Output the state of the FPU's control flags.

Table 12. Boost.Math Tuning

Macros for Tuning performance options for specific compilers	Notes
`BOOST_MATH_POLY_METHOD`	See the performance tuning section.
`BOOST_MATH_RATIONAL_METHOD`	See the performance tuning section.
`BOOST_MATH_MAX_POLY_ORDER`	See the performance tuning section.
`BOOST_MATH_INT_TABLE_TYPE`	See the performance tuning section.
`BOOST_MATH_INT_VALUE_SUFFIX`	Helper macro for appending the correct suffix to integer constants which may actually be stored as reals depending on the value of BOOST_MATH_INT_TABLE_TYPE.

Policies

Policies are a powerful fine-grain mechanism that allow you to customise the behaviour of this library according to your needs. There is more information available in the policy tutorial and the policy reference.

Generally speaking, unless you find that the default policy behaviour when encountering 'bad' argument values does not meet your needs, you should not need to worry about policies.

Policies are a compile-time mechanism that allow you to change error-handling or calculation precision either program wide, or at the call site.

Although the policy mechanism itself is rather complicated, in practice it is easy to use, and very flexible.

Using policies you can control:

- How results from 'bad' arguments are handled, including those that cannot be fully evaluated.
- How accuracy is controlled by internal promotion to use more precise types.
- What working precision should be used to calculate results.
- What to do when a mathematically undefined function is used: Should this raise a run-time or compile-time error?
- Whether discrete functions, like the binomial, should return real or only integral values, and how they are rounded.
- How many iterations a special function is permitted to perform in a series evaluation or root finding algorithm before it gives up and raises an evaluation_error.

You can control policies:

- Using macros to change any default policy: the is the preferred method for installation wide policies.
- At your chosen namespace scope for distributions and/or functions: this is the preferred method for project, namespace, or translation unit scope policies.
- In an ad-hoc manner by passing a specific policy to a special function, or to a statistical distribution.

Thread Safety

The library is fully thread safe and re-entrant for all functions regards of the data type they are instantiated on. Thread safety limitations relating to user defined types present in previous releases (prior to 1.50.0) have been removed.

Performance

By and large the performance of this library should be acceptable for most needs. However, often the library has to make a choice whether to be accurate or fast and by default it chooses accuracy over speed. If you would rather have fast rather than fully accurate routines, then refer to the performance section for information and examples on how to achieve this.

In terms of the algorithms used, this library aims to use the same "best of breed" algorithms as many other libraries: the principle difference is that this library is implemented in C++ - taking advantage of all the abstraction mechanisms that C++ offers - where as most traditional numeric libraries are implemented in C or FORTRAN. Traditionally languages such as C or FORTRAN are perceived as easier to optimise than more complex languages like C++, so in a sense this library provides a good test of current compiler technology, and the "abstraction penalty" - if any - of C++ compared to other languages.

The three most important things you can do to ensure the best performance from this library are:

1. Turn on your compilers optimisations: the difference between "release" and "debug" builds can easily be a factor of 20.
2. Pick your compiler carefully: performance differences of up to 8 fold have been found between some Windows compilers for example.
3. Disable internal use of `long double`, this will reduce accuracy but typically yield a 2x speedup on modern x64 hardware/compilers.

The performance section contains more information on the performance of this library, what you can do to fine tune it, and how this library compares to some other open source alternatives.

If and How to Build a Boost.Math Library, and its Examples and Tests

Building a Library (shared, dynamic .dll or static .lib)

The first thing you need to ask yourself is "Do I need to build anything at all?" as the bulk of this library is header only: meaning you can use it just by #including the necessary header(s).

For most simple uses, including a header (or few) is best for compile time and program size.

Refer to C99 and C++ TR1 C-style Functions for pros and cons of using the TR1 components as opposed to the header only ones.

The *only* time you *need* to build the library is if you want to use the `extern "C"` functions declared in `<boost/math/tr1.hpp>`. To build this using Boost.Build, from a commandline boost-root directory issue a command like:

```
bjam toolset=gcc --with-math install
```

that will do the job on Linux, while:

```
bjam toolset=msvc --with-math --build-type=complete stage
```

will work better on Windows (leaving libraries built in sub-folder `/stage` below your Boost root directory). Either way you should consult the getting started guide for more information.

You can also build the libraries from your favourite IDE or command line tool: each `extern "C"` function declared in `<boost/math/tr1.hpp>` has its own source file with the same name in `libs/math/src/tr1`. Just select the sources corresponding to the functions you are using and build them into a library, or else add them directly to your project. Note that the directory `libs/math/src/tr1` will need to be in your compiler's #include path as well as the boost-root directory (MSVC Tools, Options, Projects and Solutions, VC++ Directories, Include files).

Note

If you are using a Windows compiler that supports auto-linking and you have built the sources yourself (or added them directly to your project) then you will need to prevent `<boost/math/tr1.hpp>` from trying to auto-link to the binaries that Boost.Build generates. You can do this by defining either BOOST_MATH_NO_LIB or BOOST_ALL_NO_LIB at project level (so the defines get passed to each compiler invocation).

Optionally the sources in `libs/math/src/tr1` have support for using `libs/math/src/tr1/pch.hpp` as a precompiled header *if your compiler supports precompiled headers.* Note that normally this header is a do-nothing include: to activate the header so that it #includes everything required by all the sources you will need to define BOOST_BUILD_PCH_ENABLED on the command line, both when building the pre-compiled header and when building the sources. Boost.Build will do this automatically when appropriate.

Building the Examples

The examples are all located in `libs/math/example`, they can all be built without reference to any external libraries, either with Boost.Build using the supplied Jamfile, or from your compiler's command line. The only requirement is that the Boost headers are in your compilers #include search path.

Building the Tests

The tests are located in `libs/math/test` and are best built using Boost.Build and the supplied Jamfile. If you plan to build them separately from your favourite IDE then you will need to add `libs/math/test` to the list of your compiler's search paths.

You will also need to build and link to the Boost.Regex library for many of the tests: this can built from the command line by following the getting started guide, using a command such as:

```
bjam toolset=gcc --with-regex install
```

or

```
bjam toolset=msvc --with-regex --build-type=complete stage
```

depending on whether you are on Linux or Windows.

Many of the tests have optional precompiled header support using the header `libs/math/test/pch.hpp`. Note that normally this header is a do-nothing include: to activate the header so that it #includes everything required by all the sources you will need to define BOOST_BUILD_PCH_ENABLED on the command line, both when building the pre-compiled header and when building the sources. Boost.Build will do this automatically when appropriate.

History and What's New

Currently open bug reports can be viewed here.

All bug reports including closed ones can be viewed here and here.

Math-2.10.0 (Boost-1.71)

- Catmull-Rom interpolator now works in C++11
- Cardinal quadratic B-spline interpolation
- Domain of elliptic integrals extended
- sin_pi and cos_pi performance improvements
- Forward-mode automatic differentiation
- Vector valued barycentric rational interpolation
- Ooura's method for evaluation of Fourier integrals
- Multiple compatibility issues with Multiprecision fixed
- Lambert-W fixed on a rare architecture

Math-2.9.0 (Boost-1.70)

- Add Lanczos smoothing derivatives
- Move `numerical_differentiation.hpp` from `boost/math/tools/` to `boost/math/differentiation/finite_difference.hpp`.
- Add mean, variance, skewness, kurtosis, median, Gini coefficient, and median absolute deviation to `tools/univariate_statistics.hpp`.
- Add correlation coefficients and covariance to `tools/bivariate_statistics.hpp`
- Add absolute Gini coefficient, Hoyer sparsity, oracle SNR, and the M_2M_4 SNR estimator to `tools/signal_statistics.hpp`.
- Add total variation, l0, l1, l2, and sup norms, as well as corresponding distance functions to `tools/norms.hpp`.
- Add move constructors for polynomials, support complex coefficients, add `.prime()` and `.integrate()` methods.
- Add `quadratic_roots` to `tools/roots.hpp`.
- Add support for complex-valued functions to Newton's method in `roots.hpp`.
- Add Catmull-Rom interpolator.
- Fix bug in `newton_raphson_iterate` where we could terminate incorrectly under certain exceptional cases.
- Suppressed warnings from use of `Q` suffix on constants when compiling with `g++ -Wall -pedantic`.

Math-2.8.0 (Boost-1.69)

- Add LambertW functions.
- Update integration routines to support complex valued integrands and contour integrals.
- Added the derivative of the Barycentric rational approximation.

- Updated continued fraction and series evaluation code to support complex types.
- Minor fixes to better support variable precision floating point types.
- Removed use of deprecated Boost.Endian in favour of Predef.
- Prevent logic error leading to infinite loop in toms748_solve. See #138.
- Fix mean and standard_deviation for extreme_value_distribution. See #139.
- Improve heuristics used in newton_raphson_iterate. See #145.
- Fix result of `erf(NaN)`. See #141.
- Big push to reduce GCC warnings. See #136.
- Refactor polynomial addition. See PR132.
- Fix for vxWorks having a `real` function in the global namespace. See PR131.
- Improve `sinc` approximations and add better tests.
- Fix typo in Student's T hypothesis testing documentation, see #143.

Math-2.7.1 (Boost-1.68)

- Continue to improve numerical integration routines, and in particular add support for contour integrals.
- Improve accuracy of erfc function's rational approximations.

Math-2.7.0 (Boost-1.66)

- Add Gauss and Gauss-Kronrod quadrature routines.
- Add double-exponential (tanh-sinh, exp-sinh and sinh-sinh) quadrature routines.
- Add Chebyshev polynomial evaluation, roots, integration, differentiation, and interpolation routines.

Math-2.6.0 (Boost-1.65)

New Features:

- Add cubic B-Spline interpolation functions, with thanks to Nick Thompson.
- Add barycentric rational interpolation functions, with thanks to Nick Thompson.
- Add adaptive trapezoidal quadrature, with thanks to Nick Thompson.
- Add support for the zeros and derivatives of Legendre polynomials, with thanks to Nick Thompson.

Patches:

- Improve polynomial GCD algorithms, with thanks to Jeremy Murphy.
- Removed gcd/lcm routines from Boost.Math - these are now in Boost.Integer.

Math-2.5.2 (Boost-1.64)

Patches:

- Big push to ensure all functions in also in C99 are compatible with Annex F.

- Improved accuracy of the Bessel functions I0, I1, K0 and K1, see 12066.

Math-2.5.1 (Boost-1.63)

Patches:

- Fixed evaluation of zero polynomial in polynomial.hpp, see 12532.
- Fixed missing header include in boost/math/tools/tuple.hpp, see 12537.

Math-2.5.0 (Boost-1.62)

New Features:

- Enabled all the special function code to work correctly with types whose precision can change at runtime: for example type `mpfr_float` from Boost.Multiprecision.

Patches:

- Fix tgamma_delta_ratio for cases where the delta is small compared to the base.
- Fix misc GCC-4.4 test failures.

Math-2.4.0 (Boost-1.61)

New Features:

- Polynomial arithmetic added to tools.

Math-2.3.0 (Boost-1.60)

New Features:

- Promote the root finding and function minimization code to first class citizens - these are now officially supported as part of the library.
- Allow the library to be used and tested with compiler exception handling support turned off. To better facilitate this the default error handling policies have been changed to *errno_on_error* but only when the compiler has no exception handling support.

Patches:

- Fix behaviour of the non-central chi-squared distribution when the non-centrality parameter is zero to match the chi-squared, see 11557.
- Fix comments in code for the hypergeometric to match what it actually does, also fixes the parameter access functions to return the correct values. See 11556.
- Stopped using hidden visiblity library build with the Oracle compiler as it leads to unresolved externals from the C++ standard library. See 11547.
- Fix unintended use of __declspec when building with Oracle C++. See 11546.
- Fix corner case bug in root bracketing code, see 11532.
- Add some missing typecasts in arguments to std::max in Bernoulli code. See 11453.
- Fix mistaken assumptions about the possible values for FLT_EVAL_METHOD. See 11429.
- Completely revamped performance testing and error-rate measuring code so we can more easily document how well (or not!) we're doing. This information will hopefully get more frequently updated in future as it's more or less automatically generated - see the *reporting* sub-directory for more information.

- Fix some corner cases in the beta, incomplete beta, and incomplete beta derivative. With thanks to Rocco Romeo.
- Reorganized the Bessel functions internally to improve the performance of Jn and Yn.
- Fixed skewness formula for triangular distribution, see #11768.
- Fixed some examples so they compile on Unix platforms which have an ::exception struct declared, see #11827.
- Correct mistake in triangular distribution skewness formula 11768, reported by Juan Leni.

Math-2.2.1

Patch release for Boost-1.58:

- Minor patch for Haiku support.
- Fix the decimal digit count for 128-bit floating point types.
- Fix a few documentation typos.

Math-2.2.0 (boost-1.58.0)

- Added two new special functions - trigamma and polygamma.
- Fixed namespace scope constants so they are constexpr on conforming compilers, see https://svn.boost.org/trac/boost/ticket/10901.
- Fixed various cases of spurious under/overflow in the incomplete beta and gamma functions, plus the elliptic integrals, with thanks to Rocco Romeo.
- Fix 3-arg legendre_p and legendre_q functions to not call the policy based overload if the final argument is not actually a policy.
- Cleaned up some dead code in the incomplete beta function, see #10985.
- Fixed extreme-value pdf for large valued inputs, see #10938.
- Large update to the Elliptic integral code to use Carlson's latest algorithms - these should be more stable, more accurate and slightly faster than before. Also added support for Carlson's RG integral.
- Added ellint_d, jacobi_zeta and heuman_lambda elliptic integrals.
- Switched documentation to use SVG rather than PNG graphs and equations - browsers seem to have finally caught up!

Math-2.1.0 (boost-1.57.0)

- Added Hyperexponential Distribution.
- Fix some spurious overflows in the incomplete gamma functions (with thanks to Rocco Romeo).
- Fix bug in derivative of incomplete beta when a = b = 0.5 - this also effects several non-central distributions, see 10480.
- Fixed some corner cases in round.
- Don't support 80-bit floats in cstdfloat.hpp if standard library support is broken.

Math-2.0.0 (Boost-1.56.0)

- **Breaking change**: moved a number of non-core headers that are predominantly used for internal maintenance into `libs/math/include_private`. The headers effected are `boost/math/tools/test_data.hpp`, `boost/math/tools/remez.hpp`, `boost/math/constants/generate.hpp`, `boost/math/tools/solve.hpp`, `boost/math/tools/test.hpp`. You can continue to use these headers by adding `libs/math/include_private` to your compiler's include path.

- **Breaking change**: A number of distributions and special functions were returning the maximum finite value rather than raising an overflow_error, this has now been fixed, which means these functions now behave as documented. However, since the default behavior on raising an overflow_error is to throw a `std::overflow_error` exception, applications which have come to reply rely on these functions not throwing may experience exceptions where they did not before. The special functions involved are gamma_p_inva, gamma_q_inva, ibeta_inva, ibetac_inva, ibeta_invb, ibetac_invb, gamma_p_inv, gamma_q_inv. The distributions involved are Pareto Distribution, Beta Distribution, Geometric Distribution, Negative Binomial Distribution, Binomial Distribution, Chi Squared Distribution, Gamma Distribution, Inverse chi squared Distribution, Inverse Gamma Distribution. See #10111.
- Fix round and trunc functions so they can be used with integer arguments, see #10066.
- Fix Halley iteration to handle zero derivative (with non-zero second derivative), see #10046.

Math-1.9.1

- Fix Geometric distribution use of Policies, see #9833.
- Fix corner cases in the negative binomial distribution, see #9834.
- Fix compilation failures on Mac OS.

Math-1.9.0

- Changed version number to new Boost.Math specific version now that we're in the modular Boost world.
- Added Bernoulli numbers, changed arbitrary precision tgamma/lgamma to use Sterling's approximation (from Nikhar Agrawal).
- Added first derivatives of the Bessel functions: cyl_bessel_j_prime, cyl_neumann_prime, cyl_bessel_i_prime, cyl_bessel_k_prime, sph_bessel_prime and sph_neumann_prime (from Anton Bikineev).
- Fixed buggy Student's t example code, along with docs for testing sample means for equivalence.
- Documented `max_iter` parameter in root finding code better, see #9225.
- Add option to explicitly enable/disable use of __float128 in constants code, see #9240.
- Cleaned up handling of negative values in Bessel I0 and I1 code (removed dead code), see #9512.
- Fixed handling of very small values passed to tgamma and lgamma so they don't generate spurious overflows (thanks to Rocco Romeo).
- #9672 PDF and CDF of a Laplace distribution throwing domain_error Random variate can now be infinite.
- Fixed several corner cases in rising_factorial, falling_factorial and tgamma_delta_ratio with thanks to Rocco Romeo.
- Fixed several corner cases in rising_factorial, falling_factorial and tgamma_delta_ratio (thanks to Rocco Romeo).
- Removed constant `pow23_four_minus_pi` whose value did not match the name (and was unused by Boost.Math), see #9712.

Boost-1.55

- Suppress numerous warnings (mostly from GCC-4.8 and MSVC) #8384, #8855, #9107, #9109..
- Fixed PGI compilation issue #8333.
- Fixed PGI constant value initialization issue that caused erf to generate incorrect results #8621.
- Prevent macro expansion of some C99 macros that are also C++ functions #8732 and #8733..
- Fixed Student's T distribution to behave correctly with huge degrees of freedom (larger than the largest representable integer) #8837.

- Make some core functions usable with `long double` even when the platform has no standard library `long double` support #8940.
- Fix error handling of distributions to catch invalid scale and location parameters when the random variable is infinite #9042 and #9126.
- Add workaround for broken <tuple> in Intel C++ 14 #9087.
- Improve consistency of argument reduction in the elliptic integrals #9104.
- Fix bug in inverse incomplete beta that results in cancellation errors when the beta function is really an arcsine or Student's T distribution.
- Fix issue in Bessel I and K function continued fractions that causes spurious over/underflow.
- Add improvement to non-central chi squared distribution quantile due to Thomas Luu, Fast and accurate parallel computation of quantile functions for random number generation, Doctorial Thesis 2016. Efficient and Accurate Parallel Inversion of the Gamma Distribution, Thomas Luu

Boost-1.54

- Major reorganization to incorporate other Boost.Math like Integer Utilities Integer Utilities (Greatest Common Divisor and Least Common Multiple), quaternions and octonions. Making new chapter headings.
- Added many references to Boost.Multiprecision and `cpp_dec_float_50` as an example of a User-defined Type (UDT).
- Added Clang to list of supported compilers.
- Fixed constants to use a thread-safe cache of computed values when used at arbitrary precision.
- Added finding zeros of Bessel functions `cyl_bessel_j_zero`, `cyl_neumann_zero`, `airy_ai_zero` and `airy_bi_zero`(by Christopher Kormanyos).
- More accuracy improvements to the Bessel J and Y functions from Rocco Romeo.
- Fixed nasty cyclic dependency bug that caused some headers to not compile #7999.
- Fixed bug in tgamma that caused spurious overflow for arguments between 142.5 and 143.
- Fixed bug in raise_rounding_error that caused it to return an incorrect result when throwing an exception is turned off #7905.
- Added minimal __float128 support.
- Fixed bug in edge-cases of poisson quantile #8308.
- Adjusted heuristics used in Halley iteration to cope with inverting the incomplete beta in tricky regions where the derivative is flatlining. Example is computing the quantile of the Fisher F distribution for probabilities smaller than machine epsilon. See ticket #8314.

Boost-1.53

- Fixed issues #7325, #7415 and #7416, #7183, #7649, #7694, #4445, #7492, #7891, #7429.
- Fixed mistake in calculating pooled standard deviation in two-sample students t example #7402.
- Improve complex acos/asin/atan, see #7290, #7291.
- Improve accuracy in some corner cases of cyl_bessel_j and gamma_p/gamma_q thanks to suggestions from Rocco Romeo.
- Improve accuracy of Bessel J and Y for integer orders thanks to suggestions from Rocco Romeo.

Boost-1.52

- Corrected moments for small degrees of freedom #7177 (reported by Thomas Mang).
- Added Airy functions and Jacobi Elliptic functions.
- Corrected failure to detect bad parameters in many distributions #6934 (reported by Florian Schoppmann) by adding a function check_out_of_range to test many possible bad parameters. This test revealed several distributions where the checks for bad parameters were ineffective, and these have been rectified.
- Fixed issue in Hankel functions that causes incorrect values to be returned for $x < 0$ and ν odd, see #7135.
- Fixed issues #6517, #6362, #7053, #2693, #6937, #7099.
- Permitted infinite degrees of freedom #7259 implemented using the normal distribution (requested by Thomas Mang).
- Much enhanced accuracy for large degrees of freedom ν and/or large non-centrality δ by switching to use the Students t distribution (or Normal distribution for infinite degrees of freedom) centered at delta, when δ / (4 * ν) < epsilon for the floating-point type in use. #7259. It was found that the incomplete beta was suffering from serious cancellation errors when degrees of freedom was very large. (That has now been fixed in our code, but any code based on Didonato and Morris's original papers (probably every implementation out there actually) will have the same issue).

Boost-1.51

See Boost-1.52 - some items were added but not listed in time for the release.

Boost-1.50

- Promoted math constants to be 1st class citizens, including convenient access to the most widely used built-in float, double, long double via three namespaces.
- Added the Owen's T function and Skew Normal distribution written by Benjamin Sobotta: see Owens T and skew_normal_distrib.
- Added Hankel functions cyl_hankel_1, cyl_hankel_2, sph_hankel_1 and sph_hankel_2.
- Corrected issue #6627 nonfinite_num_put formatting of 0.0 is incorrect based on a patch submitted by K R Walker.
- Changed constant initialization mechanism so that it is thread safe even for user-defined types, also so that user defined types get the full precision of the constant, even when `long double` does not. So for example 128-bit rational approximations will work with UDT's and do the right thing, even though `long double` may be only 64 or 80 bits.
- Fixed issue in `bessel_jy` which causes $Y_{8.5}(4\pi)$ to yield a NaN.

Boost-1.49

- Deprecated wrongly named `twothirds` math constant in favour of `two_thirds` (with underscore separator). (issue #6199).
- Refactored test data and some special function code to improve support for arbitary precision and/or expression-template-enabled types.
- Added new faster zeta function evaluation method.

Fixed issues:

- Corrected CDF complement for Laplace distribution (issue #6151).
- Corrected branch cuts on the complex inverse trig functions, to handle signed zeros (issue #6171).
- Fixed bug in `bessel_yn` which caused incorrect overflow errors to be raised for negative *n* (issue #6367).
- Also fixed minor/cosmetic/configuration issues #6120, #6191, #5982, #6130, #6234, #6307, #6192.

Boost-1.48

- Added new series evaluation methods to the cyclic Bessel I, J, K and Y functions. Also taken great care to avoid spurious over and underflow of these functions. Fixes issue #5560
- Added an example of using Inverse Chi-Squared distribution for Bayesian statistics, provided by Thomas Mang.
- Added tests to use improved version of lexical_cast which handles C99 nonfinites without using globale facets.
- Corrected wrong out-of-bound uniform distribution CDF complement values #5733.
- Enabled long double support on OpenBSD (issue #6014).
- Changed nextafter and related functions to behave in the same way as other implementations - so that nextafter(+INF, 0) is a finite value (issue #5832).
- Changed tuple include configuration to fix issue when using in conjunction with Boost.Tr1 (issue #5934).
- Changed class eps_tolerance to behave correctly when both ends of the range are zero (issue #6001).
- Fixed missing include guards on prime.hpp (issue #5927).
- Removed unused/undocumented constants from constants.hpp (issue #5982).
- Fixed missing std:: prefix in nonfinite_num_facets.hpp (issue #5914).
- Minor patches for Cray compiler compatibility.

Boost-1.47

- Added changesign function to sign.hpp to facilitate addition of nonfinite facets.
- Addition of nonfinite facets from Johan Rade, with tests, examples of use for C99 format infinity and NaN, and documentation.
- Added tests and documentation of changesign from Johan Rade.

Boost-1.46.1

- Fixed issues #5095, #5113.

Boost-1.46.0

- Added Wald, Inverse Gaussian and geometric distributions.
- Added information about configuration macros.
- Added support for mpreal as a real-numbered type.

Boost-1.45.0

- Added warnings about potential ambiguity with std random library in distribution and function names.
- Added inverse gamma distribution and inverse chi_square and scaled inverse chi_square.
- Editorial revision of documentation, and added FAQ.

Boost-1.44.0

- Fixed incorrect range and support for Rayleigh distribution.

- Fixed numerical error in the quantile of the Student's T distribution: the function was returning garbage values for non-integer degrees of freedom between 2 and 3.

Boost-1.41.0

- Significantly improved performance for the incomplete gamma function and its inverse.

Boost-1.40.0

- Added support for MPFR as a bignum type.
- Added some full specializations of the policy classes to reduce compile times.
- Added logistic and hypergeometric distributions, from Gautam Sewani's Google Summer of Code project.
- Added Laplace distribution submitted by Thijs van den Berg.
- Updated performance test code to include new distributions, and improved the performance of the non-central distributions.
- Added SSE2 optimised Lanczos approximation code, from Gautam Sewani's Google Summer of Code project.
- Fixed bug in cyl_bessel_i that used an incorrect approximation for ν = 0.5, also effects the non-central Chi Square Distribution when ν = 3, see bug report #2877.
- Fixed minor bugs #2873.

Boost-1.38.0

- Added Johan Råde's optimised floating point classification routines.
- Fixed code so that it compiles in GCC's -pedantic mode (bug report #1451).

Boost-1.37.0

- Improved accuracy and testing of the inverse hypergeometric functions.

Boost-1.36.0

- Added Noncentral Chi Squared Distribution.
- Added Noncentral Beta Distribution.
- Added Noncentral F Distribution.
- Added Noncentral T Distribution.
- Added Exponential Integral Functions.
- Added Zeta Function.
- Added Rounding and Truncation functions.
- Added Compile time powers of runtime bases.
- Added SSE2 optimizations for Lanczos evaluation.

Boost-1.35.0: Post Review First Official Release

- Added Policy based framework that allows fine grained control over function behaviour.

- **Breaking change:** Changed default behaviour for domain, pole and overflow errors to throw an exception (based on review feedback), this behaviour can be customised using Policy's.
- **Breaking change:** Changed exception thrown when an internal evaluation error occurs to boost::math::evaluation_error.
- **Breaking change:** Changed discrete quantiles to return an integer result: this is anything up to 20 times faster than finding the true root, this behaviour can be customised using Policy's.
- Polynomial/rational function evaluation is now customisable and hopefully faster than before.
- Added performance test program.

Milestone 4: Second Review Candidate (1st March 2007)

- Moved Xiaogang Zhang's Bessel Functions code into the library, and brought them into line with the rest of the code.
- Added C# "Distribution Explorer" demo application.

Milestone 3: First Review Candidate (31st Dec 2006)

- Implemented the main probability distribution and density functions.
- Implemented digamma.
- Added more factorial functions.
- Implemented the Hermite, Legendre and Laguerre polynomials plus the spherical harmonic functions from TR1.
- Moved Xiaogang Zhang's elliptic integral code into the library, and brought them into line with the rest of the code.
- Moved Hubert Holin's existing Boost.Math special functions into this library and brought them into line with the rest of the code.

Milestone 2: Released September 10th 2006

- Implement preview release of the statistical distributions.
- Added statistical distributions tutorial.
- Implemented root finding algorithms.
- Implemented the inverses of the incomplete gamma and beta functions.
- Rewrite erf/erfc as rational approximations (valid to 128-bit precision).
- Integrated the statistical results generated from the test data with Boost.Test: uses a database of expected results, indexed by test, floating point type, platform, and compiler.
- Improved lgamma near 1 and 2 (rational approximations).
- Improved erf/erfc inverses (rational approximations).
- Implemented Rational function generation (the Remez method).

Milestone 1: Released March 31st 2006

- Implement gamma/beta/erf functions along with their incomplete counterparts.
- Generate high quality test data, against which future improvements can be judged.
- Provide tools for the evaluation of infinite series, continued fractions, and rational functions.

- Provide tools for testing against tabulated test data, and collecting statistics on error rates.
- Provide sufficient docs for people to be able to find their way around the library.

SVN Revisions:

Sandbox and trunk last synchonised at revision: .

C99 and C++ TR1 C-style Functions

Many of the special functions included in this library are also a part of the either the C99 Standard ISO/IEC 9899:1999 or the Technical Report on C++ Library Extensions. Therefore this library includes a thin wrapper header `boost/math/tr1.hpp` that provides compatibility with these two standards.

There are various pros and cons to using the library in this way:

Pros:

- The header to include is lightweight (i.e. fast to compile).
- The functions have extern "C" linkage, and so are usable from other languages (not just C and C++).
- C99 and C++ TR1 Standard compatibility.

Cons:

- You will need to compile and link to the external Boost.Math libraries.
- Limited to support for the types, `float`, `double` and `long double`.
- Error handling is handled via setting ::errno and returning NaN's and infinities: this may be less flexible than an C++ exception based approach.

Note

The separate libraries are required **only** if you choose to use boost/math/tr1.hpp rather than some other Boost.Math header, the rest of Boost.Math remains header-only.

The separate libraries required in order to use tr1.hpp can be compiled using bjam from within the libs/math/build directory, or from the Boost root directory using the usual Boost-wide install procedure. Alternatively the source files are located in libs/math/src and each have the same name as the function they implement. The various libraries are named as follows:

Name	Type	Functions
boost_math_c99f-<suffix>	float	C99 Functions
boost_math_c99-<suffix>	double	C99 Functions
boost_math_c99l-<suffix>	long double	C99 Functions
boost_math_tr1f-<suffix>	float	TR1 Functions
boost_math_tr1-<suffix>	double	TR1 Functions
boost_math_tr1l-<suffix>	long double	TR1 Functions

Where `<suffix>` encodes the compiler and build options used to build the libraries: for example "libboost_math_tr1-vc80-mt-gd.lib" would be the statically linked TR1 library to use with Visual C++ 8.0, in multithreading debug mode, with the DLL VC++ runtime, where as "boost_math_tr1-vc80-mt.lib" would be import library for the TR1 DLL to be used with Visual C++ 8.0 with the release multithreaded DLL VC++ runtime. Refer to the getting started guide for a full explanation of the `<suffix>` meanings.

Note

Visual C++ users will typically have the correct library variant to link against selected for them by boost/math/tr1.hpp based on your compiler settings.

Users will need to define BOOST_MATH_TR1_DYN_LINK when building their code if they want to link against the DLL versions of these libraries rather than the static versions.

Users can disable auto-linking by defining BOOST_MATH_TR1_NO_LIB when building: this is typically only used when linking against a customised build of the libraries.

Note

Linux and Unix users will generally only have one variant of these libraries installed, and can generally just link against -lboost_math_tr1 etc.

Usage Recomendations

This library now presents the user with a choice:

- To include the header only versions of the functions and have an easier time linking, but a longer compile time.
- To include the TR1 headers and link against an external library.

Which option you choose depends largely on how you prefer to work and how your system is set up.

For example a casual user who just needs the acosh function, would probably be better off including `<boost/math/special_functions/acosh.hpp>` and using `boost::math::acosh(x)` in their code.

However, for large scale software development where compile times are significant, and where the Boost libraries are already built and installed on the system, then including `<boost/math/tr1.hpp>` and using `boost::math::tr1::acosh(x)` will speed up compile times, reduce object files sizes (since there are no templates being instantiated any more), and also speed up debugging runtimes - since the externally compiled libraries can be compiler optimised, rather than built using full settings - the difference in performance between release and debug builds can be as much as 20 times, so for complex applications this can be a big win.

Supported C99 Functions

See also the quick reference guide for these functions.

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

typedef unspecified float_t;
typedef unspecified double_t;

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

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

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

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

double round(double x);
float roundf(float x);
long double roundl(long double x);

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

double trunc(double x);
float truncf(float x);
long double truncl(long double x);

}}}} // namespaces
```

Supported TR1 Functions

See also the quick reference guide for these functions.

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

// [5.2.1.1] 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);

// [5.2.1.2] 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);

// [5.2.1.3] beta function:
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// [5.2.1.4] (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);

// [5.2.1.5] (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);

// [5.2.1.6] (complete) elliptic integral of the third kind:
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);

// [5.2.1.8] 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);

// [5.2.1.9] 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);

// [5.2.1.10] 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);

// [5.2.1.11] 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);

// [5.2.1.12] (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);

// [5.2.1.13] (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);
// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// [5.2.1.15] exponential integral:
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// [5.2.1.16] Hermite polynomials:
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// [5.2.1.18] Laguerre polynomials:
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// [5.2.1.19] Legendre polynomials:
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// [5.2.1.20] Riemann zeta function:
double riemann_zeta(double);
float riemann_zetaf(float);
long double riemann_zetal(long double);

// [5.2.1.21] 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);

// [5.2.1.22] 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);

// [5.2.1.23] 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);

}}}} // namespaces
```

In addition sufficient additional overloads of the `double` versions of the above functions are provided, so that calling the function with any mixture of `float`, `double`, `long double`, or *integer* arguments is supported, with the return type determined by the *result type calculation rules*.

Currently Unsupported C99 Functions

```
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);

double log2(double x);
float log2f(float x);
long double log2l(long double x);

double logb(double x);
float logbf(float x);
long double logbl(long double x);

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);

double nan(const char *str);
float nanf(const char *str);
long double nanl(const char *str);

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

double remquo(double x, double y, int *pquo);
float remquof(float x, float y, int *pquo);
long double remquol(long double x, long double y, int *pquo);

double rint(double x);
float rintf(float x);
long double rintl(long double x);

double scalbln(double x, long ex);
```

```
float scalblnf(float x, long ex);
long double scalblnl(long double x, long ex);

double scalbn(double x, int ex);
float scalbnf(float x, int ex);
long double scalbnl(long double x, int ex);
```

Currently Unsupported TR1 Functions

```
// [5.2.1.7] 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);

// [5.2.1.17] 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);
```

Boost.Math Frequently Asked Questions (FAQs)

1. *I'm a FORTRAN/NAG/SPSS/SAS/Cephes/MathCad/R user and I don't see where the functions like dnorm(mean, sd) are in Boost.Math?*

   Nearly all are provided, and many more like mean, skewness, quantiles, complements ... but Boost.Math makes full use of C++, and it looks a bit different. But do not panic! See section on construction and the many examples. Briefly, the distribution is constructed with the parameters (like location and scale) (things after the | in representation like P(X=k|n, p) or ; in a common represention of pdf f(x; μσ^2). Functions like pdf, cdf are called with the name of that distribution and the random variate often called x or k. For example, `normal my_norm(0, 1); pdf(my_norm, 2.0);`

2. I'm a user of New SAS Functions for Computing Probabilities.

   You will find the interface more familar, but to be able to select a distribution (perhaps using a string) see the Extras/Future Directions section, and /boost/libs/math/dot_net_example/boost_math.cpp for an example that is used to create a C# (C sharp) utility (that you might also find useful): see Statistical Distribution Explorer.

3. *I'm allegic to reading manuals and prefer to learn from examples.*

   Fear not - you are not alone! Many examples are available for functions and distributions. Some are referenced directly from the text. Others can be found at `\boost_latest_release\libs\math\example`, for example If you are a Visual Studio user, you should be able to create projects from each of these, making sure that the Boost library is in the include directories list (there are usually NO libraries that must be built).

4. *How do I make sure that the Boost library is in the Visual Studio include directories list?*

   You can add an include path, for example, your Boost place /boost-latest_release, for example `X:/boost_1_70_0/` if you have a separate partition X for Boost releases. Or you can use an environment variable BOOST_ROOT set to your Boost place, and include that. Visual Studio before 2010 provided Tools, Options, VC++ Directories to control directories: Visual Studio 2010 instead provides property sheets to assist. You may find it convenient to create a new one adding \boost-latest_release; to the existing include items in $(IncludePath).

5. *I'm a FORTRAN/NAG/SPSS/SAS/Cephes/MathCad/R user and I don't see where the properties like mean, median, mode, variance, skewness of distributions are in Boost.Math?*

   They are all available (if defined for the parameters with which you constructed the distribution) via Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

6. *I am a C programmer. Can I user Boost.Math with C?*

   Yes you can, including all the special functions, and TR1 functions like isnan. They appear as C functions, by being declared as "extern C".

7. *I am a C# (Basic? F# FORTRAN? Other CLI?) programmer. Can I use Boost.Math with C#? (or ...)?*

   Yes you can, including all the special functions, and TR1 functions like isnan. But you **must build the Boost.Math as a dynamic library (.dll) and compile with the /CLI option**. See the boost/math/dot_net_example folder which contains an example that builds a simple statistical distribution app with a GUI. See Statistical Distribution Explorer

8. *What these "policies" things for?*

   Policies are a powerful (if necessarily complex) fine-grain mechanism that allow you to customise the behaviour of the Boost.Math library according to your precise needs. See Policies. But if, very probably, the default behaviour suits you, you don't need to know more.

9. *I am a C user and expect to see global C-style `::errno` set for overflow/errors etc?*

   You can achieve what you want - see error handling policies and user error handling and many examples.

10. *I am a C user and expect to silently return a max value for overflow?*

    You (and C++ users too) can return whatever you want on overflow - see overflow_error and error handling policies and several examples.

11. *I don't want any error message for overflow etc?*

    You can control exactly what happens for all the abnormal conditions, including the values returned. See domain_error, overflow_error error handling policies user error handling etc and examples.

12. *My environment doesn't allow and/or I don't want exceptions. Can I still user Boost.Math?*

    Yes but you must customise the error handling: see user error handling and changing policies defaults .

13. *The docs are several hundreds of pages long! Can I read the docs off-line or on paper?*

    Yes - you can download the Boost current release of most documentation as a zip of pdfs (including Boost.Math) from Sourceforge, for example https://sourceforge.net/projects/boost/files/boost-docs/1.45.0/boost_pdf_1_45_0.tar.gz/download. And you can print any pages you need (or even print all pages - but be warned that there are several hundred!). Both html and pdf versions are highly hyperlinked. The entire Boost.Math pdf can be searched with Adobe Reader, Edit, Find ... This can often find what you seek, a partial substitute for a full index.

14. *I want a compact version for an embedded application. Can I use float precision?*

    Yes - by selecting RealType template parameter as float: for example normal_distribution<float> your_normal(mean, sd); (But double may still be used internally, so space saving may be less that you hope for). You can also change the promotion policy, but accuracy might be much reduced.

15. *I seem to get somewhat different results compared to other programs. Why?*

    We hope Boost.Math to be more accurate: our priority is accuracy (over speed). See the section on accuracy. But for evaluations that require iterations there are parameters which can change the required accuracy (see Policies). You might be able to squeeze a little more (or less) accuracy at the cost of runtime.

16. *Will my program run more slowly compared to other math functions and statistical libraries?*

    Probably, thought not always, and not by too much: our priority is accuracy. For most functions, making sure you have the latest compiler version with all optimisations switched on is the key to speed. For evaluations that require iteration, you may be able to gain a little more speed at the expense of accuracy. See detailed suggestions and results on performance.

17. *How do I handle infinity and NaNs portably?*

    See nonfinite fp_facets for Facets for Floating-Point Infinities and NaNs.

18. *Where are the pre-built libraries?*

    Good news - you probably don't need any! - just `#include <boost/`*math/distribution_you_want*`>`. But in the unlikely event that you do, see building libraries.

19. *I don't see the function or distribution that I want.*

    You could try an email to ask the authors - but no promises!

20. *I need more decimal digits for values/computations.*

    You can use Boost.Math with Boost.Multiprecision: typically cpp_dec_float is a useful user-defined type to provide a fixed number of decimal digits, usually 50 or 100.

21. Why can't I write something really simple like `cpp_int one(1); cpp_dec_float_50 two(2); one * two;`

Because `cpp_int` might be bigger than `cpp_dec_float can hold`, so you must make an **explicit** conversion. See mixed multiprecision arithmetic and conversion.

22. *How do I choose between Boost.Multiprecision cpp_bin_50 and cpp_dec_50?*

    Unless you have a specific reason to choose `cpp_dec_`, then the default choice should be `cpp_bin_`, for example using the convenience `typedefs` like `boost::multiprecision::cpp_bin_50` or `boost::multiprecision::cpp_bin_100`.

    In general, both work well and give the same results and at roughly the same speed with `cpp_dec_50` sometimes faster.

    cpp_dec_ was developed first paving the way for cpp_bin_. cpp_dec_ has several guard digits and is not rounded at all, using 'brute force' to get the promised number of decimal digits correct, but making it difficult to reason about precision and computational uncertainty, for example see **https://svn.boost.org/trac10/ticket/12133**. It also has a fast but imprecise division operator giving surprising results sometimes, see **https://svn.boost.org/trac10/ticket/11178**.

    cpp_bin_ is correctly/exactly rounded making it possible to reason about both the precision and rounding of the results.

Contact Info and Support

The main support for this library is via the Boost mailing lists:

- Use the boost-user list for general support questions.
- Use the boost-developer list for discussion about implementation and or submission of extensions.

You can also find JM at john - at - johnmaddock.co.uk and PAB at pbristow - at - hetp.u-net.com.

Floating Point Utilities

Rounding Truncation and Integer Conversion

Rounding Functions

```
#include <boost/math/special_functions/round.hpp>
```

```
template <class T>
T round(const T& v);

template <class T, class Policy>
T round(const T& v, const Policy&);

template <class T>
int iround(const T& v);

template <class T, class Policy>
int iround(const T& v, const Policy&);

template <class T>
long lround(const T& v);

template <class T, class Policy>
long lround(const T& v, const Policy&);

template <class T>
long long llround(const T& v);

template <class T, class Policy>
long long llround(const T& v, const Policy&);
```

These functions return the closest integer to the argument *v*.

Halfway cases are rounded away from zero, regardless of the current rounding direction.

If the argument *v* is either non-finite or else outside the range of the result type, then returns the result of rounding_error: by default this throws an instance of `boost::math::rounding_error`.

Truncation Functions

```
#include <boost/math/special_functions/trunc.hpp>
```

```
template <class T>
T trunc(const T& v);

template <class T, class Policy>
T trunc(const T& v, const Policy&);

template <class T>
int itrunc(const T& v);

template <class T, class Policy>
int itrunc(const T& v, const Policy&);

template <class T>
long ltrunc(const T& v);

template <class T, class Policy>
long ltrunc(const T& v, const Policy&);

template <class T>
long long lltrunc(const T& v);

template <class T, class Policy>
long long lltrunc(const T& v, const Policy&);
```

The trunc functions round their argument to the integer value, nearest to but no larger in magnitude than the argument.

For example `itrunc(3.7)` would return `3` and `ltrunc(-4.6)` would return `-4`.

If the argument *v* is either non-finite or else outside the range of the result type, then returns the result of rounding_error: by default this throws an instance of `boost::math::rounding_error`.

Integer and Fractional Part Splitting (modf)

```
#include <boost/math/special_functions/modf.hpp>
```

```
template <class T>
T modf(const T& v, T* ipart);

template <class T, class Policy>
T modf(const T& v, T* ipart, const Policy&);

template <class T>
T modf(const T& v, int* ipart);

template <class T, class Policy>
T modf(const T& v, int* ipart, const Policy&);

template <class T>
T modf(const T& v, long* ipart);

template <class T, class Policy>
T modf(const T& v, long* ipart, const Policy&);

template <class T>
T modf(const T& v, long long* ipart);

template <class T, class Policy>
T modf(const T& v, long long* ipart, const Policy&);
```

The `modf` functions store the integer part of *v* in `*ipart` and return the fractional part of *v*. The sign of the integer and fractional parts are the same as the sign of *v*.

If the argument *v* is either non-finite or else outside the range of the result type, then returns the result of rounding_error: by default this throws an instance of `boost::math::rounding_error`.

Floating-Point Classification: Infinities and NaNs

Synopsis

```
#define FP_ZERO        /* implementation specific value */
#define FP_NORMAL      /* implementation specific value */
#define FP_INFINITE    /* implementation specific value */
#define FP_NAN         /* implementation specific value */
#define FP_SUBNORMAL   /* implementation specific value */

template <class T>
int fpclassify(T t);

template <class T>
bool isfinite(T z); // Neither infinity nor NaN.

template <class T>
bool isinf(T t); // Infinity (+ or -).

template <class T>
bool isnan(T t); // NaN.

template <class T>
bool isnormal(T t); // isfinite and not denormalised.

#include <boost\math\special_functions\fpclassify.hpp>
```

to use these functions.

Description

These functions provide the same functionality as the macros with the same name in C99, indeed if the C99 macros are available, then these functions are implemented in terms of them, otherwise they rely on `std::numeric_limits<>` to function.

Note that the definition of these functions *does not suppress the definition of these names as macros by math.h* on those platforms that already provide these as macros. That mean that the following have differing meanings:

```
using namespace boost::math;

// This might call a global macro if defined,
// but might not work if the type of z is unsupported
// by the std lib macro:
isnan(z);
//
// This calls the Boost version
// (found via the "using namespace boost::math" declaration)
// it works for any type that has numeric_limits support for type z:
(isnan)(z);
//
// As above but with explicit namespace qualification.
(boost::math::isnan)(z);
//
// This will cause a compiler error if isnan is a native macro:
boost::math::isnan(z);
// So always use instead:
(boost::math::isnan)(z);
//
// You can also add a using statement,
// globally to a .cpp file, or to a local function in a .hpp file.
using boost::math::isnan;
// so you can write the shorter and less cluttered
(isnan)(z)
// But, as above, if isnan is a native macro, this causes a compiler error,
// because the macro always 'gets' the name first, unless enclosed in () brackets.
```

Detailed descriptions for each of these functions follows:

```
template <class T>
int fpclassify(T t);
```

Returns an integer value that classifies the value *t*:

fpclassify value	**class of t.**
FP_ZERO	If *t* is zero.
FP_NORMAL	If *t* is a non-zero, non-denormalised finite value.
FP_INFINITE	If *t* is plus or minus infinity.
FP_NAN	If *t* is a NaN.
FP_SUBNORMAL	If *t* is a denormalised number.

```
template <class T>
bool isfinite(T z);
```

Returns true only if *z* is not an infinity or a NaN.

```
template <class T>
bool isinf(T t);
```

Returns true only if *z* is plus or minus infinity.

```
template <class T>
bool isnan(T t);
```

Returns true only if *z* is a NaN.

```
template <class T>
bool isnormal(T t);
```

Returns true only if *z* is a normal number (not zero, infinite, NaN, or denormalised).

Floating-point format

If you wish to find details of the floating-point format for any particular processor, there is a program

inspect_fp.cpp

by Johan Rade which can be used to print out the processor type, endianness, and detailed bit layout of a selection of floating-point values, including infinity and NaNs.

Sign Manipulation Functions

Synopsis

```
#include <boost/math/special_functions/sign.hpp>
```

```
namespace boost{ namespace math{

template<class T>
int signbit(T x);

template <class T>
int sign (const T& z);

template <class T, class U>
T copysign (const T& x, const U& y);

template <class T>
calculated-result-type changesign (const T& z);

}} // namespaces
```

Description

```
template<class T>
int signbit(T x);
```

Returns a non-zero value if the sign bit is set in variable *x*, otherwise `0`.

Important

The return value from this function is zero or *not-zero* and **not** zero or one.

```
template <class T>
int sign (const T& z);
```

Returns `1` if *x* `> 0`, `-1` if *x* `< 0`, and `0` if *x* is zero.

```
template <class T, class U>
calculated-result-type copysign (const T& x, const U& y);
```

Sets the sign of *x* to be the same as the sign of *y*.

See C99 7.12.11.1 The copysign functions for more detail.

```
template <class T>
T changesign (const T& z);
```

Returns a floating-point number with a binary representation where the signbit is the opposite of the sign bit in *x*, and where the other bits are the same as in *x*.

This function is widely available, but not specified in any standards.

Rationale: Not specified by TR1, but `changesign(x)` is both easier to read and more efficient than

```
copysign(x, signbit(x) ? 1.0 : -1.0);
```

For finite values, this function has the same effect as simple negation, the assignment z = -z, but for nonfinite values, infinities and NaNs, the `changesign(x)` function may be the only portable way to ensure that the sign bit is changed.

Sign bits

One of the bits in the binary representation of a floating-point number gives the sign, and the remaining bits give the absolute value. That bit is known as the sign bit. The sign bit is set = 1 for negative numbers, and is not set = 0 for positive numbers. (This is true for all binary representations of floating-point numbers that are used by modern microprocessors.)

C++ TR1 specifies `copysign` functions and function templates for accessing the sign bit.

For user-defined types (UDT), the sign may be stored in some other way. They may also not provide infinity or NaNs. To use these functions with a UDT, it may be necessary to explicitly specialize them for UDT type T.

Examples

```
signbit(3.5) is zero (or false)
signbit(-7.1) is 1 (or true)
copysign(4.2, 7.9) is 4.2
copysign(3.5 -1.4) is -3.5
copysign(-4.2, 1.0) is 4.2
copysign(-8.6, -3.3) is -8.6
changesign(6.9) is -6.9
changesign(-1.8) is 1.8
```

Portability

The library supports the following binary floating-point formats:

- IEEE 754 single precision
- IEEE 754 double precision
- IEEE 754 extended double precision with 15 exponent bits
- Intel extended double precision
- PowerPC extended double precision
- Motorola 68K extended double precision

The library does not support the VAX floating-point formats. (These are available on VMS, but the default on VMS is the IEEE 754 floating-point format.)

The main portability issues are:

- Unsupported floating-point formats.
- The library depends on the header `boost/detail/endian.hpp` to detemine endianness.
- Code such as `#if defined(__ia64) || defined(__ia64__) || defined(_M_IA64)` is used to determine the processor type.

The library has passed all tests on the following platforms:

- Win32 / MSVC 7.1 / 10.0 / x86 32 and 64-bit, and later Win32
- Win32 / Intel C++ 7.1, 8.1, 9.1 / x86

- Mac OS X / GCC 3.3, 4.0 / ppc
- Linux / Intel C++ 9.1 / x86, ia64
- Linux / GCC 3.3 / x86, x64, ia64, ppc, hppa, mips, m68k
- Linux / GCC 3.4 / x64
- HP-UX / aCC, GCC 4.1 / ia64
- HP-UX / aCC / hppa
- Tru64 / Compaq C++ 7.1 / alpha
- VMS / HP C++ 7.1 / alpha (in IEEE floating-point mode)
- VMS / HP C++ 7.2 / ia64 (in IEEE floating-point mode)

Facets for Floating-Point Infinities and NaNs

Synopsis

```
namespace boost{ namespace math
{
  // Values for flags.
  const int legacy;
  const int signed_zero;
  const int trap_infinity;
  const int trap_nan;

  template<
      class CharType,
      class OutputIterator = std::ostreambuf_iterator<CharType>
  >
  class nonfinite_num_put : public std::num_put<CharType, OutputIterator>
  {
  public:
      explicit nonfinite_num_put(int flags = 0);
  };

  template<
      class CharType,
      class InputIterator = std::istreambuf_iterator<CharType>
  >
  class nonfinite_num_get : public std::num_get<CharType, InputIterator>
  {
  public:
      explicit nonfinite_num_get(int flags = 0);  // legacy, sign_zero ...
  };
}} // namespace boost namespace math
```

To use these facets

```
#include <boost\math\special_functions\nonfinite_num_facets.hpp>
```

Introduction

The Problem

The C++98 standard does not specify how *infinity* and *NaN* are represented in text streams. As a result, different platforms use different string representations. This can cause undefined behavior when text files are moved between different platforms. Some platforms cannot even input parse their own output! So 'route-tripping' or loopback of output to input is not possible. For instance, the following test fails with MSVC:

```
stringstream ss;
double inf = numeric_limits<double>::infinity();
double r;
ss << inf; // Write out.
ss >> r; // Read back in.

cout << "infinity output was " << inf << endl; // 1.#INF
cout << "infinity input was " << r << endl; // 1

assert(inf == y); // Fails!
```

The Solution

The facets `nonfinite_num_put` and `nonfinite_num_get` format and parse all floating-point numbers, including `infinity` and `NaN`, in a consistent and portable manner.

The following test succeeds with MSVC.

```
locale old_locale;
locale tmp_locale(old_locale, new nonfinite_num_put<char>);
locale new_locale(tmp_locale, new nonfinite_num_get<char>);
```

Tip

To add two facets, `nonfinite_num_put` and `nonfinite_num_get`, you may have to add one at a time, using a temporary locale.

Or you can create a new locale in one step

`std::locale new_locale(std::locale(std::locale(std::locale(), new boost::math::non-finite_num_put<char>), new boost::math::nonfinite_num_get<char>));`

and, for example, use it to imbue an input and output stringstream.

Tip

To just change an input or output stream, you can concisely write `cout.imbue (std::locale(std::locale(), new boost::math::nonfinite_num_put<char>));` or `cin.imbue (std::locale(std::locale(), new boost::math::nonfinite_num_get<char>));`

```
stringstream ss;
ss.imbue(new_locale);
double inf = numeric_limits<double>::infinity();
ss << inf; // Write out.
assert(ss.str() == "inf");
double r;
ss >> r; // Read back in.
assert(inf == r); // Confirms that the double values really are identical.

cout << "infinity output was " << ss.str() << endl;
cout << "infinity input was " << r << endl;
// But the string representation of r displayed will be the native type
// because, when it was constructed, cout had NOT been imbued
// with the new locale containing the nonfinite_numput facet.
// So the cout output will be "1.#INF on MS platforms
// and may be "inf" or other string representation on other platforms.
```

C++0X standard for output of infinity and NaN

C++0X (final) draft standard does not explicitly specify the representation (and input) of nonfinite values, leaving it implementation-defined. So without some specific action, input and output of nonfinite values is not portable.

C99 standard for output of infinity and NaN

The C99 standard **does** specify how infinity and NaN are formatted by printf and similar output functions, and parsed by scanf and similar input functions.

The following string representations are used:

Table 13. C99 Representation of Infinity and NaN

number	string
Positive infinity	"inf" or "infinity"
Positive NaN	"nan" or "nan(...)"
Negative infinity	"-inf" or "-infinity"
Negative NaN	"-nan" or "-nan(...)"

So following C99 provides a sensible 'standard' way of handling input and output of nonfinites in C++, and this implementation follows most of these formats.

Signaling NaNs

A particular type of NaN is the signaling NaN. The usual mechanism of signaling is by raising a floating-point exception. Signaling NaNs are defined by IEEE 754-2008.

Floating-point values with layout *s*111 1111 1*a*xx xxxx xxxx xxxx xxxx xxxx where *s* is the sign, *x* is the payload, and bit *a* determines the type of NaN.

If bit *a* = 1, it is a quiet NaN.

If bit *a* is zero and the payload *x* is nonzero, then it is a signaling NaN.

Although there has been theoretical interest in the ability of a signaling NaN to raise an exception, for example to prevent use of an uninitialised variable, in practice there appears to be no useful application of signaling NaNs for most current processors. C++0X

18.3.2.2 still specifies a (implementation-defined) representation for signaling NaN, and `static constexpr bool has_signaling_NaN` a method of checking if a floating-point type has a representation for signaling NaN.

But in practice, most platforms treat signaling NaNs in the same as quiet NaNs. So, for example, they are represented by "nan" on output in C99 format, and output as `1.#QNAN` by Microsoft compilers.

Note

The C99 standard does not distinguish between the quiet NaN and signaling NaN values. A quiet NaN propagates through almost every arithmetic operation without raising a floating-point exception; a signaling NaN generally raises a floating-point exception when occurring as an arithmetic operand.

C99 specification does not define the behavior of signaling NaNs. NaNs created by IEC 60559 operations are always quiet. Therefore this implementation follows C99, and treats the signaling NaN bit as just a part of the NaN payload field. So this implementation does not distinguish between the two classes of NaN.

Note

An implementation may give zero and non-numeric values (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in the C99 Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

This might apply to user-defined types, but in practice built-in floating-point types `float`, `double` and `long double` have well-behaved signs.

The numbers can be of type `float`, `double` and `long double`. An optional + sign can be used with positive numbers (controlled by ios manipulator `showpos`). The function `printf` and similar C++ functions use standard formatting flags to put all lower or all upper case (controlled by `std::ios` manipulator `uppercase` and `lowercase`).

The function `scanf` and similar input functions are case-insensitive.

The dots in `nan(...)` stand for an arbitrary string. The meaning of that string is implementation dependent. It can be used to convey extra information about the NaN, from the 'payload'. A particular value of the payload might be used to indicate a *missing value*, for example.

This library uses the string representations specified by the C99 standard.

An example of an implementation that optionally includes the NaN payload information is at AIX NaN fprintf. That implementation specifies for Binary Floating Point NANs:

- A NaN ordinal sequence is a left-parenthesis character '(', followed by a digit sequence representing an integer n, where 1 <= n <= INT_MAX-1, followed by a right-parenthesis character ')'.
- The integer value, n, is determined by the fraction bits of the NaN argument value as follows:
- For a signalling NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an even integer value, 2*n. Then formatted output functions produce a (signalling) NaN ordinal sequence corresponding to the integer value n.
- For a quiet NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an odd integer value, 2*n-1. Then formatted output functions produce a (quiet) NaN ordinal sequence corresponding to the integer value n.

Warning

This implementation does not (yet) provide output of, or access to, the NaN payload.

Reference

The Facet nonfinite_num_put

```
template<
  class CharType, class OutputIterator = std::ostreambuf_iterator<CharType>
        >
class nonfinite_num_put;
```

The class nonfinite_num_put<CharType, OutputIterator> is derived from std::num_put<CharType, OutputIterator>. Thus it is a facet that formats numbers. The first template argument is the character type of the formatted strings, usually char or wchar_t. The second template argument is the type of iterator used to write the strings. It is required to be an output iterator. Usually the default std::ostreambuf_iterator is used. The public interface of the class consists of a single constructor only:

```
nonfinite_num_put(int flags = 0);
```

The flags argument (effectively optional because a default of no_flags is provided) is discussed below. The class template nonfinite_num_put is defined in the header boost/math/nonfinite_num_facets.hpp and lives in the namespace boost::math.

Unlike the C++ Standard facet std::num_put, the facet nonfinite_num_put formats infinity and NaN in a consistent and portable manner. It uses the following string representations:

Number	String
Positive infinity	inf
Positive NaN	nan
Negative infinity	-inf
Negative NaN	-nan

The numbers can be of type float, double and long double. The strings can be in all lower case or all upper case. An optional + sign can be used with positive numbers. This can be controlled with the uppercase, lowercase, showpos and noshowpos manipulators. Formatting of integers, boolean values and finite floating-point numbers is simply delegated to the normal std::num_put.

Facet nonfinite_num_get

```
template<class CharType, class InputIterator = std::istreambuf_iterator<CharType> > class non↵
finite_num_get;
```

The class nonfinite_num_get<CharType, InputIterator> is derived from std::num_get<CharType, IntputIterator>. Thus it is a facet that parses strings that represent numbers. The first template argument is the character type of the strings, usually char or wchar_t. The second template argument is the type of iterator used to read the strings. It is required to be an input iterator. Usually the default is used. The public interface of the class consists of a single constructor only:

```
nonfinite_num_get(int flags = 0);
```

The flags argument is discussed below. The class template nonfinite_num_get is defined in the header boost/math/nonfinite_num_facets.hpp and lives in the namespace boost::math.

Unlike the facet `std::num_get`, the facet `nonfinite_num_get` parses strings that represent `infinity` and `NaN` in a consistent and portable manner. It recognizes precisely the string representations specified by the C99 standard:

Number	String
Positive infinity	inf, infinity
Positive NaN	nan, nan(...)
Negative infinity	-inf, -infinity
Negative NaN	-nan, -nan(...)

The numbers can be of type `float`, `double` and `long double`. The facet is case-insensitive. An optional + sign can be used with positive numbers. The dots in nan(...) stand for an arbitrary string usually containing the *NaN payload*. Parsing of strings that represent integers, boolean values and finite floating-point numbers is delegated to `std::num_get`.

When the facet parses a string that represents `infinity` on a platform that lacks infinity, then the fail bit of the stream is set.

When the facet parses a string that represents `NaN` on a platform that lacks NaN, then the fail bit of the stream is set.

Flags

The constructors for `nonfinite_num_put` and `nonfinite_num_get` take an optional bit flags argument. There are four different bit flags:

- legacy
- signed_zero
- trap_infinity
- trap_nan

The flags can be combined with the OR `operator|`.

The flags are defined in the header `boost/math/nonfinite_num_facets.hpp` and live in the `namespace boost::math`.

legacy

The legacy flag has no effect with the output facet `nonfinite_num_put`.

If the legacy flag is used with the `nonfinite_num_get` input facet, then the facet will recognize all the following string representations of `infinity` and `NaN`:

Number	String
Positive infinity	inf, infinity, one#inf
Positive NaN	nan, nan(...), nanq, nans, qnan, snan, one#ind, one#qnan, one#snan
Negative infinity	-inf, -infinity, -one#inf
Negative NaN	-nan, -nan(...), -nanq, -nans, -qnan, -snan, -one#ind, - one#qnan, -one#snan

- The numbers can be of type `float`, `double` and `long double`.

- The facet is case-insensitive.
- An optional + sign can be used with the positive values.
- The dots in `nan(...)` stand for an arbitrary string.
- `one` stands for any string that `std::num_get` parses as the number `1`, typically "1.#INF", "1.QNAN" but also "000001.#INF"...

The list includes a number of non-standard string representations of infinity and NaN that are used by various existing implementations of the C++ standard library, and also string representations used by other programming languages.

signed_zero

If the `signed_zero` flag is used with `nonfinite_num_put`, then the facet will always distinguish between positive and negative zero. It will format positive zero as "0" or "+0" and negative zero as "-0". The string representation of positive zero can be controlled with the `showpos` and `noshowpos` manipulators.

The `signed_zero flag` has no effect with the input facet `nonfinite_num_get`. The input facet `nonfinite_num_get` always parses "0" and "+0" as positive zero and "-0" as negative zero, as do most implementations of `std::num_get`.

Note

If the `signed_zero` flag is not set (the default), then a negative zero value will be displayed on output in whatever way the platform normally handles it. For most platforms, this it will format positive zero as "0" or "+0" and negative zero as "-0". But setting the `signed_zero` flag may be more portable.

Tip

A negative zero value can be portably produced using the changesign function `(changesign)(static_cast<ValType>(0))` where `ValType` is `float`, `double` or `long double`, or a User-Defined floating-point type (UDT) provided that this UDT has a sign and that the changesign function is implemented.

trap_infinity

If the `trap_infinity` flag is used with `nonfinite_num_put`, then the facet will throw an exception of type `std::ios_base::failure` when an attempt is made to format positive or negative infinity. If the facet is called from a stream insertion operator, then the stream will catch that exception and set either its `fail bit` or its `bad bit`. Which bit is set is platform dependent.

If the `trap_infinity` flag is used with `nonfinite_num_get`, then the facet will set the `fail bit` of the stream when an attempt is made to parse a string that represents positive or negative infinity.

(See Design Rationale below for a discussion of this inconsistency.)

trap_nan

Same as `trap_infinity`, but positive and negative NaN are trapped instead.

Examples

Simple example with std::stringstreams

```
locale old_locale;
locale tmp_locale(old_locale, new nonfinite_num_put<char>);
locale new_locale(tmp_locale, new nonfinite_num_get<char>);
```

```
stringstream ss;
ss.imbue(new_locale);
double inf = numeric_limits<double>::infinity();
ss << inf; // Write out.
assert(ss.str() == "inf");
double r;
ss >> r; // Read back in.
assert(inf == r); // Confirms that the double values really are identical.

cout << "infinity output was " << ss.str() << endl;
cout << "infinity input was " << r << endl;
// But the string representation of r displayed will be the native type
// because, when it was constructed, cout had NOT been imbued
// with the new locale containing the nonfinite_numput facet.
// So the cout output will be "1.#INF on MS platforms
// and may be "inf" or other string representation on other platforms.
```

Use with lexical_cast

Note

From Boost 1.48, lexical_cast no longer uses stringstreams internally, and is now able to handle infinities and NaNs natively on most platforms.

Without using a new locale that contains the nonfinite facets, previous versions of `lexical_cast` using stringstream were not portable (and often failed) if nonfinite values are found.

```
locale old_locale;
locale tmp_locale(old_locale, new nonfinite_num_put<char>);
locale new_locale(tmp_locale, new nonfinite_num_get<char>);
```

Although other examples imbue individual streams with the new locale, for the streams constructed inside lexical_cast, it was necesary to assign to a global locale.

```
locale::global(new_locale);
```

`lexical_cast` then works as expected, even with infinity and NaNs.

```
double x = boost::lexical_cast<double>("inf");
assert(x == std::numeric:limits<double>::infinity());

string s = boost::lexical_cast<string>(numeric_limits<double>::infinity());
assert(s == "inf");
```

Warning

If you use stringstream inside your functions, you may still need to use a global locale to handle nonfinites correctly. Or you need to imbue your stringstream with suitable get and put facets.

Warning

You should be aware that the C++ specification does not explicitly require that input from decimal digits strings converts with rounding to the nearest representable floating-point binary value. (In contrast, decimal digits read by the compiler, for example by an assignment like `double d = 1.234567890123456789`, are guaranteed to assign the nearest representable value to double d). This implies that, no matter how many decimal digits you provide, there is a potential uncertainty of 1 least significant bit in the resulting binary value.

See conversion and rounding for more information on *nearest representable* and *rounding* and Exploring Binary for much detail on input and round-tripping difficulties.

Most iostream libraries do in fact achieve the desirable *nearest representable floating-point binary value* for all values of input. However one popular STL library does not quite achieve this for 64-bit doubles. See Decimal digit string input to double may be 1 bit wrong for the bizarre full details.

If you are expecting to 'round-trip' `lexical_cast` or `serialization`, for example archiving and loading, and want to be **absolutely certain that you will always get an exactly identical double value binary pattern**, you should use the suggested 'workaround' below that is believed to work on all platforms.

You should output using all potentially significant decimal digits, by setting stream precision to `std::numeric_limits<double>::max_digits10`, (or for the appropriate floating-point type, if not double) and crucially, **require `scientific` format**, not `fixed` or automatic (default), for example:

```
double output_value = any value;
std::stringstream s;
s << setprecison(std::numeric_limits<double>::max_digits10) << scientific << output_value;
s >> input_value;
```

Use with serialization archives

It is vital that the same locale is used when an archive is saved and when it is loaded. Otherwise, loading the archive may fail. By default, archives are saved and loaded with a classic C locale with a `boost::archive::codecvt_null` facet added. Normally you do not have to worry about that.

The constructors for the archive classes, as a side-effect, imbue the stream with such a locale. However, if you want to use the facets `nonfinite_num_put` and `nonfinite_num_get` with archives, then you have to manage the locale manually. That is done by calling the archive constructor with the flag `boost::archive::no_codecvt`, thereby ensuring that the archive constructor will **not imbue the stream with a new locale**.

The following code shows how to use `nonfinite_num_put` with a `text_oarchive`.

```
locale default_locale(locale::classic(), new boost::archive::codecvt_null<char>);
locale my_locale(default_locale, new nonfinite_num_put<char>);

ofstream ofs("test.txt");
ofs.imbue(my_locale);

boost::archive::text_oarchive oa(ofs, no_codecvt);

double x = numeric_limits<double>::infinity();
oa & x;
```

The same method works with `nonfinite_num_get` and `text_iarchive`.

If you use the `nonfinite_num_put` with `trap_infinity` and/or `trap_nan` flag with a serialization archive, then you must set the exception mask of the stream. Serialization archives do not check the stream state.

Other examples

nonfinite_facet_simple.cpp give some more simple demonstrations of the difference between using classic C locale and constructing a C99 infinty and NaN compliant locale for input and output.

See nonfinite_facet_sstream.cpp for this example of use with `std::stringstreams`.

For an example of how to enforce the MSVC 'legacy' "1.#INF" and "1.#QNAN" representations of infinity and NaNs, for input and output, see nonfinite_legacy.cpp.

Treatment of signaling NaN is demonstrated at ../../example/nonfinite_signaling_NaN.cpp

Example ../../example/nonfinite_loopback_ok.cpp shows loopback works OK.

Example ../../example/nonfinite_num_facet.cpp shows output and re-input of various finite and nonfinite values.

A simple example of trapping nonfinite output is at nonfinite_num_facet_trap.cpp.

A very basic example of using Boost.Archive is at ../../example/nonfinite_serialization_archives.cpp.

A full demonstration of serialization by Francois Mauger is at ../../example/nonfinite_num_facet_serialization.cpp

Portability

This library uses the floating-point number classification and sign-bit from Boost.Math library, and should work on all platforms where that library works. See the portability information for that library.

Design Rationale

- The flags are implemented as a const data member of the facet. Facets are reference counted, and locales can share facets. Therefore changing the flags of a facet would have effects that are hard to predict. An alternative design would be to implement the flags using `std::ios_base::xalloc` and `std::ios_base::iword`. Then one could safely modify the flags, and one could define manipulators that do so. However, for that to work with dynamically linked libraries, a `.cpp` file would have to be added to the library. It was judged be more desirable to have a header-only library, than to have mutable flags and manipulators.

- The facet `nonfinite_num_put` throws an exception when the `trap_infinity` or `trap_nan` flag is set and an attempt is made to format infinity or NaN. It would be better if the facet set the `fail bit` of the stream. However, facets derived from `std::num_put` do not have access to the stream state.

Floating-Point Representation Distance (ULP), and Finding Adjacent Floating-Point Values

Unit of Least Precision or Unit in the Last Place is the gap between two different, but as close as possible, floating-point numbers.

Most decimal values, for example 0.1, cannot be exactly represented as floating-point values, but will be stored as the closest representable floating-point.

Functions are provided for finding adjacent greater and lesser floating-point values, and estimating the number of gaps between any two floating-point values.

The floating-point type (FPT) must have has a fixed number of bits in the representation. The number of bits may set at runtime, but must be the same for all numbers. For example, NTL::quad_float type (fixed 128-bit representation), NTL::RR type (arbitrary but fixed decimal digits, default 150) or Boost.Multiprecision cpp_dec_float andcpp_bin_float are fixed at runtime, but **not** a type that extends the representation to provide an exact representation for any number, for example XRC eXact Real in C.

The accuracy of mathematical functions can be assessed and displayed in terms of Unit in the Last Place, often as a ulps plot or by binning the differences as a histogram. Samples are evaluated using the implementation under test and compared with 'known good' representation obtained using a more accurate method. Other implementations, often using arbitrary precision arithmetic, for example Wolfram Alpha are one source of references values. The other method, used widely in Boost.Math special functions, it to carry out the same algorithm, but using a higher precision type, typically using Boost.Multiprecision types like `cpp_bin_float_quad` for 128-bit (about 35 decimal digit precision), or `cpp_bin_float_50` (for 50 decimal digit precision).

When converted to a particular machine representation, say `double`, say using a `static_cast`, the value is the nearest representation possible for the `double` type. This value cannot be 'wrong' by more than half a Unit in the last place (ULP), and can be obtained using the Boost.Math function `ulp`. (Unless the algorithm is fundamentally flawed, something that should be revealed by 'sanity' checks using some independent sources).

See some discussion and example plots by Cleve Moler of Mathworks ulps plots reveal math-function accuracy.

Finding the Next Representable Value in a Specific Direction (nextafter)

Synopsis

```
#include <boost/math/special_functions/next.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT nextafter(FPT val, FPT direction);

}} // namespaces
```

Description - nextafter

This is an implementation of the `nextafter` function included in the C99 standard. (It is also effectively an implementation of the C99 `nexttoward` legacy function which differs only having a `long double` direction, and can generally serve in its place if required).

Note

The C99 functions must use suffixes f and l to distinguish `float` and `long double` versions. C++ uses the template mechanism instead.

Returns the next representable value after *x* in the direction of *y*. If `x == y` then returns *x*. If *x* is non-finite then returns the result of a domain_error. If there is no such value in the direction of *y* then returns an overflow_error.

Warning

The template parameter FTP must be a floating-point type. An integer type, for example, will produce an unhelpful error message.

Tip

Nearly always, you just want the next or prior representable value, so instead use `float_next` or `float_prior` below.

Examples - nextafter

The two representations using a 32-bit float either side of unity are:

```
The nearest (exact) representation of 1.F is      1.00000000
nextafter(1.F, 999) is                            1.00000012
nextafter(1/f, -999) is                           0.99999994

The nearest (not exact) representation of 0.1F is 0.100000001
nextafter(0.1F, 10) is                            0.100000009
nextafter(0.1F, 10) is                            0.099999994
```

Finding the Next Greater Representable Value (float_next)

Synopsis

```
#include <boost/math/special_functions/next.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT float_next(FPT val);

}} // namespaces
```

Description - float_next

Returns the next representable value which is greater than *x*. If *x* is non-finite then returns the result of a domain_error. If there is no such value greater than *x* then returns an overflow_error.

Has the same effect as

```
nextafter(val, (std::numeric_limits<FPT>::max)());
```

Finding the Next Smaller Representable Value (float_prior)

Synopsis

```
#include <boost/math/special_functions/next.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT float_prior(FPT val);

}} // namespaces
```

Description - float_prior

Returns the next representable value which is less than *x*. If *x* is non-finite then returns the result of a domain_error. If there is no such value less than *x* then returns an overflow_error.

Has the same effect as

```
nextafter(val, -(std::numeric_limits<FPT>::max)());  // Note most negative value -max.
```

Calculating the Representation Distance Between Two floating-point Values (ULP) float_distance

Function float_distance finds the number of gaps/bits/ULP between any two floating-point values. If the significands of floating-point numbers are viewed as integers, then their difference is the number of ULP/gaps/bits different.

Synopsis

```
#include <boost/math/special_functions/next.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT float_distance(FPT a, FPT b);

}} // namespaces
```

Description - float_distance

Returns the distance between *a* and *b*: the result is always a signed integer value (stored in floating-point type FPT) representing the number of distinct representations between *a* and *b*.

Note that

- `float_distance(a, a)` always returns 0.
- `float_distance(float_next(a), a)` always returns -1.
- `float_distance(float_prior(a), a)` always returns 1.

The function `float_distance` is equivalent to calculating the number of ULP (Units in the Last Place) between *a* and *b* except that it returns a signed value indicating whether `a > b` or not.

If the distance is too great then it may not be able to be represented as an exact integer by type FPT, but in practice this is unlikely to be a issue.

Advancing a floating-point Value by a Specific Representation Distance (ULP) float_advance

Function `float_advance` advances a floating-point number by a specified number of ULP.

Synopsis

```
#include <boost/math/special_functions/next.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT float_advance(FPT val, int distance);

}} // namespaces
```

Description - float_advance

Returns a floating-point number *r* such that `float_distance(val, r) == distance`.

Obtaining the Size of a Unit In the Last Place - ULP

Function `ulp` gives the size of a unit-in-the-last-place for a specified floating-point value.

Synopsis

```
#include <boost/math/special_functions/ulp.hpp>
```

```
namespace boost{ namespace math{

template <class FPT>
FPT ulp(const FPT& x);

template <class FPT, class Policy>
FPT ulp(const FPT& x, const Policy&);

}} // namespaces
```

Description - ulp

Returns one unit in the last place of *x*.

Corner cases are handled as follows:

- If the argument is a NaN, then raises a domain_error.
- If the argument is an infinity, then raises an overflow_error.

- If the argument is zero then returns the smallest representable value: for example for type `double` this would be either `std::numeric_limits<double>::min()` or `std::numeric_limits<double>::denorm_min()` depending whether denormals are supported (which have the values `2.2250738585072014e-308` and `4.9406564584124654e-324` respectively).
- If the result is too small to represent, then returns the smallest representable value.
- Always returns a positive value such that `ulp(x) == ulp(-x)`.

Important: The behavior of this function is aligned to that of Java's ulp function, please note however that this function should only ever be used for rough and ready calculations as there are enough corner cases to trap even careful programmers. In particular:

- The function is asymetrical, which is to say, given `u = ulp(x)` if `x > 0` then `x + u` is the next floating-point value, but `x - u` is not necessarily the previous value. Similarly, if `x < 0` then `x - u` is the previous floating-point value, but `x + u` is not necessarily the next value. The corner cases occur at power of 2 boundaries.
- When the argument becomes very small, it may be that there is no floating-point value that represents one ULP. Whether this is the case or not depends not only on whether the hardware may *sometimes* support denormals (as signalled by `std::numeric_limits<FPT>::has_denorm`), but also whether these are currently enabled at runtime (for example on SSE hardware, the DAZ or FTZ flags will disable denormal support). In this situation, the `ulp` function may return a value that is many orders of magnitude too large.

In light of the issues above, we recomend that:

- To move between adjacent floating-point values always use float_next, float_prior or nextafter (`std::nextafter` is another candidate, but our experience is that this also often breaks depending which optimizations and hardware flags are in effect).
- To move several floating-point values away use float_advance.
- To calculate the edit distance between two floats use Boost.Math float_distance.

There is none the less, one important use case for this function:

If it is known that the true result of some function is x_t and the calculated result is x_c, then the error measured in ulp is simply `fabs(`x_t` - `x_c`) / ulp(`x_t`)`.

Floating-point Comparison

Comparison of floating-point values has always been a source of endless difficulty and confusion.

Unlike integral values that are exact, all floating-point operations will potentially produce an inexact result that will be rounded to the nearest available binary representation. Even apparently inocuous operations such as assigning 0.1 to a double produces an inexact result (as this decimal number has no exact binary representation).

Floating-point computations also involve rounding so that some 'computational noise' is added, and hence results are also not exact (although repeatable, at least under identical platforms and compile options).

Sadly, this conflicts with the expectation of most users, as many articles and innumerable cries for help show all too well.

Some background reading is:

- Knuth D.E. The art of computer programming, vol II, section 4.2, especially Floating-Point Comparison 4.2.2, pages 198-220.
- David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic"
- Alberto Squassabia, Comparing floats listing
- Google Floating-Point_Comparison guide
- Boost.Test Floating-Point_Comparison

Boost provides a number of ways to compare floating-point values to see if they are tolerably close enough to each other, but first we must decide what kind of comparison we require:

- Absolute difference/error: the absolute difference between two values *a* and *b* is simply `fabs(a-b)`. This is the only meaningful comparison to make if we know that the result may have cancellation error (see below).
- The edit distance between the two values: i.e. how many (binary) floating-point values are between two values *a* and *b*? This is provided by the function Boost.Math float_distance, but is probably only useful when you know that the distance should be very small. This function is somewhat difficult to compute, and doesn't scale to values that are very far apart. In other words, use with care.
- The relative distance/error between two values. This is quick and easy to compute, and is generally the method of choice when checking that your results are "tolerably close" to one another. However, it is not as exact as the edit distance when dealing with small differences, and due to the way floating-point values are encoded can "wobble" by a factor of 2 compared to the "true" edit distance. This is the method documented below: if `float_distance` is a surgeon's scalpel, then `relative_difference` is more like a Swiss army knife: both have important but different use cases.

Relative Comparison of Floating-point Values

`#include <boost/math/special_functions/relative_difference.hpp>`

```
template <class T, class U>
calculated-result-type relative_difference(T a, U b);

template <class T, class U>
calculated-result-type epsilon_difference(T a, U b);
```

The function `relative_difference` returns the relative distance/error *E* between two values as defined by:

```
E = fabs((a - b) / min(a,b))
```

The function `epsilon_difference` is a convenience function that returns `relative_difference(a, b) / eps` where `eps` is the machine epsilon for the result type.

The following special cases are handled as follows:

- If either of *a* or *b* is a NaN, then returns the largest representable value for T: for example for type `double`, this is `std::numeric_limits<double>::max()` which is the same as `DBL_MAX` or `1.7976931348623157e+308`.
- If *a* and *b* differ in sign then returns the largest representable value for T.
- If both *a* and *b* are both infinities (of the same sign), then returns zero.
- If just one of *a* and *b* is an infinity, then returns the largest representable value for T.
- If both *a* and *b* are zero then returns zero.
- If just one of *a* or *b* is a zero or a denormalized value, then it is treated as if it were the smallest (non-denormalized) value representable in T for the purposes of the above calculation.

These rules were primarily designed to assist with our own test suite, they are designed to be robust enough that the function can in most cases be used blindly, including in cases where the expected result is actually too small to represent in type T and underflows to zero.

Examples

Some using statements will ensure that the functions we need are accessible.

```
using namespace boost::math;
```

or

```
using boost::math::relative_difference;
using boost::math::epsilon_difference;
using boost::math::float_next;
using boost::math::float_prior;
```

The following examples display values with all possibly significant digits. Newer compilers should provide `std::numeric_limits<FPT>::max_digits10` for this purpose, and here we use `float` precision where `max_digits10` = 9 to avoid displaying a distracting number of decimal digits.

Note

Older compilers can use this formula to calculate `max_digits10` from `std::numeric_limits<FPT>::digits10`: `int max_digits10 = 2 + std::numeric_limits<FPT>::digits10 * 3010/10000;`

One can set the display including all trailing zeros (helpful for this example to show all potentially significant digits), and also to display `bool` values as words rather than integers:

```
std::cout.precision(std::numeric_limits<float>::max_digits10);
std::cout << std::boolalpha << std::showpoint << std::endl;
```

When comparing values that are *quite close* or *approximately equal*, we could use either `float_distance` or `relative_difference/epsilon_difference`, for example with type `float`, these two values are adjacent to each other:

```
float a = 1;
float b = 1 + std::numeric_limits<float>::epsilon();
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
std::cout << "float_distance = " << float_distance(a, b) << std::endl;
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(a, b) << std::endl;
```

Which produces the output:

```
a = 1.00000000
b = 1.00000012
float_distance = 1.00000000
relative_difference = 1.19209290e-007
epsilon_difference = 1.00000000
```

In the example above, it just so happens that the edit distance as measured by `float_distance`, and the difference measured in units of epsilon were equal. However, due to the way floating point values are represented, that is not always the case:

```
a = 2.0f / 3.0f;   // 2/3 inexactly represented as a float
b = float_next(float_next(float_next(a))); // 3 floating point values above a
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
std::cout << "float_distance = " << float_distance(a, b) << std::endl;
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(a, b) << std::endl;
```

Which produces the output:

```
a = 0.666666687
b = 0.666666865
float_distance = 3.00000000
relative_difference = 2.68220901e-007
epsilon_difference = 2.25000000
```

There is another important difference between `float_distance` and the `relative_difference/epsilon_difference` functions in that `float_distance` returns a signed result that reflects which argument is larger in magnitude, where as `relative_difference/epsilon_difference` simply return an unsigned value that represents how far apart the values are. For example if we swap the order of the arguments:

```
std::cout << "float_distance = " << float_distance(b, a) << std::endl;
std::cout << "relative_difference = " << relative_difference(b, a) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(b, a) << std::endl;
```

The output is now:

```
float_distance = -3.00000000
relative_difference = 2.68220901e-007
epsilon_difference = 2.25000000
```

Zeros are always treated as equal, as are infinities as long as they have the same sign:

```
a = 0;
b = -0;  // signed zero
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
a = b = std::numeric_limits<float>::infinity();
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "relative_difference = " << relative_difference(a, -b) << std::endl;
```

Which produces the output:

```
relative_difference = 0.000000000
relative_difference = 0.000000000
relative_difference = 3.40282347e+038
```

Note that finite values are always infinitely far away from infinities even if those finite values are very large:

```
a = (std::numeric_limits<float>::max)();
b = std::numeric_limits<float>::infinity();
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(a, b) << std::endl;
```

Which produces the output:

```
a = 3.40282347e+038
b = 1.#INF0000
relative_difference = 3.40282347e+038
epsilon_difference = 3.40282347e+038
```

Finally, all denormalized values and zeros are treated as being effectively equal:

```
a = std::numeric_limits<float>::denorm_min();
b = a * 2;
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
std::cout << "float_distance = " << float_distance(a, b) << std::endl;
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(a, b) << std::endl;
a = 0;
std::cout << "a = " << a << std::endl;
std::cout << "b = " << b << std::endl;
std::cout << "float_distance = " << float_distance(a, b) << std::endl;
std::cout << "relative_difference = " << relative_difference(a, b) << std::endl;
std::cout << "epsilon_difference = " << epsilon_difference(a, b) << std::endl;
```

Which produces the output:

```
a = 1.40129846e-045
b = 2.80259693e-045
float_distance = 1.00000000
relative_difference = 0.000000000
epsilon_difference = 0.000000000
a = 0.000000000
b = 2.80259693e-045
float_distance = 2.00000000
relative_difference = 0.000000000
epsilon_difference = 0.000000000
```

Notice how, in the above example, two denormalized values that are a factor of 2 apart are none the less only one representation apart!

All the above examples are contained in float_comparison_example.cpp.

Handling Absolute Errors

Imagine we're testing the following function:

```
double myspecial(double x)
{
   return sin(x) - sin(4 * x);
}
```

This function has multiple roots, some of which are quite predicable in that both `sin(x)` and `sin(4x)` are zero together. Others occur because the values returned from those two functions precisely cancel out. At such points the relative difference between the true value of the function and the actual value returned may be *arbitrarily large* due to cancellation error.

In such a case, testing the function above by requiring that the values returned by `relative_error` or `epsilon_error` are below some threshold is pointless: the best we can do is to verify that the *absolute difference* between the true and calculated values is below some threshold.

Of course, determining what that threshold should be is often tricky, but a good starting point would be machine epsilon multiplied by the largest of the values being summed. In the example above, the largest value returned by `sin(whatever)` is 1, so simply using machine epsilon as the target for maximum absolute difference might be a good start (though in practice we may need a slightly higher value - some trial and error will be necessary).

Condition Numbers

Synopsis

```
#include <boost/math/tools/condition_numbers.hpp>


namespace boost::math::tools {

template<class Real, bool kahan=true>
class summation_condition_number {
public:
    summation_condition_number(Real const x = 0);

    void operator+=(Real const & x);

    inline void operator-=(Real const & x);

    [[nodiscard]] Real operator()() const;

    [[nodiscard]] Real sum() const;

    [[nodiscard]] Real l1_norm() const;
};

template<class F, class Real>
auto evaluation_condition_number(F const & f, Real const & x);

} // namespaces
```

Summation Condition Number

Here we compute the condition number of the alternating harmonic sum:

```
using boost::math::tools::summation_condition_number;
auto cond = summation_condition_number<float, /* kahan = */ false>();
float max_n = 10000000;
for (float n = 1; n < max_n; n += 2)
{
    cond += 1/n;
    cond -= 1/(n+1);
}
std::cout << std::setprecision(std::numeric_limits<float>::digits10);
std::cout << "ln(2) = " << boost::math::constants::ln_two<float>() << "\n";
std::cout << "Sum   = " << cond.sum() << "\n";
std::cout << "Condition number = " << cond() << "\n";
```

Output:

```
ln(2) = 0.693147
Sum   = 0.693137
Condition number = 22.22282
```

We see that we have lost roughly two digits of accuracy, consistent with the heuristic that if the condition number is 10^k, then we lose k significant digits in the sum.

Our guess it that if you're worried about whether your sum is ill-conditioned, the *last* thing you want is for your condition number estimate to be inaccurate as well. Since the condition number estimate relies on computing the (perhaps ill-conditioned) sum, we have defaulted the accumulation to use Kahan summation:

```
auto cond = boost::math::tools::summation_condition_number<float>(); // will use Kahan summation.
// ...
```

Output:

```
ln(2)      = 0.693147
Kahan sum = 0.693147
Condition number = 22.2228
```

If you are interested, the L_1 norm is also generated by this computation, so you may query it if you like:

```
float l1 = cond.l1_norm();
// l1 = 15.4
```

Condition Number of Function Evaluation

The condition number of function evaluation is defined as the absolute value of $xf'(x)f(x)^{-1}$. It is computed as follows:

```
using boost::math::tools::evaluation_condition_number;
auto f = [](double x)->double { return std::log(x); };
double x = 1.125;
double cond = evaluation_condition_number(f, 1.125);
// cond = 1/log(x)
```

Caveats

The condition number of function evaluation gives us a measure of how sensitive our function is to roundoff error. Unfortunately, evaluating the condition number requires evaluating the function and its derivative, and this calculation is itself inaccurate whenever the condition number of function evaluation is large. Sadly, this is also the regime when you are most interested in evaluating a condition number!

This seems to be a fundamental problem. However, it should not be necessary to evaluate the condition number to high accuracy, valuable insights can be obtained simply by looking at the change in condition number as the function evolves over its domain.

References

- Gautschi, Walter. *Orthogonal polynomials: computation and approximation* Oxford University Press on Demand, 2004.
- Higham, Nicholas J. *The accuracy of floating point summation.* SIAM Journal on Scientific Computing 14.4 (1993): 783-799.
- Higham, Nicholas J. *Accuracy and stability of numerical algorithms.* Vol. 80. Siam, 2002.

Specified-width floating-point typedefs

Overview

The header `<boost/cstdfloat.hpp>` provides **optional** standardized floating-point `typedefs` having **specified widths**. These are useful for writing portable code because they should behave identically on all platforms. These `typedefs` are the floating-point analog of specified-width integers in `<cstdint>` and `stdint.h`.

The `typedefs` are based on N3626 proposed for a new C++14 standard header `<cstdfloat>` and N1703 proposed for a new C language standard header `<stdfloat.h>`.

All `typedefs` are in `namespace boost` (would be in namespace `std` if eventually standardized).

The `typedefs` include `float16_t, float32_t, float64_t, float80_t, float128_t`, their corresponding least and fast types, and the corresponding maximum-width type. The `typedefs` are based on underlying built-in types such as `float`, `double`, or `long double`, or the proposed N2016 short float type, or based on other compiler-specific non-standardized types such as `__float128`. The underlying types of these `typedefs` must conform with the corresponding specifications of binary16, binary32, binary64, and binary128 in IEEE_floating_point floating-point format, and `std::numeric_limits<>::is_iec559 == true`.

The 128-bit floating-point type (of great interest in scientific and numeric programming) is not required in the Boost header, and may not be supplied for all platforms/compilers, because compiler support for a 128-bit floating-point type is not mandated by either the C standard or the C++ standard.

If 128-bit floating-point is supported, then including `boost/cstdfloat.hpp` provides a **native** 128-bit type, and includes other headers in folder `boost/math/cstdfloat` that provide C++ quad support for C math functions in `<cmath>`, `<limits>`, `<iostream>`, `<complex>`, and the available floating-point types.

One can also, more robustly, include `boost/multiprecision/float128.hpp` and this provides a thin wrapper selecting the appropriate 128-bit native type from `cstdfloat` if available, or else a 128-bit multiprecision type.

See Jahnke-Emden-Lambda function example for an example using both a `<cmath>` function and a Boost.Math function to evaluate a moderately interesting function, the Jahnke-Emden-Lambda function and normal distribution as an example of a statistical distribution from Boost.Math.

Rationale

The implementation of `<boost/cstdfloat.hpp>` is designed to utilize `<float.h>`, defined in the 1989 C standard. The preprocessor is used to query certain preprocessor definitions in `<float.h>` such as FLT_MAX, DBL_MAX, etc. Based on the results of these queries, an attempt is made to automatically detect the presence of built-in floating-point types having specified widths. An unequivocal test requiring conformance with IEEE_floating_point (IEC599) based on `std::numeric_limits<>::is_iec559` is performed with `BOOST_STATIC_ASSERT`.

In addition, this Boost implementation `<boost/cstdfloat.hpp>` supports an 80-bit floating-point `typedef` if it can be detected, and a 128-bit floating-point `typedef` if it can be detected, provided that the underlying types conform with IEEE-754 precision extension (provided `std::numeric_limits<>::is_iec559 == true` for this type).

The header `<boost/cstdfloat.hpp>` makes the standardized floating-point `typedef`s safely available in `namespace boost` without placing any names in `namespace std`. The intention is to complement rather than compete with a potential future C/C++ Standard Library that may contain these `typedef`s. Should some future C/C++ standard include `<stdfloat.h>` and `<cstdfloat>`, then `<boost/cstdfloat.hpp>` will continue to function, but will become redundant and may be safely deprecated.

Because `<boost/cstdfloat.hpp>` is a Boost header, its name conforms to the boost header naming conventions, not the C++ Standard Library header naming conventions.

Note

`<boost/cstdfloat.hpp>` **cannot synthesize or create a `typedef` if the underlying type is not provided by the compiler**. For example, if a compiler does not have an underlying floating-point type with 128 bits (highly sought-after in scientific and numeric programming), then `float128_t` and its corresponding least and fast types are **not** provided by `<boost/cstdfloat.hpp>`.

Warning

If `<boost/cstdfloat.hpp>` uses a compiler-specific non-standardized type (**not** derived from `float, double`, or `long double`) for one or more of its floating-point `typedef`s, then there is no guarantee that specializations of `numeric_limits<>` will be available for these types. Typically, specializations of `numeric_limits<>` will only be available for these types if the compiler itself supports corresponding specializations for the underlying type(s), exceptions are GCC's `__float128` type and Intel's `_Quad` type which are explicitly supported via our own code.

Warning

As an implementation artifact, certain C macro names from `<float.h>` may possibly be visible to users of `<boost/cstdfloat.hpp>`. Don't rely on using these macros; they are not part of any Boost-specified interface. Use `std::numeric_limits<>` for floating-point ranges, etc. instead.

Tip

For best results, `<boost/cstdfloat.hpp>` should be `#included` before other headers that define generic code making use of standard library functions defined in <cmath>.

This is because `<boost/cstdfloat.hpp>` may define overloads of standard library functions where a non-standard type (i.e. other than `float`, `double`, or `long double`) is used for one of the specified width types. If generic code (for example in another Boost.Math header) calls a standard library function, then the correct overload will only be found if these overloads are defined prior to the point of use. See overloading template functions with float128_t and the implementation of `cstdfloat.hpp` for more details.

For this reason, making `#include <boost/cstdfloat.hpp>` the **first include** is usually best.

Exact-Width Floating-Point `typedef`s

The `typedef float#_t`, with # replaced by the width, designates a floating-point type of exactly # bits. For example `float32_t` denotes a single-precision floating-point type with approximately 7 decimal digits of precision (equivalent to binary32 in IEEE_floating_point).

Floating-point types in C and C++ are specified to be allowed to have (optionally) implementation-specific widths and formats. However, if a platform supports underlying floating-point types (conformant with IEEE_floating_point) with widths of 16, 32, 64, 80, 128 bits, or any combination thereof, then `<boost/cstdfloat.hpp>` does provide the corresponding `typedefs` `float16_t, float32_t, float64_t, float80_t, float128_t,` their corresponding least and fast types, and the corresponding maximum-width type.

How to tell which widths are supported

The definition (or not) of a floating-point constant macro is a way to test if a **specific width floating-point** is available on a platform.

```
#if defined(BOOST_FLOAT16_C)
// Can use boost::float16_t, perhaps a proposed __short_float.
// P0192R1, Adding Fundamental Type for Short Float,
// Boris Fomitchev, Sergei Nikolaev, Olivier Giroux, Lawrence Crowl, 2016 Feb14
// http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2016.pdf
#endif

#if defined(BOOST_FLOAT32_C)
// Can use boost::float32_t, usually type `float`.
#endif

#if defined(BOOST_FLOAT64_C)
// Can use boost::float64_t, usually type `double`, and sometimes also type `long double`.
#endif

#if defined(BOOST_FLOAT80_C)
// Can use boost::float80_t, sometimes type `long double`.
#endif

#if defined(BOOST_FLOAT128_C)
// Can use boost::float128_t. Sometimes type `__float128` or `_Quad`.
#endif
```

This can be used to write code which will compile and run (albeit differently) on several platforms. Without these tests, if a width, say `float128_t` is not supported, then compilation would fail. (It is, of course, rare for `float64_t` or `float32_t` not to be supported).

The number of bits in just the significand can be determined using:

```
std::numeric_limits<boost::floatmax_t>::digits
```

and from this one can safely infer the total number of bits because the type must be IEEE754 format, `std::numeric_limits<boost::floatmax_t>::is_iec559 == true`, so, for example, if `std::numeric_limits<boost::floatmax_t>::digits == 113`, then `floatmax_t` must be `float128_t`.

The **total** number of bits using `floatmax_t` can be found thus:

```
const int fpbits =
  (std::numeric_limits<boost::floatmax_t>::digits == 113) ? 128 :
  (std::numeric_limits<boost::floatmax_t>::digits == 64) ? 80 :
  (std::numeric_limits<boost::floatmax_t>::digits == 53) ? 64 :
  (std::numeric_limits<boost::floatmax_t>::digits == 24) ? 32 :
  (std::numeric_limits<boost::floatmax_t>::digits == 11) ? 16 :
  0; // Unknown - not IEEE754 format.
 std::cout << fpbits << " bits." << std::endl;
```

and the number of 'guaranteed' decimal digits using

```
std::numeric_limits<boost::floatmax_t>::digits10
```

and the maximum number of possibly significant decimal digits using

```
std::numeric_limits<boost::floatmax_t>::max_digits10
```

Tip

`max_digits10` is not always supported, but can be calculated at compile-time using the Kahan formula, `2 + binary_digits * 0.3010` which can be calculated **at compile time** using `2 + binary_digits * 3010/10000`.

Note

One could test that

std::is_same<boost::floatmax_t, boost::float128_t>::value == true

but this would fail to compile on a platform where `boost::float128_t` is not defined. So it is better to use the MACROs `BOOST_FLOATnnn_C`.

Minimum-width floating-point `typedef`s

The `typedef float_least#_t`, with # replaced by the width, designates a floating-point type with a **width of at least # bits**, such that no floating-point type with lesser size has at least the specified width. Thus, `float_least32_t` denotes the smallest floating-point type with a width of at least 32 bits.

Minimum-width floating-point types are provided for all existing exact-width floating-point types on a given platform.

For example, if a platform supports `float32_t` and `float64_t`, then `float_least32_t` and `float_least64_t` will also be supported, etc.

Fastest floating-point `typedef`s

The `typedef float_fast#_t`, with # replaced by the width, designates the **fastest** floating-point type with a **width of at least # bits**.

There is no absolute guarantee that these types are the fastest for all purposes. In any case, however, they satisfy the precision and width requirements.

Fastest minimum-width floating-point types are provided for all existing exact-width floating-point types on a given platform.

For example, if a platform supports `float32_t` and `float64_t`, then `float_fast32_t` and `float_fast64_t` will also be supported, etc.

Greatest-width floating-point typedef

The `typedef floatmax_t` designates a floating-point type capable of representing any value of any floating-point type in a given platform most precisely.

The greatest-width `typedef` is provided for all platforms, but, of course, the size may vary.

To provide floating-point **constants** most precisely representable for a `floatmax_t` type, use the macro `BOOST_FLOATMAX_C`.

For example, replace a constant `123.4567890123456789012345678901234567890` with

```
BOOST_FLOATMAX_C(123.4567890123456789012345678901234567890)
```

If, for example, `floatmax_t` is `float64_t` then the result will be equivalent to a `long double` suffixed with L, but if `floatmax_t` is `float128_t` then the result will be equivalent to a `quad type` suffixed with Q (assuming, of course, that `float128_t` (`__float128` or `Quad`) is supported).

If we display with `max_digits10`, the maximum possibly significant decimal digits:

```
#ifdef BOOST_FLOAT32_C
  std::cout.precision(boost::max_digits10<boost::float32_t>()); // Show all significant decim↵
al digits,
  std::cout.setf(std::ios::showpoint); // including all significant trailing zeros.
  std::cout << "BOOST_FLOAT32_C(123.4567890123456789012345678901234567890) = "
    << BOOST_FLOAT32_C(123.4567890123456789012345678901234567890) << std::endl;
  //   BOOST_FLOAT32_C(123.4567890123456789012345678901234567890) = 123.456787
#endif
```

then on a 128-bit platform (GCC 4.8.1 or higher with quadmath):

```
BOOST_FLOAT32_C(123.4567890123456789012345678901234567890) = 123.456787
BOOST_FLOAT64_C(123.4567890123456789012345678901234567890) = 123.45678901234568
BOOST_FLOAT80_C(123.4567890123456789012345678901234567890) = 123.456789012345678903
BOOST_FLOAT128_C(123.4567890123456789012345678901234567890) = 123.456789012345678901234567890123453
```

Floating-Point Constant Macros

All macros of the type `BOOST_FLOAT16_C`, `BOOST_FLOAT32_C`, `BOOST_FLOAT64_C`, `BOOST_FLOAT80_C`, `BOOST_FLOAT128_C`, and `BOOST_FLOATMAX_C` are always defined after inclusion of `<boost/cstdfloat.hpp>`.

These allow floating-point **constants of at least the specified width** to be declared:

```
// Declare Archimedes' constant using float32_t with approximately 7 decimal digits of precision.
static const boost::float32_t pi = BOOST_FLOAT32_C(3.1415926536);

// Declare the Euler-gamma constant with approximately 15 decimal digits of precision.
static const boost::float64_t euler =
   BOOST_FLOAT64_C(0.57721566490153286060651209008240243104216);

// Declare the Golden Ratio constant with the maximum decimal digits of precision that the ↲
platform supports.
static const boost::floatmax_t golden_ratio =
   BOOST_FLOATMAX_C(1.61803398874989484820458683436563811772);
```

Tip

Boost.Math provides many constants 'built-in', so always use Boost.Math constants if available, for example:

```
// Display the constant pi to the maximum available precision.
  boost::floatmax_t pi_max = boost::math::constants::pi<boost::floatmax_t>();
  std::cout.precision(std::numeric_limits<boost::floatmax_t>::digits10);
  std::cout << "Most precise pi = "  << pi_max << std::endl;
// If floatmax_t is float_128_t, then
// Most precise pi = 3.141592653589793238462643383279503
```

from cstdfloat_example.cpp.

See the complete list of constants.

Examples

Jahnke-Emden-Lambda function

The following code uses `<boost/cstdfloat.hpp>` in combination with `<boost/math/special_functions.hpp>` to compute a simplified version of the Jahnke-Emden-Lambda function. Here, we specify a floating-point type with **exactly 64 bits** (i.e., `float64_t`). If we were to use, for instance, built-in `double`, then there would be no guarantee that the code would behave identically on all platforms. With `float64_t` from `<boost/cstdfloat.hpp>`, however, it is very likely to be identical.

Using `float64_t`, we know that this code is as portable as possible and uses a floating-point type with approximately 15 decimal digits of precision, regardless of the compiler or version or operating system.

```
#include <boost/cstdfloat.hpp> // For float_64_t, float128_t. Must be first include!
#include <cmath>  // for pow function.
#include <boost/math/special_functions.hpp> // For gamma function.
```

```
boost::float64_t jahnke_emden_lambda(boost::float64_t v, boost::float64_t x)
{
  const boost::float64_t gamma_v_plus_one = boost::math::tgamma(v + 1);
  const boost::float64_t x_half_pow_v     = std::pow(x /2, v);

  return gamma_v_plus_one * boost::math::cyl_bessel_j(x, v) / x_half_pow_v;
}
```

Ensure that all possibly significant digits (17) including trailing zeros are shown.

```
std::cout.precision(std::numeric_limits<boost::float64_t>::max_digits10);
std::cout.setf(std::ios::showpoint); // Show trailing zeros.

try
{ // Always use try'n'catch blocks to ensure any error messages are displayed.

// Evaluate and display an evaluation of the Jahnke-Emden lambda function:
boost::float64_t v = 1.;
boost::float64_t x = 1.;
std::cout << jahnke_emden_lambda(v, x) << std::endl; // 0.88010117148986700
```

For details, see cstdfloat_example.cpp - a extensive example program.

Normal distribution table

This example shows printing tables of a normal distribution's PDF and CDF, using `boost::math` implementation of normal distribution.

A function templated on floating-point type prints a table for a range of standard variate z values.

The example shows use of the specified-width typedefs to either use a specific width, or to use the maximum available on the platform, perhaps a high as 128-bit.

The number of digits displayed is controlled by the precision of the type, so there are no spurious insignificant decimal digits:

```
float_32_t    0    0.39894228
float_128_t  0    0.398942280401432702863218082711682655
```

Some sample output for two different platforms is appended to the code at normal_tables.cpp.

```
#ifdef BOOST_FLOAT32_C
    normal_table<boost::float32_t>(); // Usually type float
#endif
   normal_table<boost::float64_t>(); // Uusually type double. Assume that float64_t is always ↵
available.
#ifdef BOOST_FLOAT80_C
    normal_table<boost::float80_t>(); // Type long double on some X86 platforms.
#endif
#ifdef BOOST_FLOAT128_C
   normal_table<boost::float128_t>(); // Type _Quad on some Intel and __float128 on some GCC ↵
platforms.
#endif
    normal_table<boost::floatmax_t>();
```

Hints on using float128 (and __float128)

__float128 versus float128

- __float128 is the (optionally) compiler supplied hardware type, it's an C-ish extension to C++ and there is only minimal support for it in normal C++ (no IO streams or `numeric_limits` support, function names in libquadmath all have different names to the `std::` ones etc.) So you can program type `__float128` directly, but it's harder work.
- Type `float128` uses __float128 and makes it C++ and generic code friendly, with all the usual standard `iostream`, `numeric_limits`, `complex` in namspace `std::` available, so strongly recommended for C++ use.

Hints and tips

- Make sure you declare variables with the correct type, here `float128`.
- Make sure that if you pass a variable to a function then it is casted to `float128`.
- Make sure you declare literals with the correct suffix - otherwise they'll be treated as type `double` with catastrophic loss of precision. So make sure they have a Q suffix for 128-bit floating-point literals.
- All the std library functions, cmath functions, plus all the constants, and special functions from Boost.Math should then just work.
- Make sure std lib functions are called **unqualified** so that the correct overload is found via Argument Dependent Lookup (ADL). So write sqrt(variable) and not std::sqrt(variable).
- In general, try not to reinvent stuff - using constants from Boost.Math is probably less error prone than declaring your own, likewise the special functions etc.

Some examples of what can go horribly and silently wrong are at float128_example.cpp.

Implementation of Float128 type

Since few compilers implement a true 128-bit floating-point, and language features like the suffix Q (which may need an option -fext-numeric-literals to enable), and C++ Standard library functions are as-yet missing or incomplete in C++11, this Boost.Math implementation wraps __float128 provided by the GCC compiler GCC floating-point types or the _Quad type provided by the Intel compiler.

This is provided to in order to demonstrate, and users to evaluate, the feasibility and benefits of higher-precision floating-point, especially to allow use of the full <cmath> and Boost.Math library of functions and distributions at high precision.

(It is also possible to use Boost.Math with Boost.Multiprecision decimal and binary, but since these are entirely software solutions, allowing much higher precision or arbitrary precision, they are likely to be slower).

We also provide (we believe full) support for <limits>, <cmath>, I/O stream operations in <iostream>, and <complex>.

As a prototype for a future C++ standard, we place all these in namespace std. This contravenes the existing C++ standard of course, so selecting any compiler that promises to check conformance will fail.

Tip

For GCC, compile with -std=gnu++11 or -std=gnu++03 and do not use -std=stdc++11 or any 'strict' options, as these turn off full support for __float128. These requirements also apply to the Intel compiler on Linux, for Intel on Windows you need to compile with -Qoption,cpp,--extended_float_type -DBOOST_MATH_USE_FLOAT128 in order to activate 128-bit floating point support.

The __float128 type is provided by the libquadmath library on GCC or by Intel's FORTRAN library with Intel C++. They also provide a full set of <cmath> functions in namespace std.

Using C __float128 quadmath type

The source code is at quadmath_snprintf.c.

Using C++ float128 quadmath type

For C++ programs, you will want to use the C++ type float128

See example at cstdfloat_example.cpp.

A typical invocation of the compiler is

```
g++ -O3 -std=gnu++11 test.cpp -I/c/modular-boost -lquadmath -o test.exe
```

Tip

If you are trying to use the develop branch of Boost.Math, then make -I/c/modular-boost/libs/math/include the **first** include directory.

```
g++ -O3 -std=gnu++11 test.cpp -I/c/modular-boost/libs/math/include -I/c/modular-boost -lquad↵
math -o test.exe
```

Note

So far, the only missing detail that we had noted was in trying to use `<typeinfo>`, for example for `std::cout << typeid<__float_128>.name();`.

```
Link fails: undefined reference to typeinfo for __float128.
```

See GCC Bug 43622 - no C++ typeinfo for __float128. But this is reported (Marc Glisse 2015-04-04) fixed in GCC 5 (and above).

For example, with GCC6.1.1 this works as expected to a **mangled** string name, and output (if possible - not always).

```
const std::type_info& tifu128 = typeid(__float128); // OK.
//std::cout << tifu128.name() << std::endl; // On GCC, aborts (because not print↵
able string).
//std::cout << typeid(__float128).name() << std::endl; // Aborts - string name can↵
not be output.

const std::type_info& tif128 = typeid(float128); // OK.
std::cout << tif128.name() << std::endl; // OK.
std::cout << typeid(float128).name() << std::endl; // OK.

const std::type_info& tpi = typeid(pi1); // OK GCC 6.1.1 (from GCC 5 according to ↵
http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43622)
std::cout << tpi.name() << std::endl; // Output mangled name:

// N5boost14multiprecision6numberINS0_8backends16float128_backendELNS0_26expres↵
sion_template_optionE0EEE
```

Overloading template functions with float128_t

An artifact of providing C++ standard library support for quadmath may mandate the inclusion of `<boost/cstdfloat.hpp>` **before** the inclusion of other headers.

Consider a function that calls `fabs(x)` and has previously injected `std::fabs()` into local scope via a `using` directive:

```
template <class T>
bool unsigned_compare(T a, T b)
{
   using std::fabs;
   return fabs(a) == fabs(b);
}
```

In this function, the correct overload of `fabs` may be found via argument-dependent-lookup (ADL) or by calling one of the `std::fabs` overloads. There is a key difference between them however: an overload in the same namespace as T and found via ADL need ***not be defined at the time the function is declared***. However, all the types declared in `<boost/cstdfloat.hpp>` are fundamental types, so for these types we are relying on finding an overload declared in namespace `std`. In that case however, ***all such overloads must be declared prior to the definition of function `unsigned_compare` otherwise they are not considered***.

In the event that `<boost/cstdfloat.hpp>` has been included **after** the definition of the above function, the correct overload of `fabs`, while present, is simply not considered as part of the overload set. So the compiler tries to downcast the `float128_t` argument first to `long double`, then to `double`, then to `float`; the compilation fails because the result is ambiguous. However the compiler error message will appear cruelly inscrutable, at an apparently irelevant line number and making no mention of `float128`: the word *ambiguous* is the clue to what is wrong.

Provided you `#include <boost/cstdfloat.hpp>` **before** the inclusion of the any header containing generic floating point code (such as other Boost.Math headers, then the compiler will know about and use the `std::fabs(std::float128_t)` that we provide in `#include <boost/cstdfloat.hpp>`.

Exponential function

There was a bug when using any quadmath `expq` function on GCC : GCC bug #60349 caused by mingw-64 bug #368.

To work round this defect, an alternative implementation of 128-bit exp was temporarily provided by `boost/cstdfloat.hpp`.

The mingw bug was fixed at 2014-03-12 and GCC 6.1.1 now works as expected.

Tip

It is essential to link to the quadmath library

.

`typeinfo`

For GCC 4.8.1 it was not yet possible to use `typeinfo` for `float_128` on GCC: see GCC 43622.

So this code (to display the mangled name) failed to link `undefined reference to typeinfo for __float128`

```
std::cout << typeid(boost::float128_t).name() << std::endl;
```

This prevent using the existing tests for Boost.Math distributions, (unless a few lines are commented out) and if a MACRO BOOST_MATH_INSTRUMENT controlling them is defined then some diagnostic displays in Boost.Math will not work.

However this was only used for display purposes and could be commented out until this was fixed in GCC 5.

Tip

Not all managed names can be **displayed** using `std::cout`.

Mathematical Constants

Introduction

Boost.Math provides a collection of mathematical constants.

Why use Boost.Math mathematical constants?

- Readable. For the very many jobs just using built-in like `double`, you can just write expressions like

  ```
  double area = pi * r * r;
  ```

  (If that's all you want, jump direct to use in non-template code!)

- Effortless - avoiding a search of reference sources.
- Usable with both builtin floating point types, and user-defined, possibly extended precision, types such as NTL, MPFR/GMP, mp_float: in the latter case the constants are computed to the necessary precision and then cached.
- Accurate - ensuring that the values are as accurate as possible for the chosen floating-point type
  - No loss of accuracy from repeated rounding of intermediate computations.
  - Result is computed with higher precision and only rounded once.
  - Less risk of inaccurate result from functions pow, trig and log at corner cases.
  - Less risk of cancellation error.
- Portable - as possible between different systems using different floating-point precisions: see use in template code.
- Tested - by comparison with other published sources, or separately computed at long double precision.
- Faster - can avoid (re-)calculation at runtime.
  - If the value returned is a builtin type then it's returned by value as a `constexpr` (C++11 feature, if available).
  - If the value is computed and cached (or constructed from a string representation and cached), then it's returned by constant reference.

  This can be significant if:

  - Functions pow, trig or log are used.
  - Inside an inner loop.
  - Using a high-precision UDT like Boost.Multiprecision.
  - Compiler optimizations possible with built-in types, especially `double`, are not available.

Tutorial

Use in non-template code

When using the math constants at your chosen fixed precision in non-template code, you can simply add a `using namespace` declaration, for example, `using namespace boost::math::double_constants`, to make the constants of the correct precision for your code visible in the current scope, and then use each constant *as a simple variable - sans brackets*:

```
#include <boost/math/constants/constants.hpp>

double area(double r)
{
   using namespace boost::math::double_constants;
   return pi * r * r;
}
```

Had our function been written as taking a `float` rather than a `double`, we could have written instead:

```
#include <boost/math/constants/constants.hpp>

float area(float r)
{
   using namespace boost::math::float_constants;
   return pi * r * r;
}
```

Likewise, constants that are suitable for use at `long double` precision are available in the namespace `boost::math::long_double_constants`.

You can see the full list of available constants at math_toolkit.constants.

Some examples of using constants are at constants_eg1.

Use in template code

When using the constants inside a function template, we need to ensure that we use a constant of the correct precision for our template parameters. We can do this by calling the function-template versions, `pi<FPType>()`, of the constants like this:

```
#include <boost/math/constants/constants.hpp>

template <class Real>
Real area(Real r)
{
   using namespace boost::math::constants;
   return pi<Real>() * r * r;
}
```

Although this syntax is a little less "cute" than the non-template version, the code is no less efficient (at least for the built-in types `float`, `double` and `long double`) : the function template versions of the constants are simple inline functions that return a constant of the correct precision for the type used. In addition, these functions are declared `constexp` for those compilers that support this, allowing the result to be used in constant-expressions provided the template argument is a literal type.

Tip

Keep in mind the difference between the variable version, just `pi`, and the template-function version: the template-function requires both a <*`floating-point-type`*> and function call `()` brackets, for example: `pi<double>()`. You cannot write `double p = pi<>()`, nor `double p = pi()`.

Note

You can always use **both** variable and template-function versions **provided calls are fully qualified**, for example:

```
double my_pi1 = boost::math::constants::pi<double>();
double my_pi2 = boost::math::double_constants::pi;
```

Warning

It may be tempting to simply define

```
using namespace boost::math::double_constants;
using namespace boost::math::constants;
```

but if you do define two namespaces, this will, of course, create ambiguity!

```
double my_pi = pi(); // error C2872: 'pi' : ambiguous symbol
double my_pi2 = pi; // Context does not allow for disambiguation of overloaded ↵
function
```

Although the mistake above is fairly obvious, it is also not too difficult to do this accidentally, or worse, create it in someone elses code.

Therefore is it prudent to avoid this risk by **localising the scope of such definitions**, as shown above.

Tip

Be very careful with the type provided as parameter. For example, providing an **integer** instead of a floating-point type can be disastrous (a C++ feature).

```
cout << "Area = " << area(2) << endl; // Area = 12!!!
```

You should get a compiler warning

```
warning : 'return' : conversion from 'double' to 'int', possible loss of data
```

Failure to heed this warning can lead to very wrong answers!

You can also avoid this by being explicit about the type of `Area`.

```
cout << "Area = " << area<double>(2) << endl; // Area = 12.566371
```

Use With User-Defined Types

The most common example of a high-precision user-defined type will probably be Boost.Multiprecision.

The syntax for using the function-call constants with user-defined types is the same as it is in the template class, which is to say we use:

```
#include <boost/math/constants/constants.hpp>

boost::math::constants::pi<UserDefinedType>();
```

For example:

```
boost::math::constants::pi<boost::multiprecision::cpp_dec_float_50>();
```

giving π with a precision of 50 decimal digits.

However, since the precision of the user-defined type may be much greater than that of the built-in floating point types, how the value returned is created is as follows:

- If the precision of the type is known at compile time:
  - If the precision is less than or equal to that of a `float` and the type is constructable from a `float` then our code returns a `float` literal. If the user-defined type is a literal type then the function call that returns the constant will be a `constexp`.
  - If the precision is less than or equal to that of a `double` and the type is constructable from a `double` then our code returns a `double` literal. If the user-defined type is a literal type then the function call that returns the constant will be a `constexp`.
  - If the precision is less than or equal to that of a `long double` and the type is constructable from a `long double` then our code returns a `long double` literal. If the user-defined type is a literal type then the function call that returns the constant will be a `constexp`.
  - If the precision is less than or equal to that of a `__float128` (and the compiler supports such a type) and the type is constructable from a `__float128` then our code returns a `__float128` literal. If the user-defined type is a literal type then the function call that returns the constant will be a `constexp`.
  - If the precision is less than 100 decimal digits, then the constant will be constructed (just the once, then cached in a thread-safe manner) from a string representation of the constant. In this case the value is returned as a const reference to the cached value.
  - Otherwise the value is computed (just once, then cached in a thread-safe manner). In this case the value is returned as a const reference to the cached value.
- If the precision is unknown at compile time then:
  - If the runtime precision (obtained from a call to `boost::math::tools::digits<T>()`) is less than 100 decimal digits, then the constant is constructed "on the fly" from the string representation of the constant.
  - Otherwise the value is constructed "on the fly" by calculating then value of the constant using the current default precision of the type. Note that this can make use of the constants rather expensive.

In addition, it is possible to pass a `Policy` type as a second template argument, and use this to control the precision:

```
#include <boost/math/constants/constants.hpp>

typedef boost::math::policies::policy<boost::math::policies::digits2<80> > my_policy_type;
boost::math::constants::pi<MyType, my_policy_type>();
```

Note

Boost.Math doesn't know how to control the internal precision of `MyType`, the policy just controls how the selection process above is carried out, and the calculation precision if the result is computed.

It is also possible to control which method is used to construct the constant by specialising the traits class `construction_traits`:

```
namespace boost{ namespace math{ namespace constant{

template <class T, class Policy>
struct construction_traits
{
   typedef mpl::int_<N> type;
};

}}} // namespaces
```

Where *N* takes one of the following values:

N	**Meaning**
0	The precision is unavailable at compile time; either construct from a decimal digit string or calculate on the fly depending upon the runtime precision.
1	Return a float precision constant.
2	Return a double precision constant.
3	Return a long double precision constant.
4	Construct the result from the string representation, and cache the result.
Any other value *N*	Sets the compile time precision to *N* bits.

Custom Specializing a constant

In addition, for user-defined types that need special handling, it's possible to partially-specialize the internal structure used by each constant. For example, suppose we're using the C++ wrapper around MPFR `mpfr_class`: this has its own representation of Pi which we may well wish to use in place of the above mechanism. We can achieve this by specialising the class template `boost::math::constants::detail::constant_pi`:

```
namespace boost{ namespace math{ namespace constants{ namespace detail{

template<>
struct constant_pi<mpfr_class>
{
   template<int N>
   static mpfr_class get(const mpl::int_<N>&)
   {
      // The template param N is one of the values in the table above,
      // we can either handle all cases in one as is the case here,
      // or overload "get" for the different options.
      mpfr_class result;
      mpfr_const_pi(result.get_mpfr_t(), GMP_RNDN);
      return result;
   }
};

}}}} // namespaces
```

Diagnosing what meta-programmed code is doing

Finally, since it can be tricky to diagnose what meta-programmed code is doing, there is a diagnostic routine that prints information about how this library will handle a specific type, it can be used like this:

```
#include <boost/math/constants/info.hpp>

int main()
{
   boost::math::constants::print_info_on_type<MyType>();
}
```

If you wish, you can also pass an optional std::ostream argument to the `print_info_on_type` function. Typical output for a user-defined type looks like this:

```
Information on the Implementation and Handling of
Mathematical Constants for Type class boost::math::concepts::real_concept

Checking for std::numeric_limits<class boost::math::concepts::real_concept> specialisation: no
boost::math::policies::precision<class boost::math::concepts::real_concept, Policy>
reports that there is no compile type precision available.
boost::math::tools::digits<class boost::math::concepts::real_concept>()
reports that the current runtime precision is
53 binary digits.
No compile time precision is available, the construction method
will be decided at runtime and results will not be cached
- this may lead to poor runtime performance.
Current runtime precision indicates that
the constant will be constructed from a string on each call.
```

The Mathematical Constants

This section lists the mathematical constants, their use(s) (and sometimes rationale for their inclusion).

Table 14. Mathematical Constants

name	formula	Value (6 decimals)	Uses and Rationale
Rational fractions			
half	1/2	0.5	
third	1/3	0.333333	
two_thirds	2/3	0.66667	
three_quarters	3/4	0.75	
two and related			
root_two	√2	1.41421	
root_three	√3	1.73205	
half_root_two	√2 /2	0.707106	
ln_two	ln(2)	0.693147	
ln_ten	ln(10)	2.30258	
ln_ln_two	ln(ln(2))	-0.366512	Gumbel distribution median
root_ln_four	√ln(4)	1.177410	
one_div_root_two	1/√2	0.707106	
π and related			
pi	pi	3.14159	Ubiquitous. Archimedes constant π
half_pi	π/2	1.570796	
third_pi	π/3	1.04719	
sixth_pi	π/6	0.523598	
two_pi	2π	6.28318	Many uses, most simply, circumference of a circle
two_thirds_pi	2/3 π	2.09439	volume of a hemi-sphere = 4/3 π r^3
three_quarters_pi	3/4 π	2.35619	= 3/4 π
four_thirds_pi	4/3 π	4.18879	volume of a sphere = 4/3 π r^3
one_div_two_pi	1/(2π)	1.59155	Widely used
root_pi	√π	1.77245	Widely used
root_half_pi	√ π/2	1.25331	Widely used

name	**formula**	**Value (6 decimals)**	**Uses and Rationale**
root_two_pi	√ π*2	2.50662	Widely used
one_div_root_pi	1/√π	0.564189	
one_div_root_two_pi	1/√(2π)	0.398942	
root_one_div_pi	√(1/π	0.564189	
pi_minus_three	π-3	0.141593	
four_minus_pi	4 -π	0.858407	
pi_pow_e	π^e	22.4591	
pi_sqr	π^2	9.86960	
pi_sqr_div_six	$\pi^2/6$	1.64493	
pi_cubed	π^3	31.00627	
cbrt_pi	$\sqrt{}^3\ \pi$	1.46459	
one_div_cbrt_pi	$1/\sqrt{}^3\ \pi$	0.682784	
Euler's e and related			
e	e	2.71828	Euler's constant e
exp_minus_half	$e^{-1/2}$	0.606530	
e_pow_pi	e^{π}	23.14069	
root_e	√ e	1.64872	
log10_e	log10(e)	0.434294	
one_div_log10_e	1/log10(e)	2.30258	
Trigonometric			
degree	radians = π / 180	0.017453	
radian	degrees = 180 / π	57.2957	
sin_one	sin(1)	0.841470	
cos_one	cos(1)	0.54030	
sinh_one	sinh(1)	1.17520	
cosh_one	cosh(1)	1.54308	
Phi	Phidias golden ratio	Phidias golden ratio	
phi	(1 + √5) /2	1.61803	finance

name	formula	Value (6 decimals)	Uses and Rationale
ln_phi	ln(ϕ)	0.48121	
one_div_ln_phi	1/ln(ϕ)	2.07808	
Euler's Gamma			
euler	euler	0.577215	Euler-Mascheroni gamma constant
one_div_euler	1/euler	1.73245	
euler_sqr	$euler^2$	0.333177	
Misc			
zeta_two	ζ(2)	1.64493	Riemann zeta function
zeta_three	ζ(3)	1.20205	Riemann zeta function
catalan	*K*	0.915965	Catalan (or Glaisher) combinatorial constant
glaisher	*A*	1.28242	Decimal expansion of Glaisher-Kinkelin constant
khinchin	*k*	2.685452	Decimal expansion of Khinchin constant
extreme_value_skewness	$12\sqrt{6}\ \zeta(3)/\ \pi^3$	1.139547	Extreme value distribution
rayleigh_skewness	$2\sqrt{\pi}(\pi-3)/(4 - \pi)^{3/2}$	0.631110	Rayleigh distribution skewness
rayleigh_kurtosis_excess	$-(6\pi^2-24\pi+16)/(4-\pi)^2$	0.245089	Rayleigh distribution kurtosis excess
rayleigh_kurtosis	$3+(6\pi^2-24\pi+16)/(4-\pi)^2$	3.245089	Rayleigh distribution kurtosis

Note

Integer values are **not included** in this list of math constants, however interesting, because they can be so easily and exactly constructed, even for UDT, for example: `static_cast<cpp_float>(42)`.

Tip

If you know the approximate value of the constant, you can search for the value to find Boost.Math chosen name in this table.

Tip

Bernoulli numbers are available at Bernoulli numbers.

Tip

Factorials are available at factorial.

Defining New Constants

The library provides some helper code to assist in defining new constants; the process for defining a constant called `my_constant` goes like this:

1. **Define a function that calculates the value of the constant**. This should be a template function, and be placed in `boost/math/constants/calculate_constants.hpp` if the constant is to be added to this library, or else defined at the top of your source file if not.

The function should look like this:

```
namespace boost{ namespace math{ namespace constants{ namespace detail{

template <class Real>
template <int N>
Real constant_my_constant<Real>::compute(BOOST_MATH_EXPLICIT_TEMPLATE_TYPE_SPEC(mpl::int_<N>))
{
  int required_precision = N ? N : tools::digits<Real>();
  Real result = /* value computed to required_precision bits */ ;
  return result;
}

}}}} // namespaces
```

Then define a placeholder for the constant itself:

```
namespace boost{ namespace math{ namespace constants{

BOOST_DEFINE_MATH_CONSTANT(my_constant, 0.0, "0");

}}}
```

For example, to calculate π/2, add to `boost/math/constants/calculate_constants.hpp`

```
template <class T>
template<int N>
inline T constant_half_pi<T>::compute(BOOST_MATH_EXPLICIT_TEMPLATE_TYPE_SPEC(mpl::int_<N>))
{
   BOOST_MATH_STD_USING
   return pi<T, policies::policy<policies::digits2<N> > >() / static_cast<T>(2);
}
```

Then to `boost/math/constants/constants.hpp` add:

```
BOOST_DEFINE_MATH_CONSTANT(half_pi, 0.0, "0");  // Actual values are temporary, we'll replace ↵
them later.
```

Note

Previously defined constants like pi and e can be used, but by **not simply calling** `pi<T>();` specifying the precision via the policy `pi<T, policies::policy<policies::digits2<N> > >()` is essential to ensure full accuracy.

Warning

Newly defined constants can only be used once they are included in `boost/math/constants/constants.hpp`. So if you add `template <class T, class N> T constant_my_constant{...}`, then you cannot define `constant_my_constant` until you add the temporary `BOOST_DEFINE_MATH_CONSTANT(my_constant, 0.0, "0")`. Failing to do this will result in surprising compile errors:

```
error C2143: syntax error : missing ';' before '<'
error C2433: 'constant_root_two_div_pi' : 'inline' not permitted on data declara↵
tions
error C2888: 'T constant_root_two_div_pi' : symbol cannot be defined with↵
in namespace 'detail'
error C2988: unrecognizable template declaration/definition
```

2. **You will need an arbitrary precision type to use to calculate the value**. This library currently supports either `cpp_float`, `NTL::RR` or `mpfr_class` used via the bindings in `boost/math/bindings`. The default is to use `NTL::RR` unless you define an alternate macro, for example, `USE_MPFR` or `USE_CPP_FLOAT` at the start of your program.

3. It is necessary to link to the Boost.Regex library, and probably to your chosen arbitrary precision type library.

4. You need to add `libs\math\include_private` to your compiler's include path as the needed header is not installed in the usual places by default (this avoids a cyclic dependency between the Math and Multiprecision library's headers).

5. The complete program to generate the constant `half_pi` using function `calculate_half_pi` is then:

```
#define USE_CPP_FLOAT // If required.
#include <boost/math/constants/generate.hpp>

int main()
{
   BOOST_CONSTANTS_GENERATE(half_pi);
}
```

The output from the program is a snippet of C++ code (actually a macro call) that can be cut and pasted into `boost/math/constants/constants.hpp` or else into your own code, for example:

```
  BOOST_DEFINE_MATH_CONSTANT(half_pi, 1.570796326794896619231321691639751442e+00, ↵
"1.57079632679489661923132169163975144209858469968755291048747229615390820314310449931401741267105853399107404326e+00");
```

This macro BOOST_DEFINE_MATH_CONSTANT inserts a C++ struct code snippet that declares the `float`, `double` and `long double` versions of the constant, plus a decimal digit string representation correct to 100 decimal digits, and all the meta-programming machinery needed to select between them.

The result of an expanded macro for Pi is shown below.

```
// Preprocessed pi constant, annotated.

namespace boost
{
  namespace math
  {
    namespace constants
    {
      namespace detail
      {
        template <class T> struct constant_pi
        {
          private:
            // Default implementations from string of decimal digits:
            static inline T get_from_string()
            {
            static const T result
               = detail::con↵
vert_from_string<T>("3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651e+00",
               boost::is_convertible<const char*, T>());
              return result;
            }
            template <int N> static T compute();

          public:
            // Default implementations from string of decimal digits:
            static inline T get(const mpl::int_<construct_from_string>&)
            {
              constant_initializer<T, & constant_pi<T>::get_from_string >::do_nothing();
              return get_from_string();
            }
            // Float, double and long double versions:
            static inline T get(const mpl::int_<construct_from_float>)
            {
              return 3.141592653589793238462643383279502884e+00F;
            }
            static inline  T get(const mpl::int_<construct_from_double>&)
            {
              return 3.141592653589793238462643383279502884e+00;
            }
            static inline  T get(const mpl::int_<construct_from_long_double>&)
            {
              return 3.141592653589793238462643383279502884e+00L;
            }
            // For very high precision that is nonetheless can be calculated at compile time:
            template <int N> static inline T get(const mpl::int_<N>& n)
            {
            constant_initializer2<T, N, & constant_pi<T>::template compute<N> >::do_nothing();
              return compute<N>();
            }
            //For true arbitrary precision, which may well vary at runtime.
            static inline T get(const mpl::int_<0>&)
            {
             return tools::digits<T>() > max_string_digits ? compute<0>() : get(mpl::int_<con↵
struct_from_string>());
            }
         }; // template <class T> struct constant_pi
      } //  namespace detail

      // The actual forwarding function (including policy to control precision).
      template <class T, class Policy> inline T pi( )
      {
        return detail:: constant_pi<T>::get(typename construction_traits<T, Policy>::type());
      }
      // The actual forwarding function (using default policy to control precision).
      template <class T> inline  T pi()
      {
        return pi<T, boost::math::policies::policy<> >()
      }
    } //     namespace constants

    // Namespace specific versions, for the three built-in floats:
    namespace float_constants
    {
      static const float pi = 3.141592653589793238462643383279502884e+00F;
    }
    namespace double_constants
    {
      static const double pi = 3.141592653589793238462643383279502884e+00;
    }
    namespace long_double_constants
    {
      static const long double pi = 3.141592653589793238462643383279502884e+00L;
    }
    namespace constants{;
    } // namespace constants
  } // namespace math
} // namespace boost
```

Math Constants FAQs

Why are *these* Constants Chosen?

It is, of course, impossible to please everyone with a list like this.

Some of the criteria we have used are:

- Used in Boost.Math.
- Commonly used.
- Expensive to compute.
- Requested by users.
- Used in science and mathematics.
- No integer values (because so cheap to construct).

(You can easily define your own if found convenient, for example: `FPT one =static_cast<FPT>(42);`).

How are constants named?

- Not macros, so no upper case.
- All lower case (following C++ standard names).
- No CamelCase.
- Underscore as _ delimiter between words.
- Numbers spelt as words rather than decimal digits (except following pow).
- Abbreviation conventions:
  - root for square root.
  - cbrt for cube root.
  - pow for pow function using decimal digits like pow23 for $n^{2/3}$.
  - div for divided by or operator /.
  - minus for operator -, plus for operator +.
  - sqr for squared.
  - cubed for cubed n^3.
  - words for greek, like π, ζ and Γ.
  - words like half, third, three_quarters, sixth for fractions. (Digit(s) can get muddled).
  - log10 for $\log_{10}$
  - ln for $\log_e$

How are the constants derived?

The constants have all been calculated using high-precision software working with up to 300-bit precision giving about 100 decimal digits. (The precision can be arbitrarily chosen and is limited only by compute time).

How Accurate are the constants?

The minimum accuracy chosen (100 decimal digits) exceeds the accuracy of reasonably-foreseeable floating-point hardware (256-bit) and should meet most high-precision computations.

How are the constants tested?

1. Comparison using Boost.Test BOOST_CHECK_CLOSE_FRACTION using long double literals, with at least 35 decimal digits, enough to be accurate for all long double implementations. The tolerance is usually twice `long double epsilon`.

2. Comparison with calculation at long double precision. This often requires a slightly higher tolerance than two epsilon because of computational noise from round-off etc, especially when trig and other functions are called.

3. Comparison with independent published values, for example, using The On-Line Encyclopedia of Integer Sequences (OEIS) again using at least 35 decimal digits strings.

4. Comparison with independely calculated values using arbitrary precision tools like Mathematica, again using at least 35 decimal digits literal strings.

Warning

We have not yet been able to **check** that **all** constants are accurate at the full arbitrary precision, at present 100 decimal digits. But certain key values like `e` and `pi` appear to be accurate and internal consistencies suggest that others are this accurate too.

Why is Portability important?

Code written using math constants is easily portable even when using different floating-point types with differing precision.

It is a mistake to expect that results of computations will be **identical**, but you can achieve the **best accuracy possible for the floating-point type in use**.

This has no extra cost to the user, but reduces irritating, and often confusing and very hard-to-trace effects, caused by the intrinsically limited precision of floating-point calculations.

A harmless symptom of this limit is a spurious least-significant digit; at worst, slightly inaccurate constants sometimes cause iterating algorithms to diverge wildly because internal comparisons just fail.

What is the Internal Format of the constants, and why?

See tutorial above for normal use, but this FAQ explains the internal details used for the constants.

Constants are stored as 100 decimal digit values. However, some compilers do not accept decimal digits strings as long as this. So the constant is split into two parts, with the first containing at least 128-bit long double precision (35 decimal digits), and for consistency should be in scientific format with a signed exponent.

The second part is the value of the constant expressed as a string literal, accurate to at least 100 decimal digits (in practice that means at least 102 digits). Again for consistency use scientific format with a signed exponent.

For types with precision greater than a long double, then if T is constructible `T` is constructible from a `const char*` then it's directly constructed from the string, otherwise we fall back on lexical_cast to convert to type `T`. (Using a string is necessary because you can't use a numeric constant since even a `long double` might not have enough digits).

So, for example, a constant like pi is internally defined as

```
BOOST_DEFINE_MATH_CON↵
STANT(pi, 3.141592653589793238462643383279502884e+00, "3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651e+00");
```

In this case the significand is 109 decimal digits, ensuring 100 decimal digits are exact, and exponent is zero.

See defining new constants to calculate new constants.

A macro definition like this can be pasted into user code where convenient, or into `boost/math/constants.hpp` if it is to be added to the Boost.Math library.

What Floating-point Types could I use?

Apart from the built-in floating-point types `float`, `double`, `long double`, there are several arbitrary precision floating-point classes available, but most are not licensed for commercial use.

Boost.Multiprecision by Christopher Kormanyos

This work is based on an earlier work called e-float: Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations, in ACM TOMS, {VOL 37, ISSUE 4, (February 2011)} (C) ACM, 2011. http://doi.acm.org/10.1145/1916461.1916469 e_float but is now re-factored and available under the Boost license in the Boost-sandbox at multiprecision where it is being refined and prepared for review.

Boost.cpp_float by John Maddock using Expression Templates

Big Number which is a reworking of e_float by Christopher Kormanyos to use expression templates for faster execution.

NTL class quad_float

NTL by Victor Shoup has fixed and arbitrary high precision fixed and floating-point types. However none of these are licenced for commercial use.

```
#include <NTL/quad_float.h> // quad precision 106-bit, about 32 decimal digits.
using NTL::to_quad_float; // Less precise than arbitrary precision NTL::RR.
```

NTL class `quad_float`, which gives a form of quadruple precision, 106-bit significand (but without an extended exponent range.) With an IEC559/IEEE 754 compatible processor, for example Intel X86 family, with 64-bit double, and 53-bit significand, using the significands of **two** 64-bit doubles, if `std::numeric_limits<double>::digits10` is 16, then we get about twice the precision, so `std::numeric_limits<quad_float>::digits10()` should be 32. (the default `std::numeric_limits<RR>::digits10()` should be about 40). (which seems to agree with experiments). We output constants (including some noisy bits, an approximation to `std::numeric_limits<RR>::max_digits10()`) by adding 2 or 3 extra decimal digits, so using `quad_float::SetOutputPrecision(32 + 3);`

Apple Mac/Darwin uses a similar *doubledouble* 106-bit for its built-in `long double` type.

Note

The precision of all `doubledouble` floating-point types is rather odd and values given are only approximate.

New projects should use Boost.Multiprecision.

NTL class RR

Arbitrary precision floating point with NTL class RR, default is 150 bit (about 50 decimal digits) used here with 300 bit to output 100 decimal digits, enough for many practical non-'number-theoretic' C++ applications.

NTL A Library for doing Number Theory is **not licenced for commercial use**.

This class is used in Boost.Math and is an option when using big_number projects to calculate new math constants.

New projects should use Boost.Multiprecision.

GMP and MPFR

GMP and MPFR have also been used to compute constants, but are licensed under the Lesser GPL license and are **not licensed for commercial use**.

What happened to a previous collection of constants proposed for Boost?

A review concluded that the way in which the constants were presented did not meet many peoples needs. None of the methods proposed met many users' essential requirement to allow writing simply `pi` rather than `pi()`. Many science and engineering equations look difficult to read when because function call brackets can be confused with the many other brackets often needed. All the methods then proposed of avoiding the brackets failed to meet all needs, often on grounds of complexity and lack of applicability to various realistic scenarios.

So the simple namespace method, proposed on its own, but rejected at the first review, has been added to allow users to have convenient access to float, double and long double values, but combined with template struct and functions to allow simultaneous use with other non-built-in floating-point types.

Why do the constants (internally) have a struct rather than a simple function?

A function mechanism was provided by in previous versions of Boost.Math.

The new mechanism is to permit partial specialization. See Custom Specializing a constant above. It should also allow use with other packages like ttmath Bignum C++ library.

Where can I find other high precision constants?

1. Constants with very high precision and good accuracy (>40 decimal digits) from Simon Plouffe's web based collection http://pi.lacim.uqam.ca/eng/.
2. The On-Line Encyclopedia of Integer Sequences (OEIS)
3. Checks using printed text optically scanned values and converted from: D. E. Knuth, Art of Computer Programming, Appendix A, Table 1, Vol 1, ISBN 0 201 89683 4 (1997)
4. M. Abrahamovitz & I. E. Stegun, National Bureau of Standards, Handbook of Mathematical Functions, a reference source for formulae now superceded by
5. Frank W. Olver, Daniel W. Lozier, Ronald F. Boisvert, Charles W. Clark, NIST Handbook of Mathemetical Functions, Cambridge University Press, ISBN 978-0-521-14063-8, 2010.
6. John F Hart, Computer Approximations, Kreiger (1978) ISBN 0 88275 642 7.
7. Some values from Cephes Mathematical Library, Stephen L. Moshier and CALC100 100 decimal digit Complex Variable Calculator Program, a DOS utility.
8. Xavier Gourdon, Pascal Sebah, 50 decimal digits constants at Number, constants and computation.

Where are Physical Constants?

Not here in this Boost.Math collection, because physical constants:

- Are measurements, not truely constants.
- Are not truly constant and keeping changing as mensuration technology improves.
- Have a instrinsic uncertainty.

- Mathematical constants are stored and represented at varying precision, but should never be inaccurate.

Some physical constants may be available in Boost.Units.

Statistical Distributions and Functions

Statistical Distributions Tutorial

This library is centred around statistical distributions, this tutorial will give you an overview of what they are, how they can be used, and provides a few worked examples of applying the library to statistical tests.

Overview of Distributions

Headers and Namespaces

All the code in this library is inside namespace boost::math.

In order to use a distribution *my_distribution* you will need to include either the header <boost/math/my_distribution.hpp> or the "include all the distributions" header: <boost/math/distributions.hpp>.

For example, to use the Students-t distribution include either <boost/math/students_t.hpp> or <boost/math/distributions.hpp>

You also need to bring distribution names into scope, perhaps with a `using namespace boost::math;` declaration,

or specific `using` declarations like `using boost::math::normal;` (**recommended**).

Caution

Some math function names are also used in namespace std so including <random> could cause ambiguity!

Distributions are Objects

Each kind of distribution in this library is a class type - an object.

Policies provide fine-grained control of the behaviour of these classes, allowing the user to customise behaviour such as how errors are handled, or how the quantiles of discrete distribtions behave.

Tip

If you are familiar with statistics libraries using functions, and 'Distributions as Objects' seem alien, see the comparison to other statistics libraries.

Making distributions class types does two things:

- It encapsulates the kind of distribution in the C++ type system; so, for example, Students-t distributions are always a different C++ type from Chi-Squared distributions.
- The distribution objects store any parameters associated with the distribution: for example, the Students-t distribution has a *degrees of freedom* parameter that controls the shape of the distribution. This *degrees of freedom* parameter has to be provided to the Students-t object when it is constructed.

Although the distribution classes in this library are templates, there are typedefs on type *double* that mostly take the usual name of the distribution (except where there is a clash with a function of the same name: beta and gamma, in which case using the default template arguments - `RealType = double` - is nearly as convenient). Probably 95% of uses are covered by these typedefs:

```
// using namespace boost::math; // Avoid potential ambiguity with names in std <random>
// Safer to declare specific functions with using statement(s):

using boost::math::beta_distribution;
using boost::math::binomial_distribution;
using boost::math::students_t;

// Construct a students_t distribution with 4 degrees of freedom:
students_t d1(4);

// Construct a double-precision beta distribution
// with parameters a = 10, b = 20
beta_distribution<> d2(10, 20); // Note: _distribution<> suffix !
```

If you need to use the distributions with a type other than `double`, then you can instantiate the template directly: the names of the templates are the same as the `double` typedef but with `_distribution` appended, for example: Students t Distribution or Binomial Distribution:

```
// Construct a students_t distribution, of float type,
// with 4 degrees of freedom:
students_t_distribution<float> d3(4);

// Construct a binomial distribution, of long double type,
// with probability of success 0.3
// and 20 trials in total:
binomial_distribution<long double> d4(20, 0.3);
```

The parameters passed to the distributions can be accessed via getter member functions:

```
d1.degrees_of_freedom();  // returns 4.0
```

This is all well and good, but not very useful so far. What we often want is to be able to calculate the *cumulative distribution functions* and *quantiles* etc for these distributions.

Generic operations common to all distributions are non-member functions

Want to calculate the PDF (Probability Density Function) of a distribution? No problem, just use:

```
pdf(my_dist, x);  // Returns PDF (density) at point x of distribution my_dist.
```

Or how about the CDF (Cumulative Distribution Function):

```
cdf(my_dist, x);  // Returns CDF (integral from -infinity to point x)
                  // of distribution my_dist.
```

And quantiles are just the same:

```
quantile(my_dist, p);  // Returns the value of the random variable x
                       // such that cdf(my_dist, x) == p.
```

If you're wondering why these aren't member functions, it's to make the library more easily extensible: if you want to add additional generic operations - let's say the *n'th moment* - then all you have to do is add the appropriate non-member functions, overloaded for each implemented distribution type.

Tip

Random numbers that approximate Quantiles of Distributions

If you want random numbers that are distributed in a specific way, for example in a uniform, normal or triangular, see Boost.Random.

Whilst in principal there's nothing to prevent you from using the quantile function to convert a uniformly distributed random number to another distribution, in practice there are much more efficient algorithms available that are specific to random number generation.

For example, the binomial distribution has two parameters: n (the number of trials) and p (the probability of success on any one trial).

The `binomial_distribution` constructor therefore has two parameters:

```
binomial_distribution(RealType n, RealType p);
```

For this distribution the random variate is k: the number of successes observed. The probability density/mass function (pdf) is therefore written as *f(k; n, p)*.

Note

Random Variates and Distribution Parameters

The concept of a random variable is closely linked to the term random variate: a random variate is a particular value (outcome) of a random variable. and distribution parameters are conventionally distinguished (for example in Wikipedia and Wolfram MathWorld) by placing a semi-colon or vertical bar) *after* the random variable (whose value you 'choose'), to separate the variate from the parameter(s) that defines the shape of the distribution.

For example, the binomial distribution probability distribution function (PDF) is written as *f(k| n, p)* = Pr(K = k|n, p) = probability of observing k successes out of n trials. K is the random variable, k is the random variate, the parameters are n (trials) and p (probability).

Note

By convention, random variate are lower case, usually k is integral, x if real, and random variable are upper case, K if integral, X if real. But this implementation treats all as floating point values `RealType`, so if you really want an integral result, you must round: see note on Discrete Probability Distributions below for details.

As noted above the non-member function `pdf` has one parameter for the distribution object, and a second for the random variate. So taking our binomial distribution example, we would write:

```
pdf(binomial_distribution<RealType>(n, p), k);
```

The ranges of random variate values that are permitted and are supported can be tested by using two functions `range` and `support`.

The distribution (effectively the random variate) is said to be 'supported' over a range that is "the smallest closed set whose complement has probability zero". MathWorld uses the word 'defined' for this range. Non-mathematicians might say it means the 'interesting' smallest range of random variate x that has the cdf going from zero to unity. Outside are uninteresting zones where the pdf is zero, and the cdf zero or unity.

For most distributions, with probability distribution functions one might describe as 'well-behaved', we have decided that it is most useful for the supported range to **exclude** random variate values like exact zero **if the end point is discontinuous**. For example, the Weibull (scale 1, shape 1) distribution smoothly heads for unity as the random variate x declines towards zero. But at x = zero, the value of the pdf is suddenly exactly zero, by definition. If you are plotting the PDF, or otherwise calculating, zero is not the most useful value for the lower limit of supported, as we discovered. So for this, and similar distributions, we have decided it is most numerically useful to use the closest value to zero, min_value, for the limit of the supported range. (The `range` remains from zero, so you will still get `pdf(weibull, 0) == 0`). (Exponential and gamma distributions have similarly discontinuous functions).

Mathematically, the functions may make sense with an (+ or -) infinite value, but except for a few special cases (in the Normal and Cauchy distributions) this implementation limits random variates to finite values from the `max` to `min` for the `RealType`. (See Handling of Floating-Point Infinity for rationale).

Note

Discrete Probability Distributions

Note that the discrete distributions, including the binomial, negative binomial, Poisson & Bernoulli, are all mathematically defined as discrete functions: that is to say the functions `cdf` and `pdf` are only defined for integral values of the random variate.

However, because the method of calculation often uses continuous functions it is convenient to treat them as if they were continuous functions, and permit non-integral values of their parameters.

Users wanting to enforce a strict mathematical model may use `floor` or `ceil` functions on the random variate prior to calling the distribution function.

The quantile functions for these distributions are hard to specify in a manner that will satisfy everyone all of the time. The default behaviour is to return an integer result, that has been rounded *outwards*: that is to say, lower quantiles - where the probablity is less than 0.5 are rounded down, while upper quantiles - where the probability is greater than 0.5 - are rounded up. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on a discrete distribtion. The reference docs describe how to change the rounding policy for these distributions.

For similar reasons continuous distributions with parameters like "degrees of freedom" that might appear to be integral, are treated as real values (and are promoted from integer to floating-point if necessary). In this case however, there are a small number of situations where non-integral degrees of freedom do have a genuine meaning.

Complements are supported too - and when to use them

Often you don't want the value of the CDF, but its complement, which is to say `1-p` rather than `p`. It is tempting to calculate the CDF and subtract it from `1`, but if `p` is very close to `1` then cancellation error will cause you to lose accuracy, perhaps totally.

See below *"Why and when to use complements?"*

In this library, whenever you want to receive a complement, just wrap all the function arguments in a call to `complement(...)`, for example:

```
students_t dist(5);
cout << "CDF at t = 1 is " << cdf(dist, 1.0) << endl;
cout << "Complement of CDF at t = 1 is " << cdf(complement(dist, 1.0)) << endl;
```

But wait, now that we have a complement, we have to be able to use it as well. Any function that accepts a probability as an argument can also accept a complement by wrapping all of its arguments in a call to `complement(...)`, for example:

```
students_t dist(5);

for(double i = 10; i < 1e10; i *= 10)
{
   // Calculate the quantile for a 1 in i chance:
   double t = quantile(complement(dist, 1/i));
   // Print it out:
   cout << "Quantile of students-t with 5 degrees of freedom\n"
           "for a 1 in " << i << " chance is " << t << endl;
}
```

Tip

Critical values are just quantiles

Some texts talk about quantiles, or percentiles or fractiles, others about critical values, the basic rule is:

Lower critical values are the same as the quantile.

Upper critical values are the same as the quantile from the complement of the probability.

For example, suppose we have a Bernoulli process, giving rise to a binomial distribution with success ratio 0.1 and 100 trials in total. The *lower critical value* for a probability of 0.05 is given by:

`quantile(binomial(100, 0.1), 0.05)`

and the *upper critical value* is given by:

`quantile(complement(binomial(100, 0.1), 0.05))`

which return 4.82 and 14.63 respectively.

Tip

Why bother with complements anyway?

It's very tempting to dispense with complements, and simply subtract the probability from 1 when required. However, consider what happens when the probability is very close to 1: let's say the probability expressed at float precision is `0.999999940f`, then `1 - 0.999999940f = 5.96046448e-008`, but the result is actually accurate to just *one single bit*: the only bit that didn't cancel out!

Or to look at this another way: consider that we want the risk of falsely rejecting the null-hypothesis in the Student's t test to be 1 in 1 billion, for a sample size of 10,000. This gives a probability of 1 - 10^{-9}, which is exactly 1 when calculated at float precision. In this case calculating the quantile from the complement neatly solves the problem, so for example:

```
quantile(complement(students_t(10000), 1e-9))
```

returns the expected t-statistic `6.00336`, where as:

```
quantile(students_t(10000), 1-1e-9f)
```

raises an overflow error, since it is the same as:

```
quantile(students_t(10000), 1)
```

Which has no finite result.

With all distributions, even for more reasonable probability (unless the value of p can be represented exactly in the floating-point type) the loss of accuracy quickly becomes significant if you simply calculate probability from 1 - p (because it will be mostly garbage digits for p ~ 1).

So always avoid, for example, using a probability near to unity like 0.99999

```
quantile(my_distribution, 0.99999)
```

and instead use

```
quantile(complement(my_distribution, 0.00001))
```

since 1 - 0.99999 is not exactly equal to 0.00001 when using floating-point arithmetic.

This assumes that the 0.00001 value is either a constant, or can be computed by some manner other than subtracting 0.99999 from 1.

Parameters can be calculated

Sometimes it's the parameters that define the distribution that you need to find. Suppose, for example, you have conducted a Students-t test for equal means and the result is borderline. Maybe your two samples differ from each other, or maybe they don't; based on the result of the test you can't be sure. A legitimate question to ask then is "How many more measurements would I have to take before I would get an X% probability that the difference is real?" Parameter finders can answer questions like this, and are necessarily different for each distribution. They are implemented as static member functions of the distributions, for example:

```
students_t::find_degrees_of_freedom(
   1.3,        // difference from true mean to detect
   0.05,       // maximum risk of falsely rejecting the null-hypothesis.
   0.1,        // maximum risk of falsely failing to reject the null-hypothesis.
   0.13);      // sample standard deviation
```

Returns the number of degrees of freedom required to obtain a 95% probability that the observed differences in means is not down to chance alone. In the case that a borderline Students-t test result was previously obtained, this can be used to estimate how large the sample size would have to become before the observed difference was considered significant. It assumes, of course, that the sample mean and standard deviation are invariant with sample size.

Summary

- Distributions are objects, which are constructed from whatever parameters the distribution may have.
- Member functions allow you to retrieve the parameters of a distribution.
- Generic non-member functions provide access to the properties that are common to all the distributions (PDF, CDF, quantile etc).
- Complements of probabilities are calculated by wrapping the function's arguments in a call to `complement(...)`.
- Functions that accept a probability can accept a complement of the probability as well, by wrapping the function's arguments in a call to `complement(...)`.
- Static member functions allow the parameters of a distribution to be found from other information.

Now that you have the basics, the next section looks at some worked examples.

Worked Examples

Distribution Construction Examples

The structure of distributions is rather different from some other statistical libraries, for example, those written in less object-oriented language like FORTRAN and C that provide a few arguments to each free function.

Boost.Math library instead provides each distribution as a template C++ class. A distribution is constructed with a few arguments, and then member and non-member functions are used to find values of the distribution, often a function of a random variate.

For this demonstration, first we need some includes to access the negative binomial distribution (and the binomial, beta and gamma distributions too).

To demonstrate the use with a high precision User-defined floating-point type `cpp_bin_float`, we also need an include from Boost.Multiprecision. (We could equally well have used a `cpp_dec_float` multiprecision type).

We choose a typedef `cpp_bin_float_50` to provide a 50 decimal digit type, but we could equally have chosen at 128-bit type `cpp_bin_float_quad`, or on some platforms `__float128`, providing about 35 decimal digits.

```
#include <boost/math/distributions/negative_binomial.hpp> // for negative_binomial_distribution
  using boost::math::negative_binomial_distribution; // default type is double.
  using boost::math::negative_binomial; // typedef provides default type is double.
#include <boost/math/distributions/binomial.hpp> // for binomial_distribution.
#include <boost/math/distributions/beta.hpp> // for beta_distribution.
#include <boost/math/distributions/gamma.hpp> // for gamma_distribution.
#include <boost/math/distributions/normal.hpp> // for normal_distribution.

#include <boost/multiprecision/cpp_bin_float.hpp> // for cpp_bin_float_50
```

Several examples of constructing distributions follow:

First, a negative binomial distribution with 8 successes and a success fraction 0.25, 25% or 1 in 4, is constructed like this:

```
boost::math::negative_binomial_distribution<double> mydist0(8., 0.25);
```

But this is inconveniently long, so we might be tempted to write

```
using namespace boost::math;
```

but this might risk ambiguity with names in `std random` so **much** better is explicit `using boost::math::` statements, for example:

```
using boost::math::negative_binomial_distribution;
```

and we can still reduce typing.

Since the vast majority of applications use will be using `double` precision, the template argument to the distribution (`RealType`) defaults to type `double`, so we can also write:

```
negative_binomial_distribution<> mydist9(8., 0.25); // Uses default `RealType = double`.
```

But the name `negative_binomial_distribution` is still inconveniently long, so, for most distributions, a convenience `typedef` is provided, for example:

```
typedef negative_binomial_distribution<double> negative_binomial; // Reserved name of type ↵
double.
```

Caution

This convenience typedef is **not provided** if a clash would occur with the name of a function; currently only `beta` and `gamma` fall into this category.

So, after a using statement,

```
using boost::math::negative_binomial;
```

we have a convenient typedef to `negative_binomial_distribution<double>`:

```
negative_binomial mydist(8., 0.25);
```

Some more examples using the convenience typedef:

```
negative_binomial mydist10(5., 0.4); // Both arguments double.
```

And automatic conversion of arguments takes place, so you can use integers and floats:

```
negative_binomial mydist11(5, 0.4); // Using provided typedef of type double, and int and ↵
double arguments.
```

This is probably the most common usage. Other combination are possible too:

```
negative_binomial mydist12(5., 0.4F); // Double and float arguments.
negative_binomial mydist13(5, 1); // Both arguments integer.
```

Similarly for most other distributions like the binomial.

```
binomial mybinomial(1, 0.5); // is more concise than
binomial_distribution<> mybinomd1(1, 0.5);
```

For cases when the typdef distribution name would clash with a math special function (currently only beta and gamma) the typedef is deliberately not provided, and the longer version of the name must be used, so for example, do not use:

```
using boost::math::beta;
beta mybetad0(1, 0.5); // Error beta is a math FUNCTION!
```

Which produces the error messages:

```
error C2146: syntax error : missing ';' before identifier 'mybetad0'
warning C4551: function call missing argument list
error C3861: 'mybetad0': identifier not found
```

Instead you should use:

```
using boost::math::beta_distribution;
beta_distribution<> mybetad1(1, 0.5);
```

or for the gamma distribution:

```
gamma_distribution<> mygammad1(1, 0.5);
```

We can, of course, still provide the type explicitly thus:

```
// Explicit double precision:  both arguments are double:
negative_binomial_distribution<double>        mydist1(8., 0.25);

// Explicit float precision, double arguments are truncated to float:
negative_binomial_distribution<float>         mydist2(8., 0.25);

// Explicit float precision, integer & double arguments converted to float:
negative_binomial_distribution<float>         mydist3(8, 0.25);

// Explicit float precision, float arguments, so no conversion:
negative_binomial_distribution<float>         mydist4(8.F, 0.25F);

// Explicit float precision, integer arguments promoted to float.
negative_binomial_distribution<float>         mydist5(8, 1);

// Explicit double precision:
negative_binomial_distribution<double>        mydist6(5., 0.4);

// Explicit long double precision:
negative_binomial_distribution<long double>   mydist7(8., 0.25);
```

And you can use your own template RealType, for example, `boost::math::cpp_bin_float_50` (an arbitrary 50 decimal digits precision type), then we can write:

```
 using namespace boost::multiprecision;
 negative_binomial_distribution<cpp_bin_float_50>  mydist8(8, 0.25);

 // `integer` arguments are promoted to your RealType exactly, but
 // `double` argument are converted to RealType,
 // most likely losing precision!

   // So DON'T be tempted to write the 'obvious':
 negative_binomial_distribution<cpp_bin_float_50>  mydist20(8, 0.23456789012345678901234567890);
// to avoid truncation of second parameter to `0.2345678901234567` and loss of precision.

// Instead pass a quoted decimal digit string:
 negative_binomial_distribution<cpp_bin_float_50> ↵
 mydist21(8, cpp_bin_float_50("0.23456789012345678901234567890") );

 // Ensure that all potentially significant digits are shown.
 std::cout.precision(std::numeric_limits<cpp_bin_float_50>::digits10);
   //
 cpp_bin_float_50 x("1.23456789012345678901234567890");
 std::cout << pdf(mydist8, x) << std::endl;
```

```
showing  0.00012630010495970320103876754721976419438231705359935
            0.00012630010495970320103876754721976419438231528547467
```

Warning

When using multiprecision, it is all too easy to get accidental truncation!

For example, if you write

```
std::cout << pdf(mydist8, 1.23456789012345678901234567890) << std::endl;
```

showing 0.00012630010495970318465064569310967179576805651692929, which is wrong at about the 17th decimal digit!

This is because the value provided is truncated to a `double`, effectively `double x = 1.23456789012345678901234567890;`

Then the now `double x` is passed to function `pdf`, and this truncated `double` value is finally promoted to `cpp_bin_float_50`.

Another way of quietly getting the wrong answer is to write:

```
std::cout << pdf(mydist8, cpp_bin_float_50(1.23456789012345678901234567890)) << std::endl;
```

A correct way from a multi-digit string value is

```
std::cout << pdf(mydist8, cpp_bin_float_50("1.23456789012345678901234567890")) << std::endl;
```

Tip

Getting about 17 decimal digits followed by many zeros is often a sign of accidental truncation.

Default arguments to distribution constructors.

Note that default constructor arguments are only provided for some distributions. So if you wrongly assume a default argument, you will get an error message, for example:

```
negative_binomial_distribution<> mydist8;
```

```
error C2512 no appropriate default constructor available.
```

No default constructors are provided for the `negative binomial` distribution, because it is difficult to chose any sensible default values for this distribution.

For other distributions, like the normal distribution, it is obviously very useful to provide 'standard' defaults for the mean (zero) and standard deviation (unity) thus:

```
normal_distribution(RealType mean = 0, RealType sd = 1);
```

So in this case we can more tersely write:

```
  using boost::math::normal;

  normal norm1;       // Standard normal distribution N[0,1].
  normal norm2(2);    // Mean = 2, std deviation = 1.
  normal norm3(2, 3); // Mean = 2, std deviation = 3.

  }
  catch(std::exception &ex)
  {
    std::cout << ex.what() << std::endl;
  }

  return 0;
}  // int main()
```

There is no useful output from this demonstration program, of course.

See distribution_construction.cpp for full source code.

Student's t Distribution Examples

Calculating confidence intervals on the mean with the Students-t distribution

Let's say you have a sample mean, you may wish to know what confidence intervals you can place on that mean. Colloquially: "I want an interval that I can be P% sure contains the true mean". (On a technical point, note that the interval either contains the true mean or it does not: the meaning of the confidence level is subtly different from this colloquialism. More background information can be found on the NIST site).

The formula for the interval can be expressed as:

$$Y_s \pm t_{\left(\frac{\alpha}{2}, N-1\right)} \frac{s}{\sqrt{N}}$$

Where, Y_s is the sample mean, s is the sample standard deviation, N is the sample size, /α/ is the desired significance level and $t_{(\alpha/2,N-1)}$ is the upper critical value of the Students-t distribution with $N-1$ degrees of freedom.

Note

The quantity α is the maximum acceptable risk of falsely rejecting the null-hypothesis. The smaller the value of α the greater the strength of the test.

The confidence level of the test is defined as 1 - α, and often expressed as a percentage. So for example a significance level of 0.05, is equivalent to a 95% confidence level. Refer to "What are confidence intervals?" in NIST/SEMATECH e-Handbook of Statistical Methods. for more information.

Note

The usual assumptions of independent and identically distributed (i.i.d.) variables and normal distribution of course apply here, as they do in other examples.

From the formula, it should be clear that:

- The width of the confidence interval decreases as the sample size increases.
- The width increases as the standard deviation increases.
- The width increases as the *confidence level increases* (0.5 towards 0.99999 - stronger).
- The width increases as the *significance level decreases* (0.5 towards 0.00000...01 - stronger).

The following example code is taken from the example program students_t_single_sample.cpp.

We'll begin by defining a procedure to calculate intervals for various confidence levels; the procedure will print these out as a table:

```
// Needed includes:
#include <boost/math/distributions/students_t.hpp>
#include <iostream>
#include <iomanip>
// Bring everything into global namespace for ease of use:
using namespace boost::math;
using namespace std;

void confidence_limits_on_mean(
   double Sm,           // Sm = Sample Mean.
   double Sd,           // Sd = Sample Standard Deviation.
   unsigned Sn)         // Sn = Sample Size.
{
   using namespace std;
   using namespace boost::math;

   // Print out general info:
   cout <<
      "__________________________________\n"
      "2-Sided Confidence Limits For Mean\n"
      "__________________________________\n\n";
   cout << setprecision(7);
   cout << setw(40) << left << "Number of Observations" << "=  " << Sn << "\n";
   cout << setw(40) << left << "Mean" << "=  " << Sm << "\n";
   cout << setw(40) << left << "Standard Deviation" << "=  " << Sd << "\n";
```

We'll define a table of significance/risk levels for which we'll compute intervals:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Note that these are the complements of the confidence/probability levels: 0.5, 0.75, 0.9 .. 0.99999).

Next we'll declare the distribution object we'll need, note that the *degrees of freedom* parameter is the sample size less one:

```
students_t dist(Sn - 1);
```

Most of what follows in the program is pretty printing, so let's focus on the calculation of the interval. First we need the t-statistic, computed using the *quantile* function and our significance level. Note that since the significance levels are the complement of the probability, we have to wrap the arguments in a call to *complement(...)*:

```
double T = quantile(complement(dist, alpha[i] / 2));
```

Note that alpha was divided by two, since we'll be calculating both the upper and lower bounds: had we been interested in a single sided interval then we would have omitted this step.

Now to complete the picture, we'll get the (one-sided) width of the interval from the t-statistic by multiplying by the standard deviation, and dividing by the square root of the sample size:

```
double w = T * Sd / sqrt(double(Sn));
```

The two-sided interval is then the sample mean plus and minus this width.

And apart from some more pretty-printing that completes the procedure.

Let's take a look at some sample output, first using the Heat flow data from the NIST site. The data set was collected by Bob Zarr of NIST in January, 1990 from a heat flow meter calibration and stability analysis. The corresponding dataplot output for this test can be found in section 3.5.2 of the NIST/SEMATECH e-Handbook of Statistical Methods..

```
__________________________________
2-Sided Confidence Limits For Mean
__________________________________

Number of Observations                  =  195
Mean                                    =  9.26146
Standard Deviation                      =  0.02278881


___________________________________________________________________
Confidence       T           Interval          Lower          Upper
 Value (%)     Value          Width            Limit          Limit
___________________________________________________________________
    50.000     0.676       1.103e-003        9.26036        9.26256
    75.000     1.154       1.883e-003        9.25958        9.26334
    90.000     1.653       2.697e-003        9.25876        9.26416
    95.000     1.972       3.219e-003        9.25824        9.26468
    99.000     2.601       4.245e-003        9.25721        9.26571
    99.900     3.341       5.453e-003        9.25601        9.26691
    99.990     3.973       6.484e-003        9.25498        9.26794
    99.999     4.537       7.404e-003        9.25406        9.26886
```

As you can see the large sample size (195) and small standard deviation (0.023) have combined to give very small intervals, indeed we can be very confident that the true mean is 9.2.

For comparison the next example data output is taken from *P.K.Hou, O. W. Lau & M.C. Wong, Analyst (1983) vol. 108, p 64. and from Statistics for Analytical Chemistry, 3rd ed. (1994), pp 54-55 J. C. Miller and J. N. Miller, Ellis Horwood ISBN 0 13 0309907.* The values result from the determination of mercury by cold-vapour atomic absorption.

```
__________________________________
2-Sided Confidence Limits For Mean
__________________________________

Number of Observations                  =  3
Mean                                    =  37.8000000
Standard Deviation                      =  0.9643650


___________________________________________________________________
Confidence       T           Interval          Lower          Upper
 Value (%)     Value          Width            Limit          Limit
___________________________________________________________________
    50.000     0.816          0.455         37.34539       38.25461
    75.000     1.604          0.893         36.90717       38.69283
    90.000     2.920          1.626         36.17422       39.42578
    95.000     4.303          2.396         35.40438       40.19562
    99.000     9.925          5.526         32.27408       43.32592
    99.900    31.599         17.594         20.20639       55.39361
    99.990    99.992         55.673        -17.87346       93.47346
    99.999   316.225        176.067       -138.26683      213.86683
```

This time the fact that there are only three measurements leads to much wider intervals, indeed such large intervals that it's hard to be very confident in the location of the mean.

Testing a sample mean for difference from a "true" mean

When calibrating or comparing a scientific instrument or measurement method of some kind, we want to be answer the question "Does an observed sample mean differ from the "true" mean in any significant way?". If it does, then we have evidence of a systematic difference. This question can be answered with a Students-t test: more information can be found on the NIST site.

Of course, the assignment of "true" to one mean may be quite arbitrary, often this is simply a "traditional" method of measurement.

The following example code is taken from the example program students_t_single_sample.cpp.

We'll begin by defining a procedure to determine which of the possible hypothesis are rejected or not-rejected at a given significance level:

Note

Non-statisticians might say 'not-rejected' means 'accepted', (often of the null-hypothesis) implying, wrongly, that there really **IS** no difference, but statisticans eschew this to avoid implying that there is positive evidence of 'no difference'. 'Not-rejected' here means there is **no evidence** of difference, but there still might well be a difference. For example, see argument from ignorance and Absence of evidence does not constitute evidence of absence.

```
// Needed includes:
#include <boost/math/distributions/students_t.hpp>
#include <iostream>
#include <iomanip>
// Bring everything into global namespace for ease of use:
using namespace boost::math;
using namespace std;

void single_sample_t_test(double M, double Sm, double Sd, unsigned Sn, double alpha)
{
   //
   // M = true mean.
   // Sm = Sample Mean.
   // Sd = Sample Standard Deviation.
   // Sn = Sample Size.
   // alpha = Significance Level.
```

Most of the procedure is pretty-printing, so let's just focus on the calculation, we begin by calculating the t-statistic:

```
// Difference in means:
double diff = Sm - M;
// Degrees of freedom:
unsigned v = Sn - 1;
// t-statistic:
double t_stat = diff * sqrt(double(Sn)) / Sd;
```

Finally calculate the probability from the t-statistic. If we're interested in simply whether there is a difference (either less or greater) or not, we don't care about the sign of the t-statistic, and we take the complement of the probability for comparison to the significance level:

```
students_t dist(v);
double q = cdf(complement(dist, fabs(t_stat)));
```

The procedure then prints out the results of the various tests that can be done, these can be summarised in the following table:

Hypothesis	**Test**
The Null-hypothesis: there is **no difference** in means	Reject if complement of CDF for |t| < significance level / 2: `cdf(complement(dist, fabs(t))) < alpha / 2`
The Alternative-hypothesis: there **is difference** in means	Reject if complement of CDF for |t| > significance level / 2: `cdf(complement(dist, fabs(t))) > alpha / 2`
The Alternative-hypothesis: the sample mean **is less** than the true mean.	Reject if CDF of t > 1 - significance level: `cdf(complement(dist, t)) < alpha`
The Alternative-hypothesis: the sample mean **is greater** than the true mean.	Reject if complement of CDF of t < significance level: `cdf(dist, t) < alpha`

Note

Notice that the comparisons are against `alpha / 2` for a two-sided test and against `alpha` for a one-sided test

Now that we have all the parts in place, let's take a look at some sample output, first using the Heat flow data from the NIST site. The data set was collected by Bob Zarr of NIST in January, 1990 from a heat flow meter calibration and stability analysis. The corresponding dataplot output for this test can be found in section 3.5.2 of the NIST/SEMATECH e-Handbook of Statistical Methods..

```
__________________________________
Student t test for a single sample
__________________________________

Number of Observations                                 =  195
Sample Mean                                            =  9.26146
Sample Standard Deviation                              =  0.02279
Expected True Mean                                     =  5.00000

Sample Mean - Expected Test Mean                       =  4.26146
Degrees of Freedom                                     =  194
T Statistic                                            =  2611.28380
Probability that difference is due to chance           =  0.000e+000

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis     Conclusion
Mean != 5.000             NOT REJECTED
Mean  < 5.000             REJECTED
Mean  > 5.000             NOT REJECTED
```

You will note the line that says the probability that the difference is due to chance is zero. From a philosophical point of view, of course, the probability can never reach zero. However, in this case the calculated probability is smaller than the smallest representable double precision number, hence the appearance of a zero here. Whatever its "true" value is, we know it must be extraordinarily small, so the alternative hypothesis - that there is a difference in means - is not rejected.

For comparison the next example data output is taken from *P.K.Hou, O. W. Lau & M.C. Wong, Analyst (1983) vol. 108, p 64. and from Statistics for Analytical Chemistry, 3rd ed. (1994), pp 54-55 J. C. Miller and J. N. Miller, Ellis Horwood ISBN 0 13 0309907.* The values result from the determination of mercury by cold-vapour atomic absorption.

```
__________________________________
Student t test for a single sample
__________________________________

Number of Observations                                 =  3
Sample Mean                                            =  37.80000
Sample Standard Deviation                              =  0.96437
Expected True Mean                                     =  38.90000

Sample Mean - Expected Test Mean                       =  -1.10000
Degrees of Freedom                                     =  2
T Statistic                                            =  -1.97566
Probability that difference is due to chance           =  1.869e-001

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis     Conclusion
Mean != 38.900             REJECTED
Mean  < 38.900             NOT REJECTED
Mean  > 38.900             NOT REJECTED
```

As you can see the small number of measurements (3) has led to a large uncertainty in the location of the true mean. So even though there appears to be a difference between the sample mean and the expected true mean, we conclude that there is no significant difference, and are unable to reject the null hypothesis. However, if we were to lower the bar for acceptance down to alpha = 0.1 (a 90% confidence level) we see a different output:

```
__________________________________
Student t test for a single sample
__________________________________

Number of Observations                                 =  3
Sample Mean                                            =  37.80000
Sample Standard Deviation                              =  0.96437
Expected True Mean                                     =  38.90000

Sample Mean - Expected Test Mean                       =  -1.10000
Degrees of Freedom                                     =  2
T Statistic                                            =  -1.97566
Probability that difference is due to chance           =  1.869e-001

Results for Alternative Hypothesis and alpha           =  0.1000

Alternative Hypothesis     Conclusion
Mean != 38.900            REJECTED
Mean  < 38.900            NOT REJECTED
Mean  > 38.900            REJECTED
```

In this case, we really have a borderline result, and more data (and/or more accurate data), is needed for a more convincing conclusion.

Estimating how large a sample size would have to become in order to give a significant Students-t test result with a single sample test

Imagine you have conducted a Students-t test on a single sample in order to check for systematic errors in your measurements. Imagine that the result is borderline. At this point one might go off and collect more data, but it might be prudent to first ask the question "How much more?". The parameter estimators of the students_t_distribution class can provide this information.

This section is based on the example code in students_t_single_sample.cpp and we begin by defining a procedure that will print out a table of estimated sample sizes for various confidence levels:

```
// Needed includes:
#include <boost/math/distributions/students_t.hpp>
#include <iostream>
#include <iomanip>
// Bring everything into global namespace for ease of use:
using namespace boost::math;
using namespace std;

void single_sample_find_df(
   double M,          // M = true mean.
   double Sm,         // Sm = Sample Mean.
   double Sd)         // Sd = Sample Standard Deviation.
{
```

Next we define a table of significance levels:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Printing out the table of sample sizes required for various confidence levels begins with the table header:

```
cout << "\n\n"
        "________________________________________________________________\n"
        "Confidence       Estimated          Estimated\n"
        " Value (%)      Sample Size        Sample Size\n"
        "              (one sided test)    (two sided test)\n"
        "________________________________________________________________\n";
```

And now the important part: the sample sizes required. Class `students_t_distribution` has a static member function `find_degrees_of_freedom` that will calculate how large a sample size needs to be in order to give a definitive result.

The first argument is the difference between the means that you wish to be able to detect, here it's the absolute value of the difference between the sample mean, and the true mean.

Then come two probability values: alpha and beta. Alpha is the maximum acceptable risk of rejecting the null-hypothesis when it is in fact true. Beta is the maximum acceptable risk of failing to reject the null-hypothesis when in fact it is false. Also note that for a two-sided test, alpha must be divided by 2.

The final parameter of the function is the standard deviation of the sample.

In this example, we assume that alpha and beta are the same, and call `find_degrees_of_freedom` twice: once with alpha for a one-sided test, and once with alpha/2 for a two-sided test.

```
   for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
   {
      // Confidence value:
      cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
      // calculate df for single sided test:
      double df = students_t::find_degrees_of_freedom(
         fabs(M - Sm), alpha[i], alpha[i], Sd);
      // convert to sample size:
      double size = ceil(df) + 1;
      // Print size:
      cout << fixed << setprecision(0) << setw(16) << right << size;
      // calculate df for two sided test:
      df = students_t::find_degrees_of_freedom(
         fabs(M - Sm), alpha[i]/2, alpha[i], Sd);
      // convert to sample size:
      size = ceil(df) + 1;
      // Print size:
      cout << fixed << setprecision(0) << setw(16) << right << size << endl;
   }
   cout << endl;
}
```

Let's now look at some sample output using data taken from *P.K.Hou, O. W. Lau & M.C. Wong, Analyst (1983) vol. 108, p 64. and from Statistics for Analytical Chemistry, 3rd ed. (1994), pp 54-55 J. C. Miller and J. N. Miller, Ellis Horwood ISBN 0 13 0309907.* The values result from the determination of mercury by cold-vapour atomic absorption.

Only three measurements were made, and the Students-t test above gave a borderline result, so this example will show us how many samples would need to be collected:

```
_____________________________________________________________
Estimated sample sizes required for various confidence levels
_____________________________________________________________

True Mean                               =  38.90000
Sample Mean                             =  37.80000
Sample Standard Deviation               =  0.96437



_______________________________________________________________
Confidence       Estimated          Estimated
 Value (%)      Sample Size        Sample Size
              (one sided test)    (two sided test)
_______________________________________________________________
    75.000             3                4
    90.000             7                9
    95.000            11               13
    99.000            20               22
    99.900            35               37
    99.990            50               53
    99.999            66               68
```

So in this case, many more measurements would have had to be made, for example at the 95% level, 14 measurements in total for a two-sided test.

Comparing the means of two samples with the Students-t test

Imagine that we have two samples, and we wish to determine whether their means are different or not. This situation often arises when determining whether a new process or treatment is better than an old one.

In this example, we'll be using the Car Mileage sample data from the NIST website. The data compares miles per gallon of US cars with miles per gallon of Japanese cars.

The sample code is in students_t_two_samples.cpp.

There are two ways in which this test can be conducted: we can assume that the true standard deviations of the two samples are equal or not. If the standard deviations are assumed to be equal, then the calculation of the t-statistic is greatly simplified, so we'll examine that case first. In real life we should verify whether this assumption is valid with a Chi-Squared test for equal variances.

We begin by defining a procedure that will conduct our test assuming equal variances:

```
// Needed headers:
#include <boost/math/distributions/students_t.hpp>
#include <iostream>
#include <iomanip>
// Simplify usage:
using namespace boost::math;
using namespace std;

void two_samples_t_test_equal_sd(
        double Sm1,       // Sm1 = Sample 1 Mean.
        double Sd1,       // Sd1 = Sample 1 Standard Deviation.
        unsigned Sn1,     // Sn1 = Sample 1 Size.
        double Sm2,       // Sm2 = Sample 2 Mean.
        double Sd2,       // Sd2 = Sample 2 Standard Deviation.
        unsigned Sn2,     // Sn2 = Sample 2 Size.
        double alpha)     // alpha = Significance Level.
{
```

Our procedure will begin by calculating the t-statistic, assuming equal variances the needed formulae are:

$$
\begin{aligned}
t &= \frac{Sm_1 - Sm_2}{s_p\sqrt{\frac{1}{Sn_1} + \frac{1}{Sn_2}}} \\
s_p &= \sqrt{\frac{(Sn_1 - 1)Sd_1^2 + (Sn_2 - 1)Sd_2^2}{Sn_1 + Sn_2 - 2}} \\
v &= Sn_1 + Sn_2 - 2
\end{aligned}
$$

where Sp is the "pooled" standard deviation of the two samples, and *v* is the number of degrees of freedom of the two combined samples. We can now write the code to calculate the t-statistic:

```
// Degrees of freedom:
double v = Sn1 + Sn2 - 2;
cout << setw(55) << left << "Degrees of Freedom" << "=  " << v << "\n";
// Pooled variance:
double sp = sqrt(((Sn1-1) * Sd1 * Sd1 + (Sn2-1) * Sd2 * Sd2) / v);
cout << setw(55) << left << "Pooled Standard Deviation" << "=  " << sp << "\n";
// t-statistic:
double t_stat = (Sm1 - Sm2) / (sp * sqrt(1.0 / Sn1 + 1.0 / Sn2));
cout << setw(55) << left << "T Statistic" << "=  " << t_stat << "\n";
```

The next step is to define our distribution object, and calculate the complement of the probability:

```
students_t dist(v);
double q = cdf(complement(dist, fabs(t_stat)));
cout << setw(55) << left << "Probability that difference is due to chance" << "=  "
   << setprecision(3) << scientific << 2 * q << "\n\n";
```

Here we've used the absolute value of the t-statistic, because we initially want to know simply whether there is a difference or not (a two-sided test). However, we can also test whether the mean of the second sample is greater or is less (one-sided test) than that of the first: all the possible tests are summed up in the following table:

Hypothesis	**Test**
The Null-hypothesis: there is **no difference** in means	Reject if complement of CDF for |t| < significance level / 2: `cdf(complement(dist, fabs(t))) < alpha / 2`
The Alternative-hypothesis: there is a **difference** in means	Reject if complement of CDF for |t| > significance level / 2: `cdf(complement(dist, fabs(t))) > alpha / 2`
The Alternative-hypothesis: Sample 1 Mean is **less** than Sample 2 Mean.	Reject if CDF of t > significance level: `cdf(dist, t) > alpha`
The Alternative-hypothesis: Sample 1 Mean is **greater** than Sample 2 Mean.	Reject if complement of CDF of t > significance level: `cdf(complement(dist, t)) > alpha`

Note

For a two-sided test we must compare against alpha / 2 and not alpha.

Most of the rest of the sample program is pretty-printing, so we'll skip over that, and take a look at the sample output for alpha=0.05 (a 95% probability level). For comparison the dataplot output for the same data is in section 1.3.5.3 of the NIST/SEMATECH e-Handbook of Statistical Methods..

```
________________________________________________
Student t test for two samples (equal variances)
________________________________________________

Number of Observations (Sample 1)                      =  249
Sample 1 Mean                                          =  20.145
Sample 1 Standard Deviation                            =  6.4147
Number of Observations (Sample 2)                      =  79
Sample 2 Mean                                          =  30.481
Sample 2 Standard Deviation                            =  6.1077
Degrees of Freedom                                     =  326
Pooled Standard Deviation                              =  6.3426
T Statistic                                            =  -12.621
Probability that difference is due to chance           =  5.273e-030

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis              Conclusion
Sample 1 Mean != Sample 2 Mean       NOT REJECTED
Sample 1 Mean <  Sample 2 Mean       NOT REJECTED
Sample 1 Mean >  Sample 2 Mean       REJECTED
```

So with a probability that the difference is due to chance of just 5.273e-030, we can safely conclude that there is indeed a difference.

The tests on the alternative hypothesis show that we must also reject the hypothesis that Sample 1 Mean is greater than that for Sample 2: in this case Sample 1 represents the miles per gallon for Japanese cars, and Sample 2 the miles per gallon for US cars, so we conclude that Japanese cars are on average more fuel efficient.

Now that we have the simple case out of the way, let's look for a moment at the more complex one: that the standard deviations of the two samples are not equal. In this case the formula for the t-statistic becomes:

$$t = \frac{Sm_1 - Sm_2}{\sqrt{\frac{Sd_1^2}{Sn_1} + \frac{Sd_2^2}{Sn_2}}}$$

And for the combined degrees of freedom we use the Welch-Satterthwaite approximation:

$$v = \frac{\left(\frac{Sd_1^2}{Sn_1} + \frac{Sd_2^2}{Sn_2}\right)^2}{\frac{\left(\frac{Sd_1^2}{Sn_1}\right)^2}{(Sn_1 - 1)} + \frac{\left(\frac{Sd_2^2}{Sn_2}\right)^2}{(Sn_2 - 1)}}$$

Note that this is one of the rare situations where the degrees-of-freedom parameter to the Student's t distribution is a real number, and not an integer value.

Note

Some statistical packages truncate the effective degrees of freedom to an integer value: this may be necessary if you are relying on lookup tables, but since our code fully supports non-integer degrees of freedom there is no need to truncate in this case. Also note that when the degrees of freedom is small then the Welch-Satterthwaite approximation may be a significant source of error.

Putting these formulae into code we get:

```
// Degrees of freedom:
double v = Sd1 * Sd1 / Sn1 + Sd2 * Sd2 / Sn2;
v *= v;
double t1 = Sd1 * Sd1 / Sn1;
t1 *= t1;
t1 /=  (Sn1 - 1);
double t2 = Sd2 * Sd2 / Sn2;
t2 *= t2;
t2 /= (Sn2 - 1);
v /= (t1 + t2);
cout << setw(55) << left << "Degrees of Freedom" << "=  " << v << "\n";
// t-statistic:
double t_stat = (Sm1 - Sm2) / sqrt(Sd1 * Sd1 / Sn1 + Sd2 * Sd2 / Sn2);
cout << setw(55) << left << "T Statistic" << "=  " << t_stat << "\n";
```

Thereafter the code and the tests are performed the same as before. Using are car mileage data again, here's what the output looks like:

```
    __________________________________________________
    Student t test for two samples (unequal variances)
    __________________________________________________

    Number of Observations (Sample 1)                      =  249
    Sample 1 Mean                                          =  20.145
    Sample 1 Standard Deviation                            =  6.4147
    Number of Observations (Sample 2)                      =  79
    Sample 2 Mean                                          =  30.481
    Sample 2 Standard Deviation                            =  6.1077
    Degrees of Freedom                                     =  136.87
    T Statistic                                            =  -12.946
    Probability that difference is due to chance           =  1.571e-025

    Results for Alternative Hypothesis and alpha           =  0.0500

    Alternative Hypothesis              Conclusion
    Sample 1 Mean != Sample 2 Mean       NOT REJECTED
    Sample 1 Mean <  Sample 2 Mean       NOT REJECTED
    Sample 1 Mean >  Sample 2 Mean       REJECTED
```

This time allowing the variances in the two samples to differ has yielded a higher likelihood that the observed difference is down to chance alone (1.571e-025 compared to 5.273e-030 when equal variances were assumed). However, the conclusion remains the same: US cars are less fuel efficient than Japanese models.

Comparing two paired samples with the Student's t distribution

Imagine that we have a before and after reading for each item in the sample: for example we might have measured blood pressure before and after administration of a new drug. We can't pool the results and compare the means before and after the change, because each patient will have a different baseline reading. Instead we calculate the difference between before and after measurements in each patient, and calculate the mean and standard deviation of the differences. To test whether a significant change has taken place, we can then test the null-hypothesis that the true mean is zero using the same procedure we used in the single sample cases previously discussed.

That means we can:

- Calculate confidence intervals of the mean. If the endpoints of the interval differ in sign then we are unable to reject the null-hypothesis that there is no change.
- Test whether the true mean is zero. If the result is consistent with a true mean of zero, then we are unable to reject the null-hypothesis that there is no change.

- Calculate how many pairs of readings we would need in order to obtain a significant result.

Chi Squared Distribution Examples

Confidence Intervals on the Standard Deviation

Once you have calculated the standard deviation for your data, a legitimate question to ask is "How reliable is the calculated standard deviation?". For this situation the Chi Squared distribution can be used to calculate confidence intervals for the standard deviation.

The full example code & sample output is in chi_square_std_dev_test.cpp.

We'll begin by defining the procedure that will calculate and print out the confidence intervals:

```
void confidence_limits_on_std_deviation(
     double Sd,    // Sample Standard Deviation
     unsigned N)   // Sample size
{
```

We'll begin by printing out some general information:

```
cout <<
   "________________________________________________\n"
   "2-Sided Confidence Limits For Standard Deviation\n"
   "________________________________________________\n\n";
cout << setprecision(7);
cout << setw(40) << left << "Number of Observations" << "=  " << N << "\n";
cout << setw(40) << left << "Standard Deviation" << "=  " << Sd << "\n";
```

and then define a table of significance levels for which we'll calculate intervals:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

The distribution we'll need to calculate the confidence intervals is a Chi Squared distribution, with N-1 degrees of freedom:

```
chi_squared dist(N - 1);
```

For each value of alpha, the formula for the confidence interval is given by:

$$\sqrt{\frac{(N-1)s^2}{\chi^2_{\left(\frac{\alpha}{2},N-1\right)}}} \quad \le \quad \sigma \quad \le \quad \sqrt{\frac{(N-1)s^2}{\chi^2_{\left(1-\frac{\alpha}{2},N-1\right)}}}$$

Where

$$\chi^2_{\left(\frac{\alpha}{2},N-1\right)}$$

is the upper critical value, and

$$\chi^2_{\left(1-\frac{\alpha}{2},N-1\right)}$$

is the lower critical value of the Chi Squared distribution.

In code we begin by printing out a table header:

```
cout << "\n\n"
        "____________________________________________\n"
        "Confidence           Lower          Upper\n"
        " Value (%)           Limit          Limit\n"
        "____________________________________________\n";
```

and then loop over the values of alpha and calculate the intervals for each: remember that the lower critical value is the same as the quantile, and the upper critical value is the same as the quantile from the complement of the probability:

```
for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
   // Confidence value:
   cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
   // Calculate limits:
   double lower_limit = sqrt((N - 1) * Sd * Sd / quantile(complement(dist, alpha[i] / 2)));
   double upper_limit = sqrt((N - 1) * Sd * Sd / quantile(dist, alpha[i] / 2));
   // Print Limits:
   cout << fixed << setprecision(5) << setw(15) << right << lower_limit;
   cout << fixed << setprecision(5) << setw(15) << right << upper_limit << endl;
}
cout << endl;
```

To see some example output we'll use the gear data from the NIST/SEMATECH e-Handbook of Statistical Methods.. The data represents measurements of gear diameter from a manufacturing process.

```
________________________________________________
2-Sided Confidence Limits For Standard Deviation
________________________________________________

Number of Observations                  =  100
Standard Deviation                      =  0.006278908



_____________________________________________
Confidence          Lower          Upper
 Value (%)          Limit          Limit
_____________________________________________
    50.000        0.00601        0.00662
    75.000        0.00582        0.00685
    90.000        0.00563        0.00712
    95.000        0.00551        0.00729
    99.000        0.00530        0.00766
    99.900        0.00507        0.00812
    99.990        0.00489        0.00855
    99.999        0.00474        0.00895
```

So at the 95% confidence level we conclude that the standard deviation is between 0.00551 and 0.00729.

Confidence intervals as a function of the number of observations

Similarly, we can also list the confidence intervals for the standard deviation for the common confidence levels 95%, for increasing numbers of observations.

The standard deviation used to compute these values is unity, so the limits listed are **multipliers** for any particular standard deviation. For example, given a standard deviation of 0.0062789 as in the example above; for 100 observations the multiplier is 0.8780 giving the lower confidence limit of 0.8780 * 0.006728 = 0.00551.

```
___________________________________________________
Confidence level (two-sided)            =  0.0500000
Standard Deviation                      =  1.0000000
________________________________________
Observations        Lower          Upper
                    Limit          Limit
________________________________________
         2         0.4461        31.9102
         3         0.5207         6.2847
         4         0.5665         3.7285
         5         0.5991         2.8736
         6         0.6242         2.4526
         7         0.6444         2.2021
         8         0.6612         2.0353
         9         0.6755         1.9158
        10         0.6878         1.8256
        15         0.7321         1.5771
        20         0.7605         1.4606
        30         0.7964         1.3443
        40         0.8192         1.2840
        50         0.8353         1.2461
        60         0.8476         1.2197
       100         0.8780         1.1617
       120         0.8875         1.1454
      1000         0.9580         1.0459
     10000         0.9863         1.0141
     50000         0.9938         1.0062
    100000         0.9956         1.0044
   1000000         0.9986         1.0014
```

With just 2 observations the limits are from **0.445** up to to **31.9**, so the standard deviation might be about **half** the observed value up to **30 times** the observed value!

Estimating a standard deviation with just a handful of values leaves a very great uncertainty, especially the upper limit. Note especially how far the upper limit is skewed from the most likely standard deviation.

Even for 10 observations, normally considered a reasonable number, the range is still from 0.69 to 1.8, about a range of 0.7 to 2, and is still highly skewed with an upper limit **twice** the median.

When we have 1000 observations, the estimate of the standard deviation is starting to look convincing, with a range from 0.95 to 1.05 - now near symmetrical, but still about + or - 5%.

Only when we have 10000 or more repeated observations can we start to be reasonably confident (provided we are sure that other factors like drift are not creeping in).

For 10000 observations, the interval is 0.99 to 1.1 - finally a really convincing + or -1% confidence.

Chi-Square Test for the Standard Deviation

We use this test to determine whether the standard deviation of a sample differs from a specified value. Typically this occurs in process change situations where we wish to compare the standard deviation of a new process to an established one.

The code for this example is contained in chi_square_std_dev_test.cpp, and we'll begin by defining the procedure that will print out the test statistics:

```
void chi_squared_test(
    double Sd,      // Sample std deviation
    double D,       // True std deviation
    unsigned N,     // Sample size
    double alpha)  // Significance level
{
```

The procedure begins by printing a summary of the input data:

```
using namespace std;
using namespace boost::math;

// Print header:
cout <<
   "______________________________________________\n"
   "Chi Squared test for sample standard deviation\n"
   "______________________________________________\n\n";
cout << setprecision(5);
cout << setw(55) << left << "Number of Observations" << "=  " << N << "\n";
cout << setw(55) << left << "Sample Standard Deviation" << "=  " << Sd << "\n";
cout << setw(55) << left << "Expected True Standard Deviation" << "=  " << D << "\n\n";
```

The test statistic (T) is simply the ratio of the sample and "true" standard deviations squared, multiplied by the number of degrees of freedom (the sample size less one):

```
double t_stat = (N - 1) * (Sd / D) * (Sd / D);
cout << setw(55) << left << "Test Statistic" << "=  " << t_stat << "\n";
```

The distribution we need to use, is a Chi Squared distribution with N-1 degrees of freedom:

```
chi_squared dist(N - 1);
```

The various hypothesis that can be tested are summarised in the following table:

Hypothesis	**Test**
The null-hypothesis: there is no difference in standard deviation from the specified value	Reject if T < $\chi^2_{(1\text{-alpha/2; N-1})}$ or T > $\chi^2_{(\text{alpha/2; N-1})}$
The alternative hypothesis: there is a difference in standard deviation from the specified value	Reject if $\chi^2_{(1\text{-alpha/2; N-1})}$ >= T >= $\chi^2_{(\text{alpha/2; N-1})}$
The alternative hypothesis: the standard deviation is less than the specified value	Reject if $\chi^2_{(1\text{-alpha; N-1})}$ <= T
The alternative hypothesis: the standard deviation is greater than the specified value	Reject if $\chi^2_{(\text{alpha; N-1})}$ >= T

Where $\chi^2_{(\text{alpha; N-1})}$ is the upper critical value of the Chi Squared distribution, and $\chi^2_{(1\text{-alpha; N-1})}$ is the lower critical value.

Recall that the lower critical value is the same as the quantile, and the upper critical value is the same as the quantile from the complement of the probability, that gives us the following code to calculate the critical values:

```
double ucv = quantile(complement(dist, alpha));
double ucv2 = quantile(complement(dist, alpha / 2));
double lcv = quantile(dist, alpha);
double lcv2 = quantile(dist, alpha / 2);
cout << setw(55) << left << "Upper Critical Value at alpha: " << "=  "
   << setprecision(3) << scientific << ucv << "\n";
cout << setw(55) << left << "Upper Critical Value at alpha/2: " << "=  "
   << setprecision(3) << scientific << ucv2 << "\n";
cout << setw(55) << left << "Lower Critical Value at alpha: " << "=  "
   << setprecision(3) << scientific << lcv << "\n";
cout << setw(55) << left << "Lower Critical Value at alpha/2: " << "=  "
   << setprecision(3) << scientific << lcv2 << "\n\n";
```

Now that we have the critical values, we can compare these to our test statistic, and print out the result of each hypothesis and test:

```
cout << setw(55) << left <<
   "Results for Alternative Hypothesis and alpha" << "=  "
   << setprecision(4) << fixed << alpha << "\n\n";
cout << "Alternative Hypothesis              Conclusion\n";

cout << "Standard Deviation != " << setprecision(3) << fixed << D << "             ";
if((ucv2 < t_stat) || (lcv2 > t_stat))
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";

cout << "Standard Deviation  < " << setprecision(3) << fixed << D << "             ";
if(lcv > t_stat)
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";

cout << "Standard Deviation  > " << setprecision(3) << fixed << D << "             ";
if(ucv < t_stat)
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";
cout << endl << endl;
```

To see some example output we'll use the gear data from the NIST/SEMATECH e-Handbook of Statistical Methods.. The data represents measurements of gear diameter from a manufacturing process. The program output is deliberately designed to mirror the DATAPLOT output shown in the NIST Handbook Example.

```
_____________________________________________
Chi Squared test for sample standard deviation
_____________________________________________

Number of Observations                                 =  100
Sample Standard Deviation                              =  0.00628
Expected True Standard Deviation                       =  0.10000

Test Statistic                                         =  0.39030
CDF of test statistic:                                 =  1.438e-099
Upper Critical Value at alpha:                         =  1.232e+002
Upper Critical Value at alpha/2:                       =  1.284e+002
Lower Critical Value at alpha:                         =  7.705e+001
Lower Critical Value at alpha/2:                       =  7.336e+001

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis              Conclusion
Standard Deviation != 0.100            ACCEPTED
Standard Deviation  < 0.100            ACCEPTED
Standard Deviation  > 0.100            REJECTED
```

In this case we are testing whether the sample standard deviation is 0.1, and the null-hypothesis is rejected, so we conclude that the standard deviation *is not* 0.1.

For an alternative example, consider the silicon wafer data again from the NIST/SEMATECH e-Handbook of Statistical Methods.. In this scenario a supplier of 100 ohm.cm silicon wafers claims that his fabrication process can produce wafers with sufficient consistency so that the standard deviation of resistivity for the lot does not exceed 10 ohm.cm. A sample of N = 10 wafers taken from the lot has a standard deviation of 13.97 ohm.cm, and the question we ask ourselves is "Is the suppliers claim correct?".

The program output now looks like this:

```
_____________________________________________
Chi Squared test for sample standard deviation
_____________________________________________

Number of Observations                                 =  10
Sample Standard Deviation                              =  13.97000
Expected True Standard Deviation                       =  10.00000

Test Statistic                                         =  17.56448
CDF of test statistic:                                 =  9.594e-001
Upper Critical Value at alpha:                         =  1.692e+001
Upper Critical Value at alpha/2:                       =  1.902e+001
Lower Critical Value at alpha:                         =  3.325e+000
Lower Critical Value at alpha/2:                       =  2.700e+000

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis              Conclusion
Standard Deviation != 10.000            REJECTED
Standard Deviation  < 10.000            REJECTED
Standard Deviation  > 10.000            ACCEPTED
```

In this case, our null-hypothesis is that the standard deviation of the sample is less than 10: this hypothesis is rejected in the analysis above, and so we reject the manufacturers claim.

Estimating the Required Sample Sizes for a Chi-Square Test for the Standard Deviation

Suppose we conduct a Chi Squared test for standard deviation and the result is borderline, a legitimate question to ask is "How large would the sample size have to be in order to produce a definitive result?"

The class template chi_squared_distribution has a static method `find_degrees_of_freedom` that will calculate this value for some acceptable risk of type I failure *alpha*, type II failure *beta*, and difference from the standard deviation *diff*. Please note that the method used works on variance, and not standard deviation as is usual for the Chi Squared Test.

The code for this example is located in chi_square_std_dev_test.cpp.

We begin by defining a procedure to print out the sample sizes required for various risk levels:

```
void chi_squared_sample_sized(
     double diff,      // difference from variance to detect
     double variance)  // true variance
{
```

The procedure begins by printing out the input data:

```
using namespace std;
using namespace boost::math;

// Print out general info:
cout <<
   "_____________________________________________________________\n"
   "Estimated sample sizes required for various confidence levels\n"
   "_____________________________________________________________\n\n";
cout << setprecision(5);
cout << setw(40) << left << "True Variance" << "=  " << variance << "\n";
cout << setw(40) << left << "Difference to detect" << "=  " << diff << "\n";
```

And defines a table of significance levels for which we'll calculate sample sizes:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

For each value of alpha we can calculate two sample sizes: one where the sample variance is less than the true value by *diff* and one where it is greater than the true value by *diff*. Thanks to the asymmetric nature of the Chi Squared distribution these two values will not be the same, the difference in their calculation differs only in the sign of *diff* that's passed to `find_degrees_of_freedom`. Finally in this example we'll simply things, and let risk level *beta* be the same as *alpha*:

```
cout << "\n\n"
        "_______________________________________________________________\n"
        "Confidence       Estimated          Estimated\n"
        " Value (%)      Sample Size        Sample Size\n"
        "                (lower one         (upper one\n"
        "                 sided test)        sided test)\n"
        "_______________________________________________________________\n";
//
// Now print out the data for the table rows.
//
for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
   // Confidence value:
   cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
   // calculate df for a lower single sided test:
   double df = chi_squared::find_degrees_of_freedom(
      -diff, alpha[i], alpha[i], variance);
   // convert to sample size:
   double size = ceil(df) + 1;
   // Print size:
   cout << fixed << setprecision(0) << setw(16) << right << size;
   // calculate df for an upper single sided test:
   df = chi_squared::find_degrees_of_freedom(
      diff, alpha[i], alpha[i], variance);
   // convert to sample size:
   size = ceil(df) + 1;
   // Print size:
   cout << fixed << setprecision(0) << setw(16) << right << size << endl;
}
cout << endl;
```

For some example output, consider the silicon wafer data from the NIST/SEMATECH e-Handbook of Statistical Methods.. In this scenario a supplier of 100 ohm.cm silicon wafers claims that his fabrication process can produce wafers with sufficient consistency so that the standard deviation of resistivity for the lot does not exceed 10 ohm.cm. A sample of N = 10 wafers taken from the lot has a standard deviation of 13.97 ohm.cm, and the question we ask ourselves is "How large would our sample have to be to reliably detect this difference?".

To use our procedure above, we have to convert the standard deviations to variance (square them), after which the program output looks like this:

```
_____________________________________________________________
Estimated sample sizes required for various confidence levels
_____________________________________________________________

True Variance                             =  100.00000
Difference to detect                      =  95.16090



_______________________________________________________________
Confidence       Estimated          Estimated
 Value (%)      Sample Size        Sample Size
                (lower one         (upper one
                 sided test)        sided test)
_______________________________________________________________
    50.000                2                 2
    75.000                2                10
    90.000                4                32
    95.000                5                51
    99.000                7                99
    99.900               11               174
    99.990               15               251
    99.999               20               330
```

In this case we are interested in a upper single sided test. So for example, if the maximum acceptable risk of falsely rejecting the null-hypothesis is 0.05 (Type I error), and the maximum acceptable risk of failing to reject the null-hypothesis is also 0.05 (Type II error), we estimate that we would need a sample size of 51.

F Distribution Examples

Imagine that you want to compare the standard deviations of two sample to determine if they differ in any significant way, in this situation you use the F distribution and perform an F-test. This situation commonly occurs when conducting a process change comparison: "is a new process more consistent that the old one?".

In this example we'll be using the data for ceramic strength from http://www.itl.nist.gov/div898/handbook/eda/section4/eda42a1.htm. The data for this case study were collected by Said Jahanmir of the NIST Ceramics Division in 1996 in connection with a NIST/industry ceramics consortium for strength optimization of ceramic strength.

The example program is f_test.cpp, program output has been deliberately made as similar as possible to the DATAPLOT output in the corresponding NIST EngineeringStatistics Handbook example.

We'll begin by defining the procedure to conduct the test:

```
void f_test(
    double sd1,     // Sample 1 std deviation
    double sd2,     // Sample 2 std deviation
    double N1,      // Sample 1 size
    double N2,      // Sample 2 size
    double alpha)  // Significance level
{
```

The procedure begins by printing out a summary of our input data:

```
using namespace std;
using namespace boost::math;

// Print header:
cout <<
   "_____________________________________\n"
   "F test for equal standard deviations\n"
   "_____________________________________\n\n";
cout << setprecision(5);
cout << "Sample 1:\n";
cout << setw(55) << left << "Number of Observations" << "=  " << N1 << "\n";
cout << setw(55) << left << "Sample Standard Deviation" << "=  " << sd1 << "\n\n";
cout << "Sample 2:\n";
cout << setw(55) << left << "Number of Observations" << "=  " << N2 << "\n";
cout << setw(55) << left << "Sample Standard Deviation" << "=  " << sd2 << "\n\n";
```

The test statistic for an F-test is simply the ratio of the square of the two standard deviations:

$F = s_1^2 / s_2^2$

where s_1 is the standard deviation of the first sample and s_2 is the standard deviation of the second sample. Or in code:

```
double F = (sd1 / sd2);
F *= F;
cout << setw(55) << left << "Test Statistic" << "=  " << F << "\n\n";
```

At this point a word of caution: the F distribution is asymmetric, so we have to be careful how we compute the tests, the following table summarises the options available:

Hypothesis	**Test**
The null-hypothesis: there is no difference in standard deviations (two sided test)	Reject if $F \le F_{(1-alpha/2;\ N1-1,\ N2-1)}$ or $F \ge F_{(alpha/2;\ N1-1,\ N2-1)}$
The alternative hypothesis: there is a difference in means (two sided test)	Reject if $F_{(1-alpha/2;\ N1-1,\ N2-1)} \le F \le F_{(alpha/2;\ N1-1,\ N2-1)}$
The alternative hypothesis: Standard deviation of sample 1 is greater than that of sample 2	Reject if $F < F_{(alpha;\ N1-1,\ N2-1)}$
The alternative hypothesis: Standard deviation of sample 1 is less than that of sample 2	Reject if $F > F_{(1-alpha;\ N1-1,\ N2-1)}$

Where $F_{(1-alpha;\ N1-1,\ N2-1)}$ is the lower critical value of the F distribution with degrees of freedom N1-1 and N2-1, and $F_{(alpha;\ N1-1,\ N2-1)}$ is the upper critical value of the F distribution with degrees of freedom N1-1 and N2-1.

The upper and lower critical values can be computed using the quantile function:

$F_{(1-alpha;\ N1-1,\ N2-1)}$ = `quantile(fisher_f(N1-1, N2-1), alpha)`

$F_{(alpha;\ N1-1,\ N2-1)}$ = `quantile(complement(fisher_f(N1-1, N2-1), alpha))`

In our example program we need both upper and lower critical values for alpha and for alpha/2:

```
double ucv = quantile(complement(dist, alpha));
double ucv2 = quantile(complement(dist, alpha / 2));
double lcv = quantile(dist, alpha);
double lcv2 = quantile(dist, alpha / 2);
cout << setw(55) << left << "Upper Critical Value at alpha: " << "=  "
   << setprecision(3) << scientific << ucv << "\n";
cout << setw(55) << left << "Upper Critical Value at alpha/2: " << "=  "
   << setprecision(3) << scientific << ucv2 << "\n";
cout << setw(55) << left << "Lower Critical Value at alpha: " << "=  "
   << setprecision(3) << scientific << lcv << "\n";
cout << setw(55) << left << "Lower Critical Value at alpha/2: " << "=  "
   << setprecision(3) << scientific << lcv2 << "\n\n";
```

The final step is to perform the comparisons given above, and print out whether the hypothesis is rejected or not:

```
cout << setw(55) << left <<
   "Results for Alternative Hypothesis and alpha" << "=  "
   << setprecision(4) << fixed << alpha << "\n\n";
cout << "Alternative Hypothesis                                 Conclusion\n";

cout << "Standard deviations are unequal (two sided test)          ";
if((ucv2 < F) || (lcv2 > F))
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";

cout << "Standard deviation 1 is less than standard deviation 2    ";
if(lcv > F)
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";

cout << "Standard deviation 1 is greater than standard deviation 2 ";
if(ucv < F)
   cout << "ACCEPTED\n";
else
   cout << "REJECTED\n";
cout << endl << endl;
```

Using the ceramic strength data as an example we get the following output:

```
F test for equal standard deviations
____________________________________

Sample 1:
Number of Observations                                 =  240
Sample Standard Deviation                              =  65.549

Sample 2:
Number of Observations                                 =  240
Sample Standard Deviation                              =  61.854

Test Statistic                                         =  1.123

CDF of test statistic:                                 =  8.148e-001
Upper Critical Value at alpha:                         =  1.238e+000
Upper Critical Value at alpha/2:                       =  1.289e+000
Lower Critical Value at alpha:                         =  8.080e-001
Lower Critical Value at alpha/2:                       =  7.756e-001

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis                                    Conclusion
Standard deviations are unequal (two sided test)          REJECTED
Standard deviation 1 is less than standard deviation 2    REJECTED
Standard deviation 1 is greater than standard deviation 2 REJECTED
```

In this case we are unable to reject the null-hypothesis, and must instead reject the alternative hypothesis.

By contrast let's see what happens when we use some different sample data:, once again from the NIST Engineering Statistics Handbook: A new procedure to assemble a device is introduced and tested for possible improvement in time of assembly. The question being addressed is whether the standard deviation of the new assembly process (sample 2) is better (i.e., smaller) than the standard deviation for the old assembly process (sample 1).

```
____________________________________
F test for equal standard deviations
____________________________________

Sample 1:
Number of Observations                                  =  11.00000
Sample Standard Deviation                               =  4.90820

Sample 2:
Number of Observations                                  =  9.00000
Sample Standard Deviation                               =  2.58740

Test Statistic                                          =  3.59847

CDF of test statistic:                                  =  9.589e-001
Upper Critical Value at alpha:                          =  3.347e+000
Upper Critical Value at alpha/2:                        =  4.295e+000
Lower Critical Value at alpha:                          =  3.256e-001
Lower Critical Value at alpha/2:                        =  2.594e-001

Results for Alternative Hypothesis and alpha            =  0.0500

Alternative Hypothesis                                     Conclusion
Standard deviations are unequal (two sided test)           REJECTED
Standard deviation 1 is less than standard deviation 2     REJECTED
Standard deviation 1 is greater than standard deviation 2 ACCEPTED
```

In this case we take our null hypothesis as "standard deviation 1 is less than or equal to standard deviation 2", since this represents the "no change" situation. So we want to compare the upper critical value at *alpha* (a one sided test) with the test statistic, and since 3.35 < 3.6 this hypothesis must be rejected. We therefore conclude that there is a change for the better in our standard deviation.

Binomial Distribution Examples

See also the reference documentation for the Binomial Distribution.

Binomial Coin-Flipping Example

An example of a Bernoulli process is coin flipping. A variable in such a sequence may be called a Bernoulli variable.

This example shows using the Binomial distribution to predict the probability of heads and tails when throwing a coin.

The number of correct answers (say heads), X, is distributed as a binomial random variable with binomial distribution parameters number of trials (flips) n = 10 and probability (success_fraction) of getting a head p = 0.5 (a 'fair' coin).

(Our coin is assumed fair, but we could easily change the success_fraction parameter p from 0.5 to some other value to simulate an unfair coin, say 0.6 for one with chewing gum on the tail, so it is more likely to fall tails down and heads up).

First we need some includes and using statements to be able to use the binomial distribution, some std input and output, and get started:

```
#include <boost/math/distributions/binomial.hpp>
  using boost::math::binomial;

#include <iostream>
  using std::cout;  using std::endl;  using std::left;
#include <iomanip>
  using std::setw;

int main()
{
  cout << "Using Binomial distribution to predict how many heads and tails." << endl;
  try
  {
```

See note with the catch block about why a try and catch block is always a good idea.

First, construct a binomial distribution with parameters success_fraction 1/2, and how many flips.

```
const double success_fraction = 0.5; // = 50% = 1/2 for a 'fair' coin.
int flips = 10;
binomial flip(flips, success_fraction);

cout.precision(4);
```

Then some examples of using Binomial moments (and echoing the parameters).

```
cout << "From " << flips << " one can expect to get on average "
  << mean(flip) << " heads (or tails)." << endl;
cout << "Mode is " << mode(flip) << endl;
cout << "Standard deviation is " << standard_deviation(flip) << endl;
cout << "So about 2/3 will lie within 1 standard deviation and get between "
  <<  ceil(mean(flip) - standard_deviation(flip))  << " and "
  << floor(mean(flip) + standard_deviation(flip)) << " correct." << endl;
cout << "Skewness is " << skewness(flip) << endl;
// Skewness of binomial distributions is only zero (symmetrical)
// if success_fraction is exactly one half,
// for example, when flipping 'fair' coins.
cout << "Skewness if success_fraction is " << flip.success_fraction()
  << " is " << skewness(flip) << endl << endl; // Expect zero for a 'fair' coin.
```

Now we show a variety of predictions on the probability of heads:

```
cout << "For " << flip.trials() << " coin flips: " << endl;
cout << "Probability of getting no heads is " << pdf(flip, 0) << endl;
cout << "Probability of getting at least one head is " << 1. - pdf(flip, 0) << endl;
```

When we want to calculate the probability for a range or values we can sum the PDF's:

```
cout << "Probability of getting 0 or 1 heads is "
  << pdf(flip, 0) + pdf(flip, 1) << endl; // sum of exactly == probabilities
```

Or we can use the cdf.

```
cout << "Probability of getting 0 or 1 (<= 1) heads is " << cdf(flip, 1) << endl;
cout << "Probability of getting 9 or 10 heads is " << pdf(flip, 9) + pdf(flip, 10) << endl;
```

Note that using

```
cout << "Probability of getting 9 or 10 heads is " << 1. - cdf(flip, 8) << endl;
```

is less accurate than using the complement

```
cout << "Probability of getting 9 or 10 heads is " << cdf(complement(flip, 8)) << endl;
```

Since the subtraction may involve cancellation error, where as `cdf(complement(flip, 8))` does not use such a subtraction internally, and so does not exhibit the problem.

To get the probability for a range of heads, we can either add the pdfs for each number of heads

```
cout << "Probability of between 4 and 6 heads (4 or 5 or 6) is "
  //  P(X == 4) + P(X == 5) + P(X == 6)
  << pdf(flip, 4) + pdf(flip, 5) + pdf(flip, 6) << endl;
```

But this is probably less efficient than using the cdf

```
cout << "Probability of between 4 and 6 heads (4 or 5 or 6) is "
  // P(X <= 6) - P(X <= 3) == P(X < 4)
  << cdf(flip, 6) - cdf(flip, 3) << endl;
```

Certainly for a bigger range like, 3 to 7

```
cout << "Probability of between 3 and 7 heads (3, 4, 5, 6 or 7) is "
  // P(X <= 7) - P(X <= 2) == P(X < 3)
  << cdf(flip, 7) - cdf(flip, 2) << endl;
cout << endl;
```

Finally, print two tables of probability for the *exactly* and *at least* a number of heads.

```
// Print a table of probability for the exactly a number of heads.
cout << "Probability of getting exactly (==) heads" << endl;
for (int successes = 0; successes <= flips; successes++)
{ // Say success means getting a head (or equally success means getting a tail).
  double probability = pdf(flip, successes);
  cout << left << setw(2) << successes << "     " << setw(10)
    << probability << " or 1 in " << 1. / probability
    << ", or " << probability * 100. << "%" << endl;
} // for i
cout << endl;

// Tabulate the probability of getting between zero heads and 0 upto 10 heads.
cout << "Probability of getting upto (<=) heads" << endl;
for (int successes = 0; successes <= flips; successes++)
{ // Say success means getting a head
  // (equally success could mean getting a tail).
  double probability = cdf(flip, successes); // P(X <= heads)
  cout << setw(2) << successes << "        " << setw(10) << left
    << probability << " or 1 in " << 1. / probability << ", or "
    << probability * 100. << "%"<< endl;
} // for i
```

The last (0 to 10 heads) must, of course, be 100% probability.

```
      double probability = 0.3;
      double q = quantile(flip, probability);
     std::cout << "Quantile (flip, " << probability << ") = " << q << std::endl; // Quantile ↵
(flip, 0.3) = 3
      probability = 0.6;
      q = quantile(flip, probability);
     std::cout << "Quantile (flip, " << probability << ") = " << q << std::endl; // Quantile ↵
(flip, 0.6) = 5







}
catch(const std::exception& e)
{
  //
```

It is always essential to include try & catch blocks because default policies are to throw exceptions on arguments that are out of domain or cause errors like numeric-overflow.

Lacking try & catch blocks, the program will abort, whereas the message below from the thrown exception will give some helpful clues as to the cause of the problem.

```
  std::cout <<
    "\n""Message from thrown exception was:\n   " << e.what() << std::endl;
}
```

See binomial_coinflip_example.cpp for full source code, the program output looks like this:

```
Using Binomial distribution to predict how many heads and tails.
From 10 one can expect to get on average 5 heads (or tails).
Mode is 5
Standard deviation is 1.581
So about 2/3 will lie within 1 standard deviation and get between 4 and 6 correct.
Skewness is 0
Skewness if success_fraction is 0.5 is 0

For 10 coin flips:
Probability of getting no heads is 0.0009766
Probability of getting at least one head is 0.999
Probability of getting 0 or 1 heads is 0.01074
Probability of getting 0 or 1 (<= 1) heads is 0.01074
Probability of getting 9 or 10 heads is 0.01074
Probability of getting 9 or 10 heads is 0.01074
Probability of getting 9 or 10 heads is 0.01074
Probability of between 4 and 6 heads (4 or 5 or 6) is 0.6562
Probability of between 4 and 6 heads (4 or 5 or 6) is 0.6563
Probability of between 3 and 7 heads (3, 4, 5, 6 or 7) is 0.8906

Probability of getting exactly (==) heads
0      0.0009766  or 1 in 1024, or 0.09766%
1      0.009766   or 1 in 102.4, or 0.9766%
2      0.04395    or 1 in 22.76, or 4.395%
3      0.1172     or 1 in 8.533, or 11.72%
4      0.2051     or 1 in 4.876, or 20.51%
5      0.2461     or 1 in 4.063, or 24.61%
6      0.2051     or 1 in 4.876, or 20.51%
7      0.1172     or 1 in 8.533, or 11.72%
8      0.04395    or 1 in 22.76, or 4.395%
9      0.009766   or 1 in 102.4, or 0.9766%
10     0.0009766  or 1 in 1024, or 0.09766%

Probability of getting upto (<=) heads
0         0.0009766  or 1 in 1024, or 0.09766%
1         0.01074    or 1 in 93.09, or 1.074%
2         0.05469    or 1 in 18.29, or 5.469%
3         0.1719     or 1 in 5.818, or 17.19%
4         0.377      or 1 in 2.653, or 37.7%
5         0.623      or 1 in 1.605, or 62.3%
6         0.8281     or 1 in 1.208, or 82.81%
7         0.9453     or 1 in 1.058, or 94.53%
8         0.9893     or 1 in 1.011, or 98.93%
9         0.999      or 1 in 1.001, or 99.9%
10        1          or 1 in 1, or 100%
```

Binomial Quiz Example

A multiple choice test has four possible answers to each of 16 questions. A student guesses the answer to each question, so the probability of getting a correct answer on any given question is one in four, a quarter, 1/4, 25% or fraction 0.25. The conditions of the binomial experiment are assumed to be met: n = 16 questions constitute the trials; each question results in one of two possible outcomes (correct or incorrect); the probability of being correct is 0.25 and is constant if no knowledge about the subject is assumed; the questions are answered independently if the student's answer to a question in no way influences his/her answer to another question.

First, we need to be able to use the binomial distribution constructor (and some std input/output, of course).

```
#include <boost/math/distributions/binomial.hpp>
  using boost::math::binomial;

#include <iostream>
  using std::cout; using std::endl;
  using std::ios; using std::flush; using std::left; using std::right; using std::fixed;
#include <iomanip>
  using std::setw; using std::setprecision;
#include <exception>
```

The number of correct answers, X, is distributed as a binomial random variable with binomial distribution parameters: questions n and success fraction probability p. So we construct a binomial distribution:

```
int questions = 16; // All the questions in the quiz.
int answers = 4; // Possible answers to each question.
double success_fraction = 1. / answers; // If a random guess, p = 1/4 = 0.25.
binomial quiz(questions, success_fraction);
```

and display the distribution parameters we used thus:

```
cout << "In a quiz with " << quiz.trials()
  << " questions and with a probability of guessing right of "
  << quiz.success_fraction() * 100 << " %"
  << " or 1 in " << static_cast<int>(1. / quiz.success_fraction()) << endl;
```

Show a few probabilities of just guessing:

```
cout << "Probability of getting none right is " << pdf(quiz, 0) << endl; // 0.010023
cout << "Probability of getting exactly one right is " << pdf(quiz, 1) << endl;
cout << "Probability of getting exactly two right is " << pdf(quiz, 2) << endl;
int pass_score = 11;
cout << "Probability of getting exactly " << pass_score << " answers right by chance is "
  << pdf(quiz, pass_score) << endl;
cout << "Probability of getting all " << questions << " answers right by chance is "
  << pdf(quiz, questions) << endl;
```

```
Probability of getting none right is 0.0100226
Probability of getting exactly one right is 0.0534538
Probability of getting exactly two right is 0.133635
Probability of getting exactly 11 right is 0.000247132
Probability of getting exactly all 16 answers right by chance is 2.32831e-010
```

These don't give any encouragement to guessers!

We can tabulate the 'getting exactly right' (==) probabilities thus:

```
cout << "\n" "Guessed Probability" << right << endl;
for (int successes = 0; successes <= questions; successes++)
{
  double probability = pdf(quiz, successes);
  cout << setw(2) << successes << "      " << probability << endl;
}
cout << endl;
```

```
Guessed Probability
 0      0.0100226
 1      0.0534538
 2      0.133635
 3      0.207876
 4      0.225199
 5      0.180159
 6      0.110097
 7      0.0524273
 8      0.0196602
 9      0.00582526
10      0.00135923
11      0.000247132
12      3.43239e-005
13      3.5204e-006
14      2.51457e-007
15      1.11759e-008
16      2.32831e-010
```

Then we can add the probabilities of some 'exactly right' like this:

```
cout << "Probability of getting none or one right is " << pdf(quiz, 0) + pdf(quiz, 1) << endl;
```

```
Probability of getting none or one right is 0.0634764
```

But if more than a couple of scores are involved, it is more convenient (and may be more accurate) to use the Cumulative Distribution Function (cdf) instead:

```
cout << "Probability of getting none or one right is " << cdf(quiz, 1) << endl;
```

```
Probability of getting none or one right is 0.0634764
```

Since the cdf is inclusive, we can get the probability of getting up to 10 right (<=)

```
cout << "Probability of getting <= 10 right (to fail) is " << cdf(quiz, 10) << endl;
```

```
Probability of getting <= 10 right (to fail) is 0.999715
```

To get the probability of getting 11 or more right (to pass), it is tempting to use

```
1 - cdf(quiz, 10)
```

to get the probability of > 10

```
cout << "Probability of getting > 10 right (to pass) is " << 1 - cdf(quiz, 10) << endl;
```

```
Probability of getting > 10 right (to pass) is 0.000285239
```

But this should be resisted in favor of using the complements function (see why complements?).

```
cout << "Probability of getting > 10 right (to pass) is " << cdf(complement(quiz, 10)) << endl;
```

```
Probability of getting > 10 right (to pass) is 0.000285239
```

And we can check that these two, <= 10 and > 10, add up to unity.

```
BOOST_ASSERT((cdf(quiz, 10) + cdf(complement(quiz, 10))) == 1.);
```

If we want a < rather than a <= test, because the CDF is inclusive, we must subtract one from the score.

```
cout << "Probability of getting less than " << pass_score
  << " (< " << pass_score << ") answers right by guessing is "
  << cdf(quiz, pass_score -1) << endl;
```

```
Probability of getting less than 11 (< 11) answers right by guessing is 0.999715
```

and similarly to get a >= rather than a > test we also need to subtract one from the score (and can again check the sum is unity). This is because if the cdf is *inclusive*, then its complement must be *exclusive* otherwise there would be one possible outcome counted twice!

```
cout << "Probability of getting at least " << pass_score
  << "(>= " << pass_score << ") answers right by guessing is "
  << cdf(complement(quiz, pass_score-1))
  << ", only 1 in " << 1/cdf(complement(quiz, pass_score-1)) << endl;

BOOST_ASSERT((cdf(quiz, pass_score -1) + cdf(complement(quiz, pass_score-1))) == 1);
```

```
Probability of getting at least 11 (>= 11) answers right by guessing is 0.000285239, only 1 ↲
in 3505.83
```

Finally we can tabulate some probabilities:

```
cout << "\n" "At most (<=)""\n""Guessed OK   Probability" << right << endl;
for (int score = 0; score <= questions; score++)
{
  cout << setw(2) << score << "           " << setprecision(10)
    << cdf(quiz, score) << endl;
}
cout << endl;
```

```
At most (<=)
Guessed OK   Probability
 0           0.01002259576
 1           0.0634764398
 2           0.1971110499
 3           0.4049871101
 4           0.6301861752
 5           0.8103454274
 6           0.9204427481
 7           0.9728700437
 8           0.9925302796
 9           0.9983555346
10           0.9997147608
11           0.9999618928
12           0.9999962167
13           0.9999997371
14           0.9999999886
15           0.9999999998
16           1
```

```
cout << "\n" "At least (>)""\n""Guessed OK   Probability" << right << endl;
for (int score = 0; score <= questions; score++)
{
  cout << setw(2) << score << "           "  << setprecision(10)
    << cdf(complement(quiz, score)) << endl;
}
```

```
At least (>)
Guessed OK   Probability
 0           0.9899774042
 1           0.9365235602
 2           0.8028889501
 3           0.5950128899
 4           0.3698138248
 5           0.1896545726
 6           0.07955725188
 7           0.02712995629
 8           0.00746972044
 9           0.001644465374
10           0.0002852391917
11           3.810715862e-005
12           3.783265129e-006
13           2.628657967e-007
14           1.140870154e-008
15           2.328306437e-010
16           0
```

We now consider the probabilities of **ranges** of correct guesses.

First, calculate the probability of getting a range of guesses right, by adding the exact probabilities of each from low ... high.

```
int low = 3; // Getting at least 3 right.
int high = 5; // Getting as most 5 right.
double sum = 0.;
for (int i = low; i <= high; i++)
{
  sum += pdf(quiz, i);
}
cout.precision(4);
cout << "Probability of getting between "
  << low << " and " << high << " answers right by guessing is "
  << sum  << endl; // 0.61323
```

```
Probability of getting between 3 and 5 answers right by guessing is 0.6132
```

Or, usually better, we can use the difference of cdfs instead:

```
cout << "Probability of getting between " << low << " and " << high << " answers right by guess↲
ing is "
  <<  cdf(quiz, high) - cdf(quiz, low - 1) << endl; // 0.61323
```

```
Probability of getting between 3 and 5 answers right by guessing is 0.6132
```

And we can also try a few more combinations of high and low choices:

```
low = 1; high = 6;
cout << "Probability of getting between " << low << " and " << high << " answers right by guess↲
ing is "
  <<  cdf(quiz, high) - cdf(quiz, low - 1) << endl; // 1 and 6 P= 0.91042
low = 1; high = 8;
cout << "Probability of getting between " << low << " and " << high << " answers right by guess↲
ing is "
  <<  cdf(quiz, high) - cdf(quiz, low - 1) << endl; // 1 <= x 8 P = 0.9825
low = 4; high = 4;
cout << "Probability of getting between " << low << " and " << high << " answers right by guess↲
ing is "
  <<  cdf(quiz, high) - cdf(quiz, low - 1) << endl; // 4 <= x 4 P = 0.22520
```

```
Probability of getting between 1 and 6 answers right by guessing is 0.9104
Probability of getting between 1 and 8 answers right by guessing is 0.9825
Probability of getting between 4 and 4 answers right by guessing is 0.2252
```

Using Binomial distribution moments

Using moments of the distribution, we can say more about the spread of results from guessing.

```
cout << "By guessing, on average, one can expect to get " << mean(quiz) << " correct an↲
swers." << endl;
cout << "Standard deviation is " << standard_deviation(quiz) << endl;
cout << "So about 2/3 will lie within 1 standard deviation and get between "
  <<  ceil(mean(quiz) - standard_deviation(quiz))  << " and "
  << floor(mean(quiz) + standard_deviation(quiz)) << " correct." << endl;
cout << "Mode (the most frequent) is " << mode(quiz) << endl;
cout << "Skewness is " << skewness(quiz) << endl;
```

```
By guessing, on average, one can expect to get 4 correct answers.
Standard deviation is 1.732
So about 2/3 will lie within 1 standard deviation and get between 3 and 5 correct.
Mode (the most frequent) is 4
Skewness is 0.2887
```

Quantiles

The quantiles (percentiles or percentage points) for a few probability levels:

```
cout << "Quartiles " << quantile(quiz, 0.25) << " to "
  << quantile(complement(quiz, 0.25)) << endl; // Quartiles
cout << "1 standard deviation " << quantile(quiz, 0.33) << " to "
  << quantile(quiz, 0.67) << endl; // 1 sd
cout << "Deciles " << quantile(quiz, 0.1)  << " to "
  << quantile(complement(quiz, 0.1))<< endl; // Deciles
cout << "5 to 95% " << quantile(quiz, 0.05)  << " to "
  << quantile(complement(quiz, 0.05))<< endl; // 5 to 95%
cout << "2.5 to 97.5% " << quantile(quiz, 0.025) << " to "
  <<  quantile(complement(quiz, 0.025)) << endl; // 2.5 to 97.5%
cout << "2 to 98% " << quantile(quiz, 0.02)  << " to "
  << quantile(complement(quiz, 0.02)) << endl; //  2 to 98%

cout << "If guessing then percentiles 1 to 99% will get " << quantile(quiz, 0.01)
  << " to " << quantile(complement(quiz, 0.01)) << " right." << endl;
```

Notice that these output integral values because the default policy is `integer_round_outwards`.

```
Quartiles 2 to 5
1 standard deviation 2 to 5
Deciles 1 to 6
5 to 95% 0 to 7
2.5 to 97.5% 0 to 8
2 to 98% 0 to 8
```

Quantiles values are controlled by the understanding discrete quantiles quantile policy chosen. The default is `integer_round_outwards`, so the lower quantile is rounded down, and the upper quantile is rounded up.

But we might believe that the real values tell us a little more - see discrete functions.

We could control the policy for **all** distributions by

```
#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real

at the head of the program would make this policy apply
```

to this **one, and only**, translation unit.

Or we can now create a (typedef for) policy that has discrete quantiles real (here avoiding any 'using namespaces ...' statements):

```
using boost::math::policies::policy;
using boost::math::policies::discrete_quantile;
using boost::math::policies::real;
using boost::math::policies::integer_round_outwards; // Default.
typedef boost::math::policies::policy<discrete_quantile<real> > real_quantile_policy;
```

Add a custom binomial distribution called

```
real_quantile_binomial
```

that uses

```
real_quantile_policy
```

```
using boost::math::binomial_distribution;
typedef binomial_distribution<double, real_quantile_policy> real_quantile_binomial;
```

Construct an object of this custom distribution:

```
real_quantile_binomial quiz_real(questions, success_fraction);
```

And use this to show some quantiles - that now have real rather than integer values.

```
cout << "Quartiles " << quantile(quiz, 0.25) << " to "
  << quantile(complement(quiz_real, 0.25)) << endl; // Quartiles 2 to 4.6212
cout << "1 standard deviation " << quantile(quiz_real, 0.33) << " to "
  << quantile(quiz_real, 0.67) << endl; // 1 sd 2.6654 4.194
cout << "Deciles " << quantile(quiz_real, 0.1)  << " to "
  << quantile(complement(quiz_real, 0.1))<< endl; // Deciles 1.3487 5.7583
cout << "5 to 95% " << quantile(quiz_real, 0.05)  << " to "
  << quantile(complement(quiz_real, 0.05))<< endl; // 5 to 95% 0.83739 6.4559
cout << "2.5 to 97.5% " << quantile(quiz_real, 0.025) << " to "
  <<  quantile(complement(quiz_real, 0.025)) << endl; // 2.5 to 97.5% 0.42806 7.0688
cout << "2 to 98% " << quantile(quiz_real, 0.02)  << " to "
  << quantile(complement(quiz_real, 0.02)) << endl; //  2 to 98% 0.31311 7.7880

cout << "If guessing, then percentiles 1 to 99% will get " << quantile(quiz_real, 0.01)
  << " to " << quantile(complement(quiz_real, 0.01)) << " right." << endl;
```

```
Real Quantiles
Quartiles 2 to 4.621
1 standard deviation 2.665 to 4.194
Deciles 1.349 to 5.758
5 to 95% 0.8374 to 6.456
2.5 to 97.5% 0.4281 to 7.069
2 to 98% 0.3131 to 7.252
If guessing then percentiles 1 to 99% will get 0 to 7.788 right.
```

See binomial_quiz_example.cpp for full source code and output.

Calculating Confidence Limits on the Frequency of Occurrence for a Binomial Distribution

Imagine you have a process that follows a binomial distribution: for each trial conducted, an event either occurs or does it does not, referred to as "successes" and "failures". If, by experiment, you want to measure the frequency with which successes occur, the best estimate is given simply by *k* / *N*, for *k* successes out of *N* trials. However our confidence in that estimate will be shaped by how many trials were conducted, and how many successes were observed. The static member functions `binomial_distri-`

bution<>::find_lower_bound_on_p and binomial_distribution<>::find_upper_bound_on_p allow you to calculate the confidence intervals for your estimate of the occurrence frequency.

The sample program binomial_confidence_limits.cpp illustrates their use. It begins by defining a procedure that will print a table of confidence limits for various degrees of certainty:

```
#include <iostream>
#include <iomanip>
#include <boost/math/distributions/binomial.hpp>

void confidence_limits_on_frequency(unsigned trials, unsigned successes)
{
   //
   // trials = Total number of trials.
   // successes = Total number of observed successes.
   //
   // Calculate confidence limits for an observed
   // frequency of occurrence that follows a binomial
   // distribution.
   //
   using namespace std;
   using namespace boost::math;

   // Print out general info:
   cout <<
      "___________________________________________\n"
      "2-Sided Confidence Limits For Success Ratio\n"
      "___________________________________________\n\n";
   cout << setprecision(7);
   cout << setw(40) << left << "Number of Observations" << "=  " << trials << "\n";
   cout << setw(40) << left << "Number of successes" << "=  " << successes << "\n";
   cout << setw(40) << left << "Sample frequency of occurrence" << "=  " << double(suc↵
cesses) / trials << "\n";
```

The procedure now defines a table of significance levels: these are the probabilities that the true occurrence frequency lies outside the calculated interval:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Some pretty printing of the table header follows:

```
cout << "\n\n"
        "_______________________________________________________________________\n"
        "Confidence        Lower CP       Upper CP       Lower JP       Upper JP\n"
        " Value (%)        Limit          Limit          Limit          Limit\n"
        "_______________________________________________________________________\n";
```

And now for the important part - the intervals themselves - for each value of *alpha*, we call find_lower_bound_on_p and find_lower_upper_on_p to obtain lower and upper bounds respectively. Note that since we are calculating a two-sided interval, we must divide the value of alpha in two.

Please note that calculating two separate *single sided bounds*, each with risk level α is not the same thing as calculating a two sided interval. Had we calculate two single-sided intervals each with a risk that the true value is outside the interval of α, then:

- The risk that it is less than the lower bound is α.

and

- The risk that it is greater than the upper bound is also α.

So the risk it is outside **upper or lower bound**, is **twice** alpha, and the probability that it is inside the bounds is therefore not nearly as high as one might have thought. This is why α/2 must be used in the calculations below.

In contrast, had we been calculating a single-sided interval, for example: *"Calculate a lower bound so that we are P% sure that the true occurrence frequency is greater than some value"* then we would **not** have divided by two.

Finally note that `binomial_distribution` provides a choice of two methods for the calculation, we print out the results from both methods in this example:

```
   for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
   {
      // Confidence value:
      cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
      // Calculate Clopper Pearson bounds:
      double l = binomial_distribution<>::find_lower_bound_on_p(
                     trials, successes, alpha[i]/2);
      double u = binomial_distribution<>::find_upper_bound_on_p(
                     trials, successes, alpha[i]/2);
      // Print Clopper Pearson Limits:
      cout << fixed << setprecision(5) << setw(15) << right << l;
      cout << fixed << setprecision(5) << setw(15) << right << u;
      // Calculate Jeffreys Prior Bounds:
      l = binomial_distribution<>::find_lower_bound_on_p(
            trials, successes, alpha[i]/2,
            binomial_distribution<>::jeffreys_prior_interval);
      u = binomial_distribution<>::find_upper_bound_on_p(
            trials, successes, alpha[i]/2,
            binomial_distribution<>::jeffreys_prior_interval);
      // Print Jeffreys Prior Limits:
      cout << fixed << setprecision(5) << setw(15) << right << l;
      cout << fixed << setprecision(5) << setw(15) << right << u << std::endl;
   }
   cout << endl;
}
```

And that's all there is to it. Let's see some sample output for a 2 in 10 success ratio, first for 20 trials:

```
___________________________________________
2-Sided Confidence Limits For Success Ratio
___________________________________________

Number of Observations                  =  20
Number of successes                     =  4
Sample frequency of occurrence          =  0.2


_______________________________________________________________________
Confidence        Lower CP       Upper CP       Lower JP       Upper JP
 Value (%)        Limit          Limit          Limit          Limit
_______________________________________________________________________
    50.000        0.12840        0.29588        0.14974        0.26916
    75.000        0.09775        0.34633        0.11653        0.31861
    90.000        0.07135        0.40103        0.08734        0.37274
    95.000        0.05733        0.43661        0.07152        0.40823
    99.000        0.03576        0.50661        0.04655        0.47859
    99.900        0.01905        0.58632        0.02634        0.55960
    99.990        0.01042        0.64997        0.01530        0.62495
    99.999        0.00577        0.70216        0.00901        0.67897
```

As you can see, even at the 95% confidence level the bounds are really quite wide (this example is chosen to be easily compared to the one in the NIST/SEMATECH e-Handbook of Statistical Methods. here). Note also that the Clopper-Pearson calculation method (CP above) produces quite noticeably more pessimistic estimates than the Jeffreys Prior method (JP above).

Compare that with the program output for 2000 trials:

```
___________________________________________
2-Sided Confidence Limits For Success Ratio
___________________________________________

Number of Observations                  =  2000
Number of successes                     =  400
Sample frequency of occurrence          =  0.2000000


_______________________________________________________________________
Confidence        Lower CP       Upper CP       Lower JP       Upper JP
 Value (%)        Limit          Limit          Limit          Limit
_______________________________________________________________________
    50.000        0.19382        0.20638        0.19406        0.20613
    75.000        0.18965        0.21072        0.18990        0.21047
    90.000        0.18537        0.21528        0.18561        0.21503
    95.000        0.18267        0.21821        0.18291        0.21796
    99.000        0.17745        0.22400        0.17769        0.22374
    99.900        0.17150        0.23079        0.17173        0.23053
    99.990        0.16658        0.23657        0.16681        0.23631
    99.999        0.16233        0.24169        0.16256        0.24143
```

Now even when the confidence level is very high, the limits are really quite close to the experimentally calculated value of 0.2. Furthermore the difference between the two calculation methods is now really quite small.

Estimating Sample Sizes for a Binomial Distribution.

Imagine you have a critical component that you know will fail in 1 in N "uses" (for some suitable definition of "use"). You may want to schedule routine replacement of the component so that its chance of failure between routine replacements is less than P%. If the failures follow a binomial distribution (each time the component is "used" it either fails or does not) then the static member function `binomial_distibution<>::find_maximum_number_of_trials` can be used to estimate the maximum number of "uses" of that component for some acceptable risk level *alpha*.

The example program binomial_sample_sizes.cpp demonstrates its usage. It centres on a routine that prints out a table of maximum sample sizes for various probability thresholds:

```
void find_max_sample_size(
   double p,              // success ratio.
   unsigned successes)    // Total number of observed successes permitted.
{
```

The routine then declares a table of probability thresholds: these are the maximum acceptable probability that *successes* or fewer events will be observed. In our example, *successes* will be always zero, since we want no component failures, but in other situations non-zero values may well make sense.

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Much of the rest of the program is pretty-printing, the important part is in the calculation of maximum number of permitted trials for each value of alpha:

```
for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
   // Confidence value:
   cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
   // calculate trials:
   double t = binomial::find_maximum_number_of_trials(
                  successes, p, alpha[i]);
   t = floor(t);
   // Print Trials:
   cout << fixed << setprecision(5) << setw(15) << right << t << endl;
}
```

Note that since we're calculating the maximum number of trials permitted, we'll err on the safe side and take the floor of the result. Had we been calculating the *minimum* number of trials required to observe a certain number of *successes* using `find_minimum_number_of_trials` we would have taken the ceiling instead.

We'll finish off by looking at some sample output, firstly for a 1 in 1000 chance of component failure with each use:

```
________________________
Maximum Number of Trials
________________________

Success ratio                          =  0.001
Maximum Number of "successes" permitted =  0



____________________________
Confidence        Max Number
 Value (%)        Of Trials
____________________________
    50.000          692
    75.000          287
    90.000          105
    95.000           51
    99.000           10
    99.900            0
    99.990            0
    99.999            0
```

So 51 "uses" of the component would yield a 95% chance that no component failures would be observed.

Compare that with a 1 in 1 million chance of component failure:

```
________________________
Maximum Number of Trials
________________________

Success ratio                        =  0.0000010
Maximum Number of "successes" permitted =  0



____________________________
Confidence        Max Number
 Value (%)        Of Trials
____________________________
    50.000          693146
    75.000          287681
    90.000          105360
    95.000           51293
    99.000           10050
    99.900            1000
    99.990             100
    99.999              10
```

In this case, even 1000 uses of the component would still yield a less than 1 in 1000 chance of observing a component failure (i.e. a 99.9% chance of no failure).

Geometric Distribution Examples

For this example, we will opt to #define two macros to control the error and discrete handling policies. For this simple example, we want to avoid throwing an exception (the default policy) and just return infinity. We want to treat the distribution as if it was continuous, so we choose a discrete_quantile policy of real, rather than the default policy integer_round_outwards.

```
#define BOOST_MATH_OVERFLOW_ERROR_POLICY ignore_error
#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real
```

Caution

It is vital to #include distributions etc **after** the above #defines

After that we need some includes to provide easy access to the negative binomial distribution, and we need some std library iostream, of course.

```
#include <boost/math/distributions/geometric.hpp>
  // for geometric_distribution
  using ::boost::math::geometric_distribution; //
  using ::boost::math::geometric; // typedef provides default type is double.
  using  ::boost::math::pdf; // Probability mass function.
  using  ::boost::math::cdf; // Cumulative density function.
  using  ::boost::math::quantile;

#include <boost/math/distributions/negative_binomial.hpp>
  // for negative_binomial_distribution
  using boost::math::negative_binomial; // typedef provides default type is double.

#include <boost/math/distributions/normal.hpp>
  // for negative_binomial_distribution
  using boost::math::normal; // typedef provides default type is double.

#include <iostream>
  using std::cout; using std::endl;
  using std::noshowpoint; using std::fixed; using std::right; using std::left;
#include <iomanip>
  using std::setprecision; using std::setw;

#include <limits>
  using std::numeric_limits;
```

It is always sensible to use try and catch blocks because defaults policies are to throw an exception if anything goes wrong.

Simple try'n'catch blocks (see below) will ensure that you get a helpful error message instead of an abrupt (and silent) program abort.

Throwing a dice

The Geometric distribution describes the probability (*p*) of a number of failures to get the first success in *k* Bernoulli trials. (A Bernoulli trial is one with only two possible outcomes, success of failure, and *p* is the probability of success).

Suppose an 'fair' 6-face dice is thrown repeatedly:

```
double success_fraction = 1./6; // success_fraction (p) = 0.1666
// (so failure_fraction is 1 - success_fraction = 5./6 = 1- 0.1666 = 0.8333)
```

If the dice is thrown repeatedly until the **first** time a *three* appears. The probablility distribution of the number of times it is thrown **not** getting a *three* (*not-a-threes* number of failures to get a *three*) is a geometric distribution with the success_fraction = 1/6 = 0.1666 .

We therefore start by constructing a geometric distribution with the one parameter success_fraction, the probability of success.

```
geometric g6(success_fraction); // type double by default.
```

To confirm, we can echo the success_fraction parameter of the distribution.

```
cout << "success fraction of a six-sided dice is " << g6.success_fraction() << endl;
```

So the probability of getting a three at the first throw (zero failures) is

```
cout << pdf(g6, 0) << endl; // 0.1667
cout << cdf(g6, 0) << endl; // 0.1667
```

Note that the cdf and pdf are identical because the is only one throw. If we want the probability of getting the first *three* on the 2nd throw:

```
cout << pdf(g6, 1) << endl; // 0.1389
```

If we want the probability of getting the first *three* on the 1st or 2nd throw (allowing one failure):

```
cout << "pdf(g6, 0) + pdf(g6, 1) = " << pdf(g6, 0) + pdf(g6, 1) << endl;
```

Or more conveniently, and more generally, we can use the Cumulative Distribution Function CDF.

```
cout << "cdf(g6, 1) = " << cdf(g6, 1) << endl; // 0.3056
```

If we allow many more (12) throws, the probability of getting our *three* gets very high:

```
cout << "cdf(g6, 12) = " << cdf(g6, 12) << endl; // 0.9065 or 90% probability.
```

If we want to be much more confident, say 99%, we can estimate the number of throws to be this sure using the inverse or quantile.

```
cout << "quantile(g6, 0.99) = " << quantile(g6, 0.99) << endl; // 24.26
```

Note that the value returned is not an integer: if you want an integer result you should use either floor, round or ceil functions, or use the policies mechanism.

See understanding discrete quantiles.

The geometric distribution is related to the negative binomial `negative_binomial_distribution(RealType r, RealType p);` with parameter *r* = 1. So we could get the same result using the negative binomial, but using the geometric the results will be faster, and may be more accurate.

```
negative_binomial nb(1, success_fraction);
cout << pdf(nb, 1) << endl; // 0.1389
cout << cdf(nb, 1) << endl; // 0.3056
```

We could also the complement to express the required probability as 1 - 0.99 = 0.01 (and get the same result):

```
cout << "quantile(complement(g6, 1 - p))  " << quantile(complement(g6, 0.01)) << endl; // 24.26
```

Note too that Boost.Math geometric distribution is implemented as a continuous function. Unlike other implementations (for example R) it **uses** the number of failures as a **real** parameter, not as an integer. If you want this integer behaviour, you may need to enforce this by rounding the parameter you pass, probably rounding down, to the nearest integer. For example, R returns the success fraction probability for all values of failures from 0 to 0.999999 thus:

```
   R> formatC(pgeom(0.0001,0.5, FALSE), digits=17) "               0.5"
```

So in Boost.Math the equivalent is

```
    geometric g05(0.5);  // Probability of success = 0.5 or 50%
    // Output all potentially significant digits for the type, here double.

#ifdef BOOST_NO_CXX11_NUMERIC_LIMITS
  int max_digits10 = 2 + (boost::math::policies::di↵
gits<double, boost::math::policies::policy<> >() * 30103UL) / 100000UL;
  cout << "BOOST_NO_CXX11_NUMERIC_LIMITS is defined" << endl;
#else
  int max_digits10 = std::numeric_limits<double>::max_digits10;
#endif
  cout << "Show all potentially significant decimal digits std::numeric_limits<double>::max_di↵
gits10 = "
    << max_digits10 << endl;
  cout.precision(max_digits10); //

    cout << cdf(g05, 0.0001) << endl; // returns 0.5000346561579232, not exact 0.5.
```

To get the R discrete behaviour, you simply need to round with, for example, the `floor` function.

```
cout << cdf(g05, floor(0.0001)) << endl; // returns exactly 0.5
```

```
> formatC(pgeom(0.9999999,0.5, FALSE), digits=17) [1] "               0.25"
> formatC(pgeom(1.999999,0.5, FALSE), digits=17)[1] "               0.25" k = 1
> formatC(pgeom(1.9999999,0.5, FALSE), digits=17)[1] "0.12500000000000003" k = 2
```

shows that R makes an arbitrary round-up decision at about 1e7 from the next integer above. This may be convenient in practice, and could be replicated in C++ if desired.

Surveying customers to find one with a faulty product

A company knows from warranty claims that 2% of their products will be faulty, so the 'success_fraction' of finding a fault is 0.02. It wants to interview a purchaser of faulty products to assess their 'user experience'.

To estimate how many customers they will probably need to contact in order to find one who has suffered from the fault, we first construct a geometric distribution with probability 0.02, and then chose a confidence, say 80%, 95%, or 99% to finding a customer with a fault. Finally, we probably want to round up the result to the integer above using the `ceil` function. (We could also use a policy, but that is hardly worthwhile for this simple application.)

(This also assumes that each customer only buys one product: if customers bought more than one item, the probability of finding a customer with a fault obviously improves.)

```
cout.precision(5);
geometric g(0.02); // On average, 2 in 100 products are faulty.
double c = 0.95; // 95% confidence.
cout << " quantile(g, " << c << ") = " << quantile(g, c) << endl;

cout << "To be " << c * 100
  << "% confident of finding we customer with a fault, need to survey "
  <<  ceil(quantile(g, c)) << " customers." << endl; // 148
c = 0.99; // Very confident.
cout << "To be " << c * 100
  << "% confident of finding we customer with a fault, need to survey "
  <<  ceil(quantile(g, c)) << " customers." << endl; // 227
c = 0.80; // Only reasonably confident.
cout << "To be " << c * 100
  << "% confident of finding we customer with a fault, need to survey "
  <<  ceil(quantile(g, c)) << " customers." << endl; // 79
```

Basket Ball Shooters

According to Wikipedia, average pro basket ball players get free throws in the baskets 70 to 80 % of the time, but some get as high as 95%, and others as low as 50%. Suppose we want to compare the probabilities of failing to get a score only on the first or on the fifth shot? To start we will consider the average shooter, say 75%. So we construct a geometric distribution with success_fraction parameter 75/100 = 0.75.

```
cout.precision(2);
geometric gav(0.75); // Shooter averages 7.5 out of 10 in the basket.
```

What is probability of getting 1st try in the basket, that is with no failures?

```
cout << "Probability of score on 1st try = " << pdf(gav, 0) << endl; // 0.75
```

This is, of course, the success_fraction probability 75%. What is the probability that the shooter only scores on the fifth shot? So there are 5-1 = 4 failures before the first success.

```
cout << "Probability of score on 5th try = " << pdf(gav, 4) << endl; // 0.0029
```

Now compare this with the poor and the best players success fraction. We need to constructing new distributions with the different success fractions, and then get the corresponding probability density functions values:

```
geometric gbest(0.95);
cout << "Probability of score on 5th try = " << pdf(gbest, 4) << endl; // 5.9e-6
geometric gmediocre(0.50);
cout << "Probability of score on 5th try = " << pdf(gmediocre, 4) << endl; // 0.031
```

So we can see the very much smaller chance (0.000006) of 4 failures by the best shooters, compared to the 0.03 of the mediocre.

Estimating failures

Of course one man's failure is an other man's success. So a fault can be defined as a 'success'.

If a fault occurs once after 100 flights, then one might naively say that the risk of fault is obviously 1 in 100 = 1/100, a probability of 0.01.

This is the best estimate we can make, but while it is the truth, it is not the whole truth, for it hides the big uncertainty when estimating from a single event. "One swallow doesn't make a summer." To show the magnitude of the uncertainty, the geometric (or the negative binomial) distribution can be used.

If we chose the popular 95% confidence in the limits, corresponding to an alpha of 0.05, because we are calculating a two-sided interval, we must divide alpha by two.

```
double alpha = 0.05;
double k = 100; // So frequency of occurrence is 1/100.
cout << "Probability is failure is " << 1/k << endl;
double t = geometric::find_lower_bound_on_p(k, alpha/2);
cout << "geometric::find_lower_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
  << t << endl; // 0.00025
t = geometric::find_upper_bound_on_p(k, alpha/2);
cout << "geometric::find_upper_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
  << t << endl; // 0.037
```

So while we estimate the probability is 0.01, it might lie between 0.0003 and 0.04. Even if we relax our confidence to alpha = 90%, the bounds only contract to 0.0005 and 0.03. And if we require a high confidence, they widen to 0.00005 to 0.05.

```
alpha = 0.1; // 90% confidence.
t = geometric::find_lower_bound_on_p(k, alpha/2);
cout << "geometric::find_lower_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
  << t << endl; // 0.0005
t = geometric::find_upper_bound_on_p(k, alpha/2);
cout << "geometric::find_upper_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
  << t << endl; // 0.03

alpha = 0.01; // 99% confidence.
t = geometric::find_lower_bound_on_p(k, alpha/2);
cout << "geometric::find_lower_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
  << t << endl; // 5e-005
t = geometric::find_upper_bound_on_p(k, alpha/2);
cout << "geometric::find_upper_bound_on_p(" << int(k) << ", " << alpha/2 << ") = "
    << t << endl; // 0.052
```

In real life, there will usually be more than one event (fault or success), when the negative binomial, which has the neccessary extra parameter, will be needed.

As noted above, using a catch block is always a good idea, even if you hope not to use it!

```
}
catch(const std::exception& e)
{ // Since we have set an overflow policy of ignore_error,
  // an overflow exception should never be thrown.
   std::cout << "\nMessage from thrown exception was:\n " << e.what() << std::endl;
```

For example, without a ignore domain error policy, if we asked for

```
pdf(g, -1)
```

for example, we would get an unhelpful abort, but with a catch:

```
Message from thrown exception was:
 Error in function boost::math::pdf(const exponential_distribution<double>&, double):
 Number of failures argument is -1, but must be >= 0 !
```

See full source C++ of this example at geometric_examples.cpp

See negative_binomial confidence interval example.

Negative Binomial Distribution Examples

(See also the reference documentation for the Negative Binomial Distribution.)

Calculating Confidence Limits on the Frequency of Occurrence for the Negative Binomial Distribution

Imagine you have a process that follows a negative binomial distribution: for each trial conducted, an event either occurs or does it does not, referred to as "successes" and "failures". The frequency with which successes occur is variously referred to as the success fraction, success ratio, success percentage, occurrence frequency, or probability of occurrence.

If, by experiment, you want to measure the the best estimate of success fraction is given simply by *k* / *N*, for *k* successes out of *N* trials.

However our confidence in that estimate will be shaped by how many trials were conducted, and how many successes were observed. The static member functions `negative_binomial_distribution<>::find_lower_bound_on_p` and `negative_binomial_distribution<>::find_upper_bound_on_p` allow you to calculate the confidence intervals for your estimate of the success fraction.

The sample program neg_binom_confidence_limits.cpp illustrates their use.

First we need some includes to access the negative binomial distribution (and some basic std output of course).

```
#include <boost/math/distributions/negative_binomial.hpp>
using boost::math::negative_binomial;

#include <iostream>
using std::cout; using std::endl;
#include <iomanip>
using std::setprecision;
using std::setw; using std::left; using std::fixed; using std::right;
```

First define a table of significance levels: these are the probabilities that the true occurrence frequency lies outside the calculated interval:

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Confidence value as % is (1 - alpha) * 100, so alpha 0.05 == 95% confidence that the true occurrence frequency lies **inside** the calculated interval.

We need a function to calculate and print confidence limits for an observed frequency of occurrence that follows a negative binomial distribution.

```
void confidence_limits_on_frequency(unsigned trials, unsigned successes)
{
   // trials = Total number of trials.
   // successes = Total number of observed successes.
   // failures = trials - successes.
   // success_fraction = successes /trials.
   // Print out general info:
   cout <<
      "______________________________________________\n"
      "2-Sided Confidence Limits For Success Fraction\n"
      "______________________________________________\n\n";
   cout << setprecision(7);
   cout << setw(40) << left << "Number of trials" << " =  " << trials << "\n";
   cout << setw(40) << left << "Number of successes" << " =  " << successes << "\n";
   cout << setw(40) << left << "Number of failures" << " =  " << trials - successes << "\n";
   cout << setw(40) << left << "Observed frequency of occurrence" << " =  " << double(suc↲
cesses) / trials << "\n";

   // Print table header:
   cout << "\n\n"
           "___________________________________________\n"
           "Confidence        Lower          Upper\n"
           " Value (%)        Limit          Limit\n"
           "___________________________________________\n";
```

And now for the important part - the bounds themselves. For each value of *alpha*, we call `find_lower_bound_on_p` and `find_upper_bound_on_p` to obtain lower and upper bounds respectively. Note that since we are calculating a two-sided interval, we must divide the value of alpha in two. Had we been calculating a single-sided interval, for example: *"Calculate a lower bound so that we are P% sure that the true occurrence frequency is greater than some value"* then we would **not** have divided by two.

```
   // Now print out the upper and lower limits for the alpha table values.
   for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
   {
      // Confidence value:
      cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
      // Calculate bounds:
      double lower = negative_binomial::find_lower_bound_on_p(trials, successes, alpha[i]/2);
      double upper = negative_binomial::find_upper_bound_on_p(trials, successes, alpha[i]/2);
      // Print limits:
      cout << fixed << setprecision(5) << setw(15) << right << lower;
      cout << fixed << setprecision(5) << setw(15) << right << upper << endl;
   }
   cout << endl;
} // void confidence_limits_on_frequency(unsigned trials, unsigned successes)
```

And then call confidence_limits_on_frequency with increasing numbers of trials, but always the same success fraction 0.1, or 1 in 10.

```
int main()
{
  confidence_limits_on_frequency(20, 2); // 20 trials, 2 successes, 2 in 20, = 1 in 10 = 0.1 ↲
success fraction.
  confidence_limits_on_frequency(200, 20); // More trials, but same 0.1 success fraction.
  confidence_limits_on_frequency(2000, 200); // Many more trials, but same 0.1 success fraction.

  return 0;
} // int main()
```

Let's see some sample output for a 1 in 10 success ratio, first for a mere 20 trials:

```
_______________________________________________
2-Sided Confidence Limits For Success Fraction
_______________________________________________
Number of trials                           =  20
Number of successes                        =  2
Number of failures                         =  18
Observed frequency of occurrence           =  0.1
____________________________________________
Confidence        Lower          Upper
 Value (%)        Limit          Limit
____________________________________________
    50.000        0.04812        0.13554
    75.000        0.03078        0.17727
    90.000        0.01807        0.22637
    95.000        0.01235        0.26028
    99.000        0.00530        0.33111
    99.900        0.00164        0.41802
    99.990        0.00051        0.49202
    99.999        0.00016        0.55574
```

As you can see, even at the 95% confidence level the bounds (0.012 to 0.26) are really very wide, and very asymmetric about the observed value 0.1.

Compare that with the program output for a mass 2000 trials:

```
_______________________________________________
2-Sided Confidence Limits For Success Fraction
_______________________________________________
Number of trials                           =  2000
Number of successes                        =  200
Number of failures                         =  1800
Observed frequency of occurrence           =  0.1
____________________________________________
Confidence        Lower          Upper
 Value (%)        Limit          Limit
____________________________________________
    50.000        0.09536        0.10445
    75.000        0.09228        0.10776
    90.000        0.08916        0.11125
    95.000        0.08720        0.11352
    99.000        0.08344        0.11802
    99.900        0.07921        0.12336
    99.990        0.07577        0.12795
    99.999        0.07282        0.13206
```

Now even when the confidence level is very high, the limits (at 99.999%, 0.07 to 0.13) are really quite close and nearly symmetric to the observed value of 0.1.

Estimating Sample Sizes for the Negative Binomial.

Imagine you have an event (let's call it a "failure" - though we could equally well call it a success if we felt it was a 'good' event) that you know will occur in 1 in N trials. You may want to know how many trials you need to conduct to be P% sure of observing at least k such failures. If the failure events follow a negative binomial distribution (each trial either succeeds or fails) then the static member function `negative_binomial_distibution<>::find_minimum_number_of_trials` can be used to estimate the minimum number of trials required to be P% sure of observing the desired number of failures.

The example program neg_binomial_sample_sizes.cpp demonstrates its usage.

It centres around a routine that prints out a table of minimum sample sizes (number of trials) for various probability thresholds:

```
void find_number_of_trials(double failures, double p);
```

First define a table of significance levels: these are the maximum acceptable probability that *failure* or fewer events will be observed.

```
double alpha[] = { 0.5, 0.25, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Confidence value as % is (1 - alpha) * 100, so alpha 0.05 == 95% confidence that the desired number of failures will be observed. The values range from a very low 0.5 or 50% confidence up to an extremely high confidence of 99.999.

Much of the rest of the program is pretty-printing, the important part is in the calculation of minimum number of trials required for each value of alpha using:

```
(int)ceil(negative_binomial::find_minimum_number_of_trials(failures, p, alpha[i]);
```

find_minimum_number_of_trials returns a double, so `ceil` rounds this up to ensure we have an integral minimum number of trials.

```
void find_number_of_trials(double failures, double p)
{
   // trials = number of trials
   // failures = number of failures before achieving required success(es).
   // p        = success fraction (0 <= p <= 1.).
   //
   // Calculate how many trials we need to ensure the
   // required number of failures DOES exceed "failures".

  cout << "\n""Target number of failures = " << (int)failures;
  cout << ",   Success fraction = " << fixed << setprecision(1) << 100 * p << "%" << endl;
   // Print table header:
   cout << "____________________________\n"
           "Confidence        Min Number\n"
           " Value (%)        Of Trials \n"
           "____________________________\n";
   // Now print out the data for the alpha table values.
  for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
   { // Confidence values %:
      cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]) << "      "
      // find_minimum_number_of_trials
      << setw(6) << right
      << (int)ceil(negative_binomial::find_minimum_number_of_trials(failures, p, alpha[i]))
      << endl;
   }
   cout << endl;
} // void find_number_of_trials(double failures, double p)
```

finally we can produce some tables of minimum trials for the chosen confidence levels:

```
int main()
{
    find_number_of_trials(5, 0.5);
    find_number_of_trials(50, 0.5);
    find_number_of_trials(500, 0.5);
    find_number_of_trials(50, 0.1);
    find_number_of_trials(500, 0.1);
    find_number_of_trials(5, 0.9);

    return 0;
} // int main()
```

Note

Since we're calculating the *minimum* number of trials required, we'll err on the safe side and take the ceiling of the result. Had we been calculating the *maximum* number of trials permitted to observe less than a certain number of *failures* then we would have taken the floor instead. We would also have called `find_minimum_number_of_trials` like this:

```
floor(negative_binomial::find_minimum_number_of_trials(failures, p, alpha[i]))
```

which would give us the largest number of trials we could conduct and still be P% sure of observing *failures or less* failure events, when the probability of success is *p*.

We'll finish off by looking at some sample output, firstly suppose we wish to observe at least 5 "failures" with a 50/50 (0.5) chance of success or failure:

```
Target number of failures = 5,   Success fraction = 50%


____________________________
Confidence        Min Number
 Value (%)        Of Trials
____________________________
    50.000          11
    75.000          14
    90.000          17
    95.000          18
    99.000          22
    99.900          27
    99.990          31
    99.999          36
```

So 18 trials or more would yield a 95% chance that at least our 5 required failures would be observed.

Compare that to what happens if the success ratio is 90%:

```
Target number of failures = 5.000,   Success fraction = 90.000%


____________________________
Confidence        Min Number
 Value (%)        Of Trials
____________________________
    50.000          57
    75.000          73
    90.000          91
    95.000         103
    99.000         127
    99.900         159
    99.990         189
    99.999         217
```

So now 103 trials are required to observe at least 5 failures with 95% certainty.

Negative Binomial Sales Quota Example.

This example program negative_binomial_example1.cpp (full source code) demonstrates a simple use to find the probability of meeting a sales quota.

Based on a problem by Dr. Diane Evans, Professor of Mathematics at Rose-Hulman Institute of Technology.

Pat is required to sell candy bars to raise money for the 6th grade field trip. There are thirty houses in the neighborhood, and Pat is not supposed to return home until five candy bars have been sold. So the child goes door to door, selling candy bars. At each house, there is a 0.4 probability (40%) of selling one candy bar and a 0.6 probability (60%) of selling nothing.

What is the probability mass (density) function (pdf) for selling the last (fifth) candy bar at the nth house?

The Negative Binomial(r, p) distribution describes the probability of k failures and r successes in k+r Bernoulli(p) trials with success on the last trial. (A Bernoulli trial is one with only two possible outcomes, success of failure, and p is the probability of success). See also Bernoulli distribution and Bernoulli applications.

In this example, we will deliberately produce a variety of calculations and outputs to demonstrate the ways that the negative binomial distribution can be implemented with this library: it is also deliberately over-commented.

First we need to #define macros to control the error and discrete handling policies. For this simple example, we want to avoid throwing an exception (the default policy) and just return infinity. We want to treat the distribution as if it was continuous, so we choose a discrete_quantile policy of real, rather than the default policy integer_round_outwards.

```
#define BOOST_MATH_OVERFLOW_ERROR_POLICY ignore_error
#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real
```

After that we need some includes to provide easy access to the negative binomial distribution,

Caution

It is vital to #include distributions etc **after** the above #defines

and we need some std library iostream, of course.

```
#include <boost/math/distributions/negative_binomial.hpp>
  // for negative_binomial_distribution
  using boost::math::negative_binomial; // typedef provides default type is double.
  using  ::boost::math::pdf; // Probability mass function.
  using  ::boost::math::cdf; // Cumulative density function.
  using  ::boost::math::quantile;

#include <iostream>
  using std::cout; using std::endl;
  using std::noshowpoint; using std::fixed; using std::right; using std::left;
#include <iomanip>
  using std::setprecision; using std::setw;

#include <limits>
  using std::numeric_limits;
```

It is always sensible to use try and catch blocks because defaults policies are to throw an exception if anything goes wrong.

A simple catch block (see below) will ensure that you get a helpful error message instead of an abrupt program abort.

```
try
{
```

Selling five candy bars means getting five successes, so successes r = 5. The total number of trials (n, in this case, houses visited) this takes is therefore = sucesses + failures or k + r = k + 5.

```
double sales_quota = 5; // Pat's sales quota - successes (r).
```

At each house, there is a 0.4 probability (40%) of selling one candy bar and a 0.6 probability (60%) of selling nothing.

```
double success_fraction = 0.4; // success_fraction (p) - so failure_fraction is 0.6.
```

The Negative Binomial(r, p) distribution describes the probability of k failures and r successes in k+r Bernoulli(p) trials with success on the last trial. (A Bernoulli trial is one with only two possible outcomes, success of failure, and p is the probability of success).

We therefore start by constructing a negative binomial distribution with parameters sales_quota (required successes) and probability of success.

```
negative_binomial nb(sales_quota, success_fraction); // type double by default.
```

To confirm, display the success_fraction & successes parameters of the distribution.

```
cout << "Pat has a sales per house success rate of " << success_fraction
  << ".\nTherefore he would, on average, sell " << nb.success_fraction() * 100
  << " bars after trying 100 houses." << endl;

int all_houses = 30; // The number of houses on the estate.

cout << "With a success rate of " << nb.success_fraction()
  << ", he might expect, on average,\n"
    "to need to visit about " << success_fraction * all_houses
    << " houses in order to sell all " << nb.successes() << " bars. " << endl;
```

```
Pat has a sales per house success rate of 0.4.
Therefore he would, on average, sell 40 bars after trying 100 houses.
With a success rate of 0.4, he might expect, on average,
to need to visit about 12 houses in order to sell all 5 bars.
```

The random variable of interest is the number of houses that must be visited to sell five candy bars, so we substitute k = n - 5 into a negative_binomial(5, 0.4) and obtain the Probability Density Function of the distribution of houses visited. Obviously, the best possible case is that Pat makes sales on all the first five houses.

We calculate this using the pdf function:

```
cout << "Probability that Pat finishes on the " << sales_quota << "th house is "
  << pdf(nb, 5 - sales_quota) << endl; // == pdf(nb, 0)
```

Of course, he could not finish on fewer than 5 houses because he must sell 5 candy bars. So the 5th house is the first that he could possibly finish on.

To finish on or before the 8th house, Pat must finish at the 5th, 6th, 7th or 8th house. The probability that he will finish on **exactly** (==) on any house is the Probability Density Function (pdf).

```
cout << "Probability that Pat finishes on the 6th house is "
  << pdf(nb, 6 - sales_quota) << endl;
cout << "Probability that Pat finishes on the 7th house is "
  << pdf(nb, 7 - sales_quota) << endl;
cout << "Probability that Pat finishes on the 8th house is "
  << pdf(nb, 8 - sales_quota) << endl;
```

```
Probability that Pat finishes on the 6th house is 0.03072
Probability that Pat finishes on the 7th house is 0.055296
Probability that Pat finishes on the 8th house is 0.077414
```

The sum of the probabilities for these houses is the Cumulative Distribution Function (cdf). We can calculate it by adding the individual probabilities.

```
cout << "Probability that Pat finishes on or before the 8th house is sum "
  "\n" << "pdf(sales_quota) + pdf(6) + pdf(7) + pdf(8) = "
  // Sum each of the mass/density probabilities for houses sales_quota = 5, 6, 7, & 8.
  << pdf(nb, 5 - sales_quota) // 0 failures.
    + pdf(nb, 6 - sales_quota) // 1 failure.
    + pdf(nb, 7 - sales_quota) // 2 failures.
    + pdf(nb, 8 - sales_quota) // 3 failures.
  << endl;
```

```
pdf(sales_quota) + pdf(6) + pdf(7) + pdf(8) = 0.17367
```

Or, usually better, by using the negative binomial **cumulative** distribution function.

```
cout << "\nProbability of selling his quota of " << sales_quota
  << " bars\non or before the " << 8 << "th house is "
  << cdf(nb, 8 - sales_quota) << endl;
```

```
Probability of selling his quota of 5 bars on or before the 8th house is 0.17367
```

```
cout << "\nProbability that Pat finishes exactly on the 10th house is "
  << pdf(nb, 10 - sales_quota) << endl;
cout << "\nProbability of selling his quota of " << sales_quota
  << " bars\non or before the " << 10 << "th house is "
  << cdf(nb, 10 - sales_quota) << endl;
```

```
Probability that Pat finishes exactly on the 10th house is 0.10033
Probability of selling his quota of 5 bars on or before the 10th house is 0.3669
```

```
cout << "Probability that Pat finishes exactly on the 11th house is "
  << pdf(nb, 11 - sales_quota) << endl;
cout << "\nProbability of selling his quota of " << sales_quota
  << " bars\non or before the " << 11 << "th house is "
  << cdf(nb, 11 - sales_quota) << endl;
```

```
Probability that Pat finishes on the 11th house is 0.10033
Probability of selling his quota of 5 candy bars
on or before the 11th house is 0.46723
```

```
cout << "Probability that Pat finishes exactly on the 12th house is "
  << pdf(nb, 12 - sales_quota) << endl;

cout << "\nProbability of selling his quota of " << sales_quota
  << " bars\non or before the " << 12 << "th house is "
  << cdf(nb, 12 - sales_quota) << endl;
```

```
Probability that Pat finishes on the 12th house is 0.094596
Probability of selling his quota of 5 candy bars
on or before the 12th house is 0.56182
```

Finally consider the risk of Pat not selling his quota of 5 bars even after visiting all the houses. Calculate the probability that he *will* sell on or before the last house: Calculate the probability that he would sell all his quota on the very last house.

```
cout << "Probability that Pat finishes on the " << all_houses
  << " house is " << pdf(nb, all_houses - sales_quota) << endl;
```

Probability of selling his quota of 5 bars on the 30th house is

```
Probability that Pat finishes on the 30 house is 0.00069145
```

when he'd be very unlucky indeed!

What is the probability that Pat exhausts all 30 houses in the neighborhood, and **still** doesn't sell the required 5 candy bars?

```
cout << "\nProbability of selling his quota of " << sales_quota
  << " bars\non or before the " << all_houses << "th house is "
  << cdf(nb, all_houses - sales_quota) << endl;
```

```
Probability of selling his quota of 5 bars
on or before the 30th house is 0.99849
```

So the risk of failing even after visiting all the houses is 1 - this probability,

```
1 - cdf(nb, all_houses - sales_quota
```

But using this expression may cause serious inaccuracy, so it would be much better to use the complement of the cdf: So the risk of failing even at, or after, the 31th (non-existent) houses is 1 - this probability,

```
1 - cdf(nb, all_houses - sales_quota)
```

But using this expression may cause serious inaccuracy. So it would be much better to use the __complement of the cdf (see why complements?).

```
cout << "\nProbability of failing to sell his quota of " << sales_quota
  << " bars\neven after visiting all " << all_houses << " houses is "
  << cdf(complement(nb, all_houses - sales_quota)) << endl;
```

```
Probability of failing to sell his quota of 5 bars
even after visiting all 30 houses is 0.0015101
```

We can also use the quantile (percentile), the inverse of the cdf, to predict which house Pat will finish on. So for the 8th house:

```
double p = cdf(nb, (8 - sales_quota));
cout << "Probability of meeting sales quota on or before 8th house is "<< p << endl;
```

```
Probability of meeting sales quota on or before 8th house is 0.174
```

```
cout << "If the confidence of meeting sales quota is " << p
    << ", then the finishing house is " << quantile(nb, p) + sales_quota << endl;

cout<< " quantile(nb, p) = " << quantile(nb, p) << endl;
```

```
If the confidence of meeting sales quota is 0.17367, then the finishing house is 8
```

Demanding absolute certainty that all 5 will be sold, implies an infinite number of trials. (Of course, there are only 30 houses on the estate, so he can't ever be **certain** of selling his quota).

```
cout << "If the confidence of meeting sales quota is " << 1.
    << ", then the finishing house is " << quantile(nb, 1) + sales_quota << endl;
//  1.#INF == infinity.
```

```
If the confidence of meeting sales quota is 1, then the finishing house is 1.#INF
```

And similarly for a few other probabilities:

```
cout << "If the confidence of meeting sales quota is " << 0.
    << ", then the finishing house is " << quantile(nb, 0.) + sales_quota << endl;

cout << "If the confidence of meeting sales quota is " << 0.5
    << ", then the finishing house is " << quantile(nb, 0.5) + sales_quota << endl;

cout << "If the confidence of meeting sales quota is " << 1 - 0.00151 // 30 th
    << ", then the finishing house is " << quantile(nb, 1 - 0.00151) + sales_quota << endl;
```

```
If the confidence of meeting sales quota is 0, then the finishing house is 5
If the confidence of meeting sales quota is 0.5, then the finishing house is 11.337
If the confidence of meeting sales quota is 0.99849, then the finishing house is 30
```

Notice that because we chose a discrete quantile policy of real, the result can be an 'unreal' fractional house.

If the opposite is true, we don't want to assume any confidence, then this is tantamount to assuming that all the first sales_quota trials will be successful sales.

```
cout << "If confidence of meeting quota is zero\n(we assume all houses are successful sales)"
  ", then finishing house is " << sales_quota << endl;
```

```
If confidence of meeting quota is zero (we assume all houses are successful sales), then fin↵
ishing house is 5
If confidence of meeting quota is 0, then finishing house is 5
```

We can list quantiles for a few probabilities:

```
 double ps[] = {0., 0.001, 0.01, 0.05, 0.1, 0.5, 0.9, 0.95, 0.99, 0.999, 1.};
 // Confidence as fraction = 1-alpha, as percent =  100 * (1-alpha[i]) %
 cout.precision(3);
 for (unsigned i = 0; i < sizeof(ps)/sizeof(ps[0]); i++)
 {
   cout << "If confidence of meeting quota is " << ps[i]
     << ", then finishing house is " << quantile(nb, ps[i]) + sales_quota
     << endl;
}
```

```
If confidence of meeting quota is 0, then finishing house is 5
If confidence of meeting quota is 0.001, then finishing house is 5
If confidence of meeting quota is 0.01, then finishing house is 5
If confidence of meeting quota is 0.05, then finishing house is 6.2
If confidence of meeting quota is 0.1, then finishing house is 7.06
If confidence of meeting quota is 0.5, then finishing house is 11.3
If confidence of meeting quota is 0.9, then finishing house is 17.8
If confidence of meeting quota is 0.95, then finishing house is 20.1
If confidence of meeting quota is 0.99, then finishing house is 24.8
If confidence of meeting quota is 0.999, then finishing house is 31.1
If confidence of meeting quota is 1, then finishing house is 1.#INF
```

We could have applied a ceil function to obtain a 'worst case' integer value for house.

```
ceil(quantile(nb, ps[i]))
```

Or, if we had used the default discrete quantile policy, integer_outside, by omitting

```
#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real
```

we would have achieved the same effect.

The real result gives some suggestion which house is most likely. For example, compare the real and integer_outside for 95% confidence.

```
If confidence of meeting quota is 0.95, then finishing house is 20.1
If confidence of meeting quota is 0.95, then finishing house is 21
```

The real value 20.1 is much closer to 20 than 21, so integer_outside is pessimistic. We could also use integer_round_nearest policy to suggest that 20 is more likely.

Finally, we can tabulate the probability for the last sale being exactly on each house.

```
cout << "\nHouse for " << sales_quota << "th (last) sale.  Probability (%)" << endl;
cout.precision(5);
for (int i = (int)sales_quota; i < all_houses+1; i++)
{
  cout << left << setw(3) << i << "                              ↵
" << setw(8) << cdf(nb, i - sales_quota)  << endl;
}
cout << endl;
```

```
House for 5 th (last) sale.  Probability (%)
5                               0.01024
6                               0.04096
7                               0.096256
8                               0.17367
9                               0.26657
10                              0.3669
11                              0.46723
12                              0.56182
13                              0.64696
14                              0.72074
15                              0.78272
16                              0.83343
17                              0.874
18                              0.90583
19                              0.93039
20                              0.94905
21                              0.96304
22                              0.97342
23                              0.98103
24                              0.98655
25                              0.99053
26                              0.99337
27                              0.99539
28                              0.99681
29                              0.9978
30                              0.99849
```

As noted above, using a catch block is always a good idea, even if you do not expect to use it.

```
}
catch(const std::exception& e)
{ // Since we have set an overflow policy of ignore_error,
  // an overflow exception should never be thrown.
   std::cout << "\nMessage from thrown exception was:\n " << e.what() << std::endl;
```

For example, without a ignore domain error policy, if we asked for

```
pdf(nb, -1)
```

for example, we would get:

```
Message from thrown exception was:
 Error in function boost::math::pdf(const negative_binomial_distribution<double>&, double):
 Number of failures argument is -1, but must be >= 0 !
```

Negative Binomial Table Printing Example.

Example program showing output of a table of values of cdf and pdf for various k failures.

```
// Print a table of values that can be used to plot
// using Excel, or some other superior graphical display tool.

cout.precision(17); // Use max_digits10 precision, the maximum available for a reference table.
cout << showpoint << endl; // include trailing zeros.
// This is a maximum possible precision for the type (here double) to suit a reference table.
int maxk = static_cast<int>(2. * mynbdist.successes() /  mynbdist.success_fraction());
// This maxk shows most of the range of interest, probability about 0.0001 to 0.999.
cout << "\n"" k            pdf                        cdf""\n" << endl;
for (int k = 0; k < maxk; k++)
{
  cout << right << setprecision(17) << showpoint
    << right << setw(3) << k  << ", "
    << left << setw(25) << pdf(mynbdist, static_cast<double>(k))
    << left << setw(25) << cdf(mynbdist, static_cast<double>(k))
    << endl;
}
cout << endl;
```

```
k            pdf                        cdf
 0, 1.5258789062500000e-005  1.5258789062500003e-005
 1, 9.1552734375000000e-005  0.00010681152343750000
 2, 0.00030899047851562522   0.00041580200195312500
 3, 0.00077247619628906272   0.0011882781982421875
 4, 0.0015932321548461918    0.0027815103530883789
 5, 0.0028678178787231476    0.0056493282318115234
 6, 0.0046602040529251142    0.010309532284736633
 7, 0.0069903060793876605    0.017299838364124298
 8, 0.0098301179241389001    0.027129956288263202
 9, 0.013106823898851871     0.040236780187115073
10, 0.016711200471036140     0.056947980658151209
11, 0.020509200578089786     0.077457181236241013
12, 0.024354675686481652     0.10181185692272265
13, 0.028101548869017230     0.12991340579173993
14, 0.031614242477644432     0.16152764826938440
15, 0.034775666725408917     0.19630331499479325
16, 0.037492515688331451     0.23379583068312471
17, 0.039697957787645101     0.27349378847076977
18, 0.041352039362130305     0.31484582783290005
19, 0.042440250924291580     0.35728607875719176
20, 0.042970754060845245     0.40025683281803687
21, 0.042970754060845225     0.44322758687888220
22, 0.042482450037426581     0.48571003691630876
23, 0.041558918514873783     0.52726895543118257
24, 0.040260202311284021     0.56752915774246648
25, 0.038649794218832620     0.60617895196129912
26, 0.036791631035234917     0.64297058299653398
27, 0.034747651533277427     0.67771823452981139
28, 0.032575923312447595     0.71029415784225891
29, 0.030329307911589130     0.74062346575384819
30, 0.028054609818219924     0.76867807557206813
31, 0.025792141284492545     0.79447021685656061
32, 0.023575629142856460     0.81804584599941710
33, 0.021432390129869489     0.83947823612928651
34, 0.019383705779220189     0.85886194190850684
35, 0.017445335201298231     0.87630727710980494
36, 0.015628112784496322     0.89193538989430121
37, 0.013938587078064250     0.90587397697236549
38, 0.012379666154859701     0.91825364312722524
39, 0.010951243136991251     0.92920488626421649
40, 0.0096507830144735539    0.93885566927869002
41, 0.0084738582566109364    0.94732952753530097
```

```
42, 0.0074146259745345548    0.95474415350983555
43, 0.0064662435824429246    0.96121039709227851
44, 0.0056212231142827853    0.96683162020656122
45, 0.0048717266990450708    0.97170334690560634
46, 0.0042098073105878630    0.97591315421619418
47, 0.0036275999165703964    0.97954075413276465
48, 0.0031174686783026818    0.98265822281106729
49, 0.0026721160099737302    0.98533033882104104
50, 0.0022846591885275322    0.98761499800956853
51, 0.0019486798960970148    0.98956367790566557
52, 0.0016582516423517923    0.99122192954801736
53, 0.0014079495076571762    0.99262987905567457
54, 0.0011928461106539983    0.99382272516632852
55, 0.0010084971662802015    0.99483122233260868
56, 0.00085091948404891532   0.99568214181665760
57, 0.00071656377604119542   0.99639870559269883
58, 0.00060228420831048650   0.99700098980100937
59, 0.00050530624256557675   0.99750629604357488
60, 0.00042319397814867202   0.99792949002172360
61, 0.00035381791615708398   0.99828330793788067
62, 0.00029532382517950324   0.99857863176306016
63, 0.00024610318764958566   0.99882473495070978
```

Normal Distribution Examples

(See also the reference documentation for the Normal Distribution.)

Some Miscellaneous Examples of the Normal (Gaussian) Distribution

The sample program normal_misc_examples.cpp illustrates their use.

Traditional Tables

First we need some includes to access the normal distribution (and some std output of course).

```
#include <boost/math/distributions/normal.hpp> // for normal_distribution
  using boost::math::normal; // typedef provides default type is double.

#include <iostream>
  using std::cout; using std::endl; using std::left; using std::showpoint; using std::noshow↵
point;
#include <iomanip>
  using std::setw; using std::setprecision;
#include <limits>
  using std::numeric_limits;

int main()
{
  cout << "Example: Normal distribution, Miscellaneous Applications.";

  try
  {
    { // Traditional tables and values.
```

Let's start by printing some traditional tables.

```
double step = 1.; // in z
double range = 4; // min and max z = -range to +range.
int precision = 17; // traditional tables are only computed to much lower precision.
// but std::numeric_limits<double>::max_digits10; on new Standard Libraries gives
// 17, the maximum number of digits that can possibly be significant.
// std::numeric_limits<double>::digits10; == 15 is number of guaranteed digits,
// the other two digits being 'noisy'.

// Construct a standard normal distribution s
  normal s; // (default mean = zero, and standard deviation = unity)
  cout << "Standard normal distribution, mean = "<< s.mean()
    << ", standard deviation = " << s.standard_deviation() << endl;
```

First the probability distribution function (pdf).

```
cout << "Probability distribution function values" << endl;
cout << "  z " "      pdf " << endl;
cout.precision(5);
for (double z = -range; z < range + step; z += step)
{
  cout << left << setprecision(3) << setw(6) << z << " "
    << setprecision(precision) << setw(12) << pdf(s, z) << endl;
}
cout.precision(6); // default
```

And the area under the normal curve from -∞ up to z, the cumulative distribution function (cdf).

```
// For a standard normal distribution
cout << "Standard normal mean = "<< s.mean()
  << ", standard deviation = " << s.standard_deviation() << endl;
cout << "Integral (area under the curve) from - infinity up to z " << endl;
cout << "  z " "      cdf " << endl;
for (double z = -range; z < range + step; z += step)
{
  cout << left << setprecision(3) << setw(6) << z << " "
    << setprecision(precision) << setw(12) << cdf(s, z) << endl;
}
cout.precision(6); // default
```

And all this you can do with a nanoscopic amount of work compared to the team of **human computers** toiling with Milton Abramovitz and Irene Stegen at the US National Bureau of Standards (now NIST). Starting in 1938, their "Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables", was eventually published in 1964, and has been reprinted numerous times since. (A major replacement is planned at Digital Library of Mathematical Functions).

Pretty-printing a traditional 2-dimensional table is left as an exercise for the student, but why bother now that the Math Toolkit lets you write

```
double z = 2.;
cout << "Area for z = " << z << " is " << cdf(s, z) << endl; // to get the area for z.
```

Correspondingly, we can obtain the traditional 'critical' values for significance levels. For the 95% confidence level, the significance level usually called alpha, is 0.05 = 1 - 0.95 (for a one-sided test), so we can write

```
  cout << "95% of area has a z below " << quantile(s, 0.95) << endl;
// 95% of area has a z below 1.64485
```

and a two-sided test (a comparison between two levels, rather than a one-sided test)

```
  cout << "95% of area has a z between " << quantile(s, 0.975)
    << " and " << -quantile(s, 0.975) << endl;
// 95% of area has a z between 1.95996 and -1.95996
```

First, define a table of significance levels: these are the probabilities that the true occurrence frequency lies outside the calculated interval.

It is convenient to have an alpha level for the probability that z lies outside just one standard deviation. This will not be some nice neat number like 0.05, but we can easily calculate it,

```
double alpha1 = cdf(s, -1) * 2; // 0.3173105078629142
cout << setprecision(17) << "Significance level for z == 1 is " << alpha1 << endl;
```

and place in our array of favorite alpha values.

```
double alpha[] = {0.3173105078629142, // z for 1 standard deviation.
  0.20, 0.1, 0.05, 0.01, 0.001, 0.0001, 0.00001 };
```

Confidence value as % is (1 - alpha) * 100 (so alpha 0.05 == 95% confidence) that the true occurrence frequency lies **inside** the calculated interval.

```
cout << "level of significance (alpha)" << setprecision(4) << endl;
cout << "2-sided       1 -sided          z(alpha) " << endl;
for (unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
  cout << setw(15) << alpha[i] << setw(15) << alpha[i] /2 << setw(10) << quantile(complement(s, ↵
 alpha[i]/2)) << endl;
  // Use quantile(complement(s, alpha[i]/2)) to avoid potential loss of accuracy from ↵
quantile(s,  1 - alpha[i]/2)
}
cout << endl;
```

Notice the distinction between one-sided (also called one-tailed) where we are using a > **or** < test (and not both) and considering the area of the tail (integral) from z up to +∞, and a two-sided test where we are using two > **and** < tests, and thus considering two tails, from -∞ up to z low and z high up to +∞.

So the 2-sided values alpha[i] are calculated using alpha[i]/2.

If we consider a simple example of alpha = 0.05, then for a two-sided test, the lower tail area from -∞ up to -1.96 is 0.025 (alpha/2) and the upper tail area from +z up to +1.96 is also 0.025 (alpha/2), and the area between -1.96 up to 12.96 is alpha = 0.95. and the sum of the two tails is 0.025 + 0.025 = 0.05,

Standard deviations either side of the Mean

Armed with the cumulative distribution function, we can easily calculate the easy to remember proportion of values that lie within 1, 2 and 3 standard deviations from the mean.

```
cout.precision(3);
cout << showpoint << "cdf(s, s.standard_deviation()) = "
  << cdf(s, s.standard_deviation()) << endl;  // from -infinity to 1 sd
cout << "cdf(complement(s, s.standard_deviation())) = "
  << cdf(complement(s, s.standard_deviation())) << endl;
cout << "Fraction 1 standard deviation within either side of mean is "
  << 1 -  cdf(complement(s, s.standard_deviation())) * 2 << endl;
cout << "Fraction 2 standard deviations within either side of mean is "
  << 1 -  cdf(complement(s, 2 * s.standard_deviation())) * 2 << endl;
cout << "Fraction 3 standard deviations within either side of mean is "
  << 1 -  cdf(complement(s, 3 * s.standard_deviation())) * 2 << endl;
```

To a useful precision, the 1, 2 & 3 percentages are 68, 95 and 99.7, and these are worth memorising as useful 'rules of thumb', as, for example, in standard deviation:

```
Fraction 1 standard deviation within either side of mean is 0.683
Fraction 2 standard deviations within either side of mean is 0.954
Fraction 3 standard deviations within either side of mean is 0.997
```

We could of course get some really accurate values for these confidence intervals by using cout.precision(15);

```
Fraction 1 standard deviation within either side of mean is 0.682689492137086
Fraction 2 standard deviations within either side of mean is 0.954499736103642
Fraction 3 standard deviations within either side of mean is 0.997300203936740
```

But before you get too excited about this impressive precision, don't forget that the **confidence intervals of the standard deviation** are surprisingly wide, especially if you have estimated the standard deviation from only a few measurements.

Some simple examples

Life of light bulbs

Examples from K. Krishnamoorthy, Handbook of Statistical Distributions with Applications, ISBN 1 58488 635 8, page 125... implemented using the Math Toolkit library.

A few very simple examples are shown here:

```
// K. Krishnamoorthy, Handbook of Statistical Distributions with Applications,
 // ISBN 1 58488 635 8, page 125, example 10.3.5
```

Mean lifespan of 100 W bulbs is 1100 h with standard deviation of 100 h. Assuming, perhaps with little evidence and much faith, that the distribution is normal, we construct a normal distribution called *bulbs* with these values:

```
double mean_life = 1100.;
double life_standard_deviation = 100.;
normal bulbs(mean_life, life_standard_deviation);
double expected_life = 1000.;
```

The we can use the Cumulative distribution function to predict fractions (or percentages, if * 100) that will last various lifetimes.

```
cout << "Fraction of bulbs that will last at best (<=) " // P(X <= 1000)
  << expected_life << " is "<< cdf(bulbs, expected_life) << endl;
cout << "Fraction of bulbs that will last at least (>) " // P(X > 1000)
  << expected_life << " is "<< cdf(complement(bulbs, expected_life)) << endl;
double min_life = 900;
double max_life = 1200;
cout << "Fraction of bulbs that will last between "
  << min_life << " and " << max_life << " is "
  << cdf(bulbs, max_life)  // P(X <= 1200)
   - cdf(bulbs, min_life) << endl; // P(X <= 900)
```

Note

Real-life failures are often very ab-normal, with a significant number that 'dead-on-arrival' or suffer failure very early in their life: the lifetime of the survivors of 'early mortality' may be well described by the normal distribution.

How many onions?

Weekly demand for 5 lb sacks of onions at a store is normally distributed with mean 140 sacks and standard deviation 10.

```
double mean = 140.; // sacks per week.
double standard_deviation = 10;
normal sacks(mean, standard_deviation);

double stock = 160.; // per week.
cout << "Percentage of weeks overstocked "
  << cdf(sacks, stock) * 100. << endl; // P(X <=160)
// Percentage of weeks overstocked 97.7
```

So there will be lots of mouldy onions! So we should be able to say what stock level will meet demand 95% of the weeks.

```
double stock_95 = quantile(sacks, 0.95);
cout << "Store should stock " << int(stock_95) << " sacks to meet 95% of demands." << endl;
```

And it is easy to estimate how to meet 80% of demand, and waste even less.

```
double stock_80 = quantile(sacks, 0.80);
cout << "Store should stock " << int(stock_80) << " sacks to meet 8 out of 10 demands." << endl;
```

Packing beef

A machine is set to pack 3 kg of ground beef per pack. Over a long period of time it is found that the average packed was 3 kg with a standard deviation of 0.1 kg. Assuming the packing is normally distributed, we can find the fraction (or %) of packages that weigh more than 3.1 kg.

```
double mean = 3.; // kg
double standard_deviation = 0.1; // kg
normal packs(mean, standard_deviation);

double max_weight = 3.1; // kg
cout << "Percentage of packs > " << max_weight << " is "
<< cdf(complement(packs, max_weight)) << endl; // P(X > 3.1)

double under_weight = 2.9;
cout <<"fraction of packs <= " << under_weight << " with a mean of " << mean
  << " is " << cdf(complement(packs, under_weight)) << endl;
// fraction of packs <= 2.9 with a mean of 3 is 0.841345
// This is 0.84 - more than the target 0.95
// Want 95% to be over this weight, so what should we set the mean weight to be?
// KK StatCalc says:
double over_mean = 3.0664;
normal xpacks(over_mean, standard_deviation);
cout << "fraction of packs >= " << under_weight
<< " with a mean of " << xpacks.mean()
  << " is " << cdf(complement(xpacks, under_weight)) << endl;
// fraction of packs >= 2.9 with a mean of 3.06449 is 0.950005
double under_fraction = 0.05;  // so 95% are above the minimum weight mean - sd = 2.9
double low_limit = standard_deviation;
double offset = mean - low_limit - quantile(packs, under_fraction);
double nominal_mean = mean + offset;

normal nominal_packs(nominal_mean, standard_deviation);
cout << "Setting the packer to " << nominal_mean << " will mean that "
  << "fraction of packs >= " << under_weight
  << " is " << cdf(complement(nominal_packs, under_weight)) << endl;
```

Setting the packer to 3.06449 will mean that fraction of packs >= 2.9 is 0.95.

Setting the packer to 3.13263 will mean that fraction of packs >= 2.9 is 0.99, but will more than double the mean loss from 0.0644 to 0.133.

Alternatively, we could invest in a better (more precise) packer with a lower standard deviation.

To estimate how much better (how much smaller standard deviation) it would have to be, we need to get the 5% quantile to be located at the under_weight limit, 2.9

```
double p = 0.05; // wanted p th quantile.
cout << "Quantile of " << p << " = " << quantile(packs, p)
  << ", mean = " << packs.mean() << ", sd = " << packs.standard_deviation() << endl; //
```

Quantile of 0.05 = 2.83551, mean = 3, sd = 0.1

With the current packer (mean = 3, sd = 0.1), the 5% quantile is at 2.8551 kg, a little below our target of 2.9 kg. So we know that the standard deviation is going to have to be smaller.

Let's start by guessing that it (now 0.1) needs to be halved, to a standard deviation of 0.05

```
normal pack05(mean, 0.05);
cout << "Quantile of " << p << " = " << quantile(pack05, p)
  << ", mean = " << pack05.mean() << ", sd = " << pack05.standard_deviation() << endl;

cout <<"Fraction of packs >= " << under_weight << " with a mean of " << mean
  << " and standard deviation of " << pack05.standard_deviation()
  << " is " << cdf(complement(pack05, under_weight)) << endl;
//
```

Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.05 is 0.9772

So 0.05 was quite a good guess, but we are a little over the 2.9 target, so the standard deviation could be a tiny bit more. So we could do some more guessing to get closer, say by increasing to 0.06

```
normal pack06(mean, 0.06);
cout << "Quantile of " << p << " = " << quantile(pack06, p)
  << ", mean = " << pack06.mean() << ", sd = " << pack06.standard_deviation() << endl;

cout <<"Fraction of packs >= " << under_weight << " with a mean of " << mean
  << " and standard deviation of " << pack06.standard_deviation()
  << " is " << cdf(complement(pack06, under_weight)) << endl;
```

Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.06 is 0.9522

Now we are getting really close, but to do the job properly, we could use root finding method, for example the tools provided, and used elsewhere, in the Math Toolkit, see root-finding without derivatives.

But in this normal distribution case, we could be even smarter and make a direct calculation.

```
normal s; // For standard normal distribution,
double sd = 0.1;
double x = 2.9; // Our required limit.
// then probability p = N((x - mean) / sd)
// So if we want to find the standard deviation that would be required to meet this limit,
// so that the p th quantile is located at x,
// in this case the 0.95 (95%) quantile at 2.9 kg pack weight, when the mean is 3 kg.

double prob =  pdf(s, (x - mean) / sd);
double qp = quantile(s, 0.95);
cout << "prob = " << prob << ", quantile(p) " << qp << endl; // p = 0.241971, quantile(p) 1.64485
// Rearranging, we can directly calculate the required standard deviation:
double sd95 = std::abs((x - mean)) / qp;

cout << "If we want the "<< p << " th quantile to be located at "
  << x << ", would need a standard deviation of " << sd95 << endl;

normal pack95(mean, sd95);  // Distribution of the 'ideal better' packer.
cout <<"Fraction of packs >= " << under_weight << " with a mean of " << mean
  << " and standard deviation of " << pack95.standard_deviation()
  << " is " << cdf(complement(pack95, under_weight)) << endl;

// Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.0608 is 0.95
```

Notice that these two deceptively simple questions (do we over-fill or measure better) are actually very common. The weight of beef might be replaced by a measurement of more or less anything. But the calculations rely on the accuracy of the standard deviation - something that is almost always less good than we might wish, especially if based on a few measurements.

Length of bolts

A bolt is usable if between 3.9 and 4.1 long. From a large batch of bolts, a sample of 50 show a mean length of 3.95 with standard deviation 0.1. Assuming a normal distribution, what proportion is usable? The true sample mean is unknown, but we can use the sample mean and standard deviation to find approximate solutions.

```
    normal bolts(3.95, 0.1);
    double top = 4.1;
    double bottom = 3.9;

cout << "Fraction long enough [ P(X <= " << top << ") ] is " << cdf(bolts, top) << endl;
cout << "Fraction too short [ P(X <= " << bottom << ") ] is " << cdf(bolts, bottom) << endl;
cout << "Fraction OK  -between " << bottom << " and " << top
  << "[ P(X <= " << top  << ") - P(X<= " << bottom << " ) ] is "
  << cdf(bolts, top) - cdf(bolts, bottom) << endl;

cout << "Fraction too long [ P(X > " << top << ") ] is "
  << cdf(complement(bolts, top)) << endl;

cout << "95% of bolts are shorter than " << quantile(bolts, 0.95) << endl;
```

Inverse Chi-Squared Distribution Bayes Example

The scaled-inversed-chi-squared distribution is the conjugate prior distribution for the variance (σ^2) parameter of a normal distribution with known expectation (μ). As such it has widespread application in Bayesian statistics:

In Bayesian inference, the strength of belief into certain parameter values is itself described through a distribution. Parameters hence become themselves modelled and interpreted as random variables.

In this worked example, we perform such a Bayesian analysis by using the scaled-inverse-chi-squared distribution as prior and posterior distribution for the variance parameter of a normal distribution.

For more general information on Bayesian type of analyses, see:

- Andrew Gelman, John B. Carlin, Hal E. Stern, Donald B. Rubin, Bayesian Data Analysis, 2003, ISBN 978-1439840955.
- Jim Albert, Bayesian Compution with R, Springer, 2009, ISBN 978-0387922973.

(As the scaled-inversed-chi-squared is another parameterization of the inverse-gamma distribution, this example could also have used the inverse-gamma distribution).

Consider precision machines which produce balls for a high-quality ball bearing. Ideally each ball should have a diameter of precisely 3000 μm (3 mm). Assume that machines generally produce balls of that size on average (mean), but individual balls can vary slightly in either direction following (approximately) a normal distribution. Depending on various production conditions (e.g. raw material used for balls, workplace temperature and humidity, maintenance frequency and quality) some machines produce balls tighter distributed around the target of 3000 μm, while others produce balls with a wider distribution. Therefore the variance parameter of the normal distribution of the ball sizes varies from machine to machine. An extensive survey by the precision machinery manufacturer, however, has shown that most machines operate with a variance between 15 and 50, and near 25 μm^2 on average.

Using this information, we want to model the variance of the machines. The variance is strictly positive, and therefore we look for a statistical distribution with support in the positive domain of the real numbers. Given the expectation of the normal distribution of the balls is known (3000 μm), for reasons of conjugacy, it is customary practice in Bayesian statistics to model the variance to be scaled-inverse-chi-squared distributed.

In a first step, we will try to use the survey information to model the general knowledge about the variance parameter of machines measured by the manufacturer. This will provide us with a generic prior distribution that is applicable if nothing more specific is known about a particular machine.

In a second step, we will then combine the prior-distribution information in a Bayesian analysis with data on a specific single machine to derive a posterior distribution for that machine.

Step one: Using the survey information.

Using the survey results, we try to find the parameter set of a scaled-inverse-chi-squared distribution so that the properties of this distribution match the results. Using the mathematical properties of the scaled-inverse-chi-squared distribution as guideline, we see that that both the mean and mode of the scaled-inverse-chi-squared distribution are approximately given by the scale parameter (s) of the distribution. As the survey machines operated at a variance of 25 µm^2 on average, we hence set the scale parameter (s_{prior}) of our prior distribution equal to this value. Using some trial-and-error and calls to the global quantile function, we also find that a value of 20 for the degrees-of-freedom (ν_{prior}) parameter is adequate so that most of the prior distribution mass is located between 15 and 50 (see figure below).

We first construct our prior distribution using these values, and then list out a few quantiles:

```
double priorDF = 20.0;
double priorScale = 25.0;

inverse_chi_squared prior(priorDF, priorScale);
// Using an inverse_gamma distribution instead, we could equivalently write
// inverse_gamma prior(priorDF / 2.0, priorScale * priorDF / 2.0);

cout << "Prior distribution:" << endl << endl;
cout << "  2.5% quantile: " << quantile(prior, 0.025) << endl;
cout << "  50% quantile: " << quantile(prior, 0.5) << endl;
cout << "  97.5% quantile: " << quantile(prior, 0.975) << endl << endl;
```

This produces this output:

```
Prior distribution:

2.5% quantile: 14.6
50% quantile: 25.9
97.5% quantile: 52.1
```

Based on this distribution, we can now calculate the probability of having a machine working with an unusual work precision (variance) at <= 15 or > 50. For this task, we use calls to the `boost::math::` functions `cdf` and `complement`, respectively, and find a probability of about 0.031 (3.1%) for each case.

```
cout << "  probability variance <= 15: " << boost::math::cdf(prior, 15.0) << endl;
cout << "  probability variance <= 25: " << boost::math::cdf(prior, 25.0) << endl;
cout << "  probability variance > 50: "
  << boost::math::cdf(boost::math::complement(prior, 50.0))
<< endl << endl;
```

This produces this output:

```
probability variance <= 15: 0.031
probability variance <= 25: 0.458
probability variance > 50: 0.0318
```

Therefore, only 3.1% of all precision machines produce balls with a variance of 15 or less (particularly precise machines), but also only 3.2% of all machines produce balls with a variance of as high as 50 or more (particularly imprecise machines). Moreover, slightly more than one-half (1 - 0.458 = 54.2%) of the machines work at a variance greater than 25.

Notice here the distinction between a Bayesian analysis and a frequentist analysis: because we model the variance as random variable itself, we can calculate and straightforwardly interpret probabilities for given parameter values directly, while such an approach is not possible (and interpretationally a strict *must-not*) in the frequentist world.

Step 2: Investigate a single machine

In the second step, we investigate a single machine, which is suspected to suffer from a major fault as the produced balls show fairly high size variability. Based on the prior distribution of generic machinery performance (derived above) and data on balls produced by the suspect machine, we calculate the posterior distribution for that machine and use its properties for guidance regarding continued machine operation or suspension.

It can be shown that if the prior distribution was chosen to be scaled-inverse-chi-square distributed, then the posterior distribution is also scaled-inverse-chi-squared-distributed (prior and posterior distributions are hence conjugate). For more details regarding conjugacy and formula to derive the parameters set for the posterior distribution see Conjugate prior.

Given the prior distribution parameters and sample data (of size n), the posterior distribution parameters are given by the two expressions:

$$\nu_{posterior} = \nu_{prior} + n$$

which gives the posteriorDF below, and

$$s_{posterior} = (\nu_{prior}s_{prior} + \Sigma^{n}_{i=1}(x_i - \mu)^2) / (\nu_{prior} + n)$$

which after some rearrangement gives the formula for the posteriorScale below.

Machine-specific data consist of 100 balls which were accurately measured and show the expected mean of 3000 µm and a sample variance of 55 (calculated for a sample mean defined to be 3000 exactly). From these data, the prior parameterization, and noting that the term $\Sigma^{n}_{i=1}(x_i - \mu)^2$ equals the sample variance multiplied by n - 1, it follows that the posterior distribution of the variance parameter is scaled-inverse-chi-squared distribution with degrees-of-freedom ($\nu_{posterior}$) = 120 and scale ($s_{posterior}$) = 49.54.

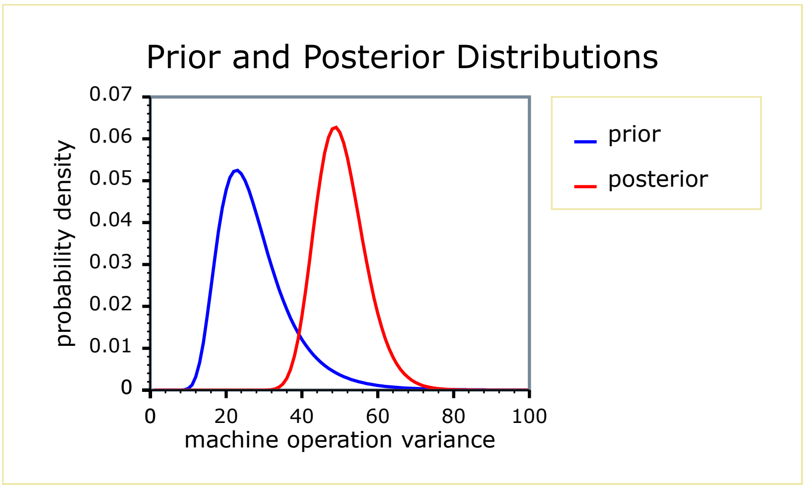
```
int ballsSampleSize = 100;
cout <<"balls sample size: " << ballsSampleSize << endl;
double ballsSampleVariance = 55.0;
cout <<"balls sample variance: " << ballsSampleVariance << endl;

double posteriorDF = priorDF + ballsSampleSize;
cout << "prior degrees-of-freedom: " << priorDF << endl;
cout << "posterior degrees-of-freedom: " << posteriorDF << endl;

double posteriorScale =
  (priorDF * priorScale + (ballsSampleVariance * (ballsSampleSize - 1))) / posteriorDF;
cout << "prior scale: " << priorScale  << endl;
cout << "posterior scale: " << posteriorScale << endl;
```

An interesting feature here is that one needs only to know a summary statistics of the sample to parameterize the posterior distribution: the 100 individual ball measurements are irrelevant, just knowledge of the sample variance and number of measurements is sufficient.

That produces this output:

```
balls sample size: 100
balls sample variance: 55
prior degrees-of-freedom: 20
posterior degrees-of-freedom: 120
prior scale: 25
posterior scale: 49.5
```

To compare the generic machinery performance with our suspect machine, we calculate again the same quantiles and probabilities as above, and find a distribution clearly shifted to greater values (see figure).



```
inverse_chi_squared posterior(posteriorDF, posteriorScale);

 cout << "Posterior distribution:" << endl << endl;
 cout << "  2.5% quantile: " << boost::math::quantile(posterior, 0.025) << endl;
 cout << "  50% quantile: " << boost::math::quantile(posterior, 0.5) << endl;
 cout << "  97.5% quantile: " << boost::math::quantile(posterior, 0.975) << endl << endl;

 cout << "  probability variance <= 15: " << boost::math::cdf(posterior, 15.0) << endl;
 cout << "  probability variance <= 25: " << boost::math::cdf(posterior, 25.0) << endl;
 cout << "  probability variance > 50: "
   << boost::math::cdf(boost::math::complement(posterior, 50.0)) << endl;
```

This produces this output:

```
Posterior distribution:

   2.5% quantile: 39.1
   50% quantile: 49.8
   97.5% quantile: 64.9

   probability variance <= 15: 2.97e-031
   probability variance <= 25: 8.85e-010
   probability variance > 50: 0.489
```

Indeed, the probability that the machine works at a low variance (<= 15) is almost zero, and even the probability of working at average or better performance is negligibly small (less than one-millionth of a permille). On the other hand, with an almost near-half probability (49%), the machine operates in the extreme high variance range of > 50 characteristic for poorly performing machines.

Based on this information the operation of the machine is taken out of use and serviced.

In summary, the Bayesian analysis allowed us to make exact probabilistic statements about a parameter of interest, and hence provided us results with straightforward interpretation.

A full sample output is:

```
Inverse_chi_squared_distribution Bayes example:

  Prior distribution:

   2.5% quantile: 14.6
   50% quantile: 25.9
   97.5% quantile: 52.1

   probability variance <= 15: 0.031
   probability variance <= 25: 0.458
   probability variance > 50: 0.0318

 balls sample size: 100
 balls sample variance: 55
 prior degrees-of-freedom: 20
 posterior degrees-of-freedom: 120
 prior scale: 25
 posterior scale: 49.5
 Posterior distribution:

   2.5% quantile: 39.1
   50% quantile: 49.8
   97.5% quantile: 64.9

   probability variance <= 15: 2.97e-031
   probability variance <= 25: 8.85e-010
   probability variance > 50: 0.489
```

(See also the reference documentation for the Inverse chi squared Distribution.)

See the full source C++ of this example at ../../example/inverse_chi_squared_bayes_eg.cpp

Non Central Chi Squared Example

(See also the reference documentation for the Noncentral Chi Squared Distribution.)

Tables of the power function of the chi^2 test.

This example computes a table of the power of the χ^2 test at the 5% significance level, for various degrees of freedom and non-centrality parameters. The table is deliberately the same as Table 6 from "The Non-Central χ^2 and F-Distributions and their applications.", P. B. Patnaik, Biometrika, Vol. 36, No. 1/2 (June 1949), 202-232.

First we need some includes to access the non-central chi squared distribution (and some basic std output of course).

```cpp
#include <boost/math/distributions/non_central_chi_squared.hpp>
using boost::math::chi_squared;
using boost::math::non_central_chi_squared;

#include <iostream>
using std::cout; using std::endl;
using std::setprecision;

int main()
{
```

Create a table of the power of the χ^2 test at 5% significance level, start with a table header:

```
cout << "[table\n[[[nu]]";
for(int lam = 2; lam <= 20; lam += 2)
{
   cout << "[[lambda]=" << lam << "]";
}
cout << "]\n";
```

(Note: the enclosing [] brackets are to format as a table in Boost.Quickbook).

Enumerate the rows and columns and print the power of the test for each table cell:

```
for(int n = 2; n <= 20; ++n)
{
   cout << "[[" << n << "]";
   for(int lam = 2; lam <= 20; lam += 2)
   {
```

Calculate the χ^2 statistic for a 5% significance:

```
double cs = quantile(complement(chi_squared(n), 0.05));
```

The power of the test is given by the complement of the CDF of the non-central χ^2 distribution:

```
double beta = cdf(complement(non_central_chi_squared(n, lam), cs));
```

Then output the cell value:

```
         cout << "[" << setprecision(3) << beta << "]";
      }
      cout << "]" << endl;
   }
   cout << "]" << endl;
}
```

The output from this program is a table in Boost.Quickbook format as shown below.

We can interpret this as follows - for example if $\nu=10$ and $\lambda=10$ then the power of the test is 0.542 - so we have only a 54% chance of correctly detecting that our null hypothesis is false, and a 46% chance of incurring a type II error (failing to reject the null hypothesis when it is in fact false):

ν	**λ=2**	**λ=4**	**λ=6**	**λ=8**	**λ=10**	**λ=12**	**λ=14**	**λ=16**	**λ=18**	**λ=20**
2	0.226	0.415	0.584	0.718	0.815	0.883	0.928	0.957	0.974	0.985
3	0.192	0.359	0.518	0.654	0.761	0.84	0.896	0.934	0.959	0.975
4	0.171	0.32	0.47	0.605	0.716	0.802	0.866	0.912	0.943	0.964
5	0.157	0.292	0.433	0.564	0.677	0.769	0.839	0.89	0.927	0.952
6	0.146	0.27	0.403	0.531	0.644	0.738	0.813	0.869	0.911	0.94
7	0.138	0.252	0.378	0.502	0.614	0.71	0.788	0.849	0.895	0.928
8	0.131	0.238	0.357	0.477	0.588	0.685	0.765	0.829	0.879	0.915
9	0.125	0.225	0.339	0.454	0.564	0.661	0.744	0.811	0.863	0.903
10	0.121	0.215	0.323	0.435	0.542	0.64	0.723	0.793	0.848	0.891
11	0.117	0.206	0.309	0.417	0.523	0.62	0.704	0.775	0.833	0.878
12	0.113	0.198	0.297	0.402	0.505	0.601	0.686	0.759	0.818	0.866
13	0.11	0.191	0.286	0.387	0.488	0.584	0.669	0.743	0.804	0.854
14	0.108	0.185	0.276	0.374	0.473	0.567	0.653	0.728	0.791	0.842
15	0.105	0.179	0.267	0.362	0.459	0.552	0.638	0.713	0.777	0.83
16	0.103	0.174	0.259	0.351	0.446	0.538	0.623	0.699	0.764	0.819
17	0.101	0.169	0.251	0.341	0.434	0.525	0.609	0.686	0.752	0.807
18	0.0992	0.165	0.244	0.332	0.423	0.512	0.596	0.673	0.74	0.796
19	0.0976	0.161	0.238	0.323	0.412	0.5	0.584	0.66	0.728	0.786
20	0.0961	0.158	0.232	0.315	0.402	0.489	0.572	0.648	0.716	0.775

See nc_chi_sq_example.cpp for the full C++ source code.

Error Handling Example

See error handling documentation for a detailed explanation of the mechanism of handling errors, including the common "bad" arguments to distributions and functions, and how to use Policies to control it.

But, by default, **exceptions will be raised**, for domain errors, pole errors, numeric overflow, and internal evaluation errors. To avoid the exceptions from getting thrown and instead get an appropriate value returned, usually a NaN (domain errors pole errors or internal errors), or infinity (from overflow), you need to change the policy.

The following example demonstrates the effect of setting the macro BOOST_MATH_DOMAIN_ERROR_POLICY when an invalid argument is encountered. For the purposes of this example, we'll pass a negative degrees of freedom parameter to the student's t distribution.

Since we know that this is a single file program we could just add:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error
```

to the top of the source file to change the default policy to one that simply returns a NaN when a domain error occurs. Alternatively we could use:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

To ensure the `::errno` is set when a domain error occurs as well as returning a NaN.

This is safe provided the program consists of a single translation unit *and* we place the define *before* any #includes. Note that should we add the define after the includes then it will have no effect! A warning such as:

```
warning C4005: 'BOOST_MATH_OVERFLOW_ERROR_POLICY' : macro redefinition
```

is a certain sign that it will *not* have the desired effect.

We'll begin our sample program with the needed includes:

```
   #define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error

// Boost
#include <boost/math/distributions/students_t.hpp>
   using boost::math::students_t;  // Probability of students_t(df, t).

// std
#include <iostream>
   using std::cout;
   using std::endl;

#include <stdexcept>


#include <cstddef>
   // using ::errno
```

Next we'll define the program's main() to call the student's t distribution with an invalid degrees of freedom parameter, the program is set up to handle either an exception or a NaN:

```
int main()
{
   cout << "Example error handling using Student's t function. " << endl;
   cout << "BOOST_MATH_DOMAIN_ERROR_POLICY is set to: "
      << BOOST_STRINGIZE(BOOST_MATH_DOMAIN_ERROR_POLICY) << endl;

   double degrees_of_freedom = -1; // A bad argument!
   double t = 10;

   try
   {
      errno = 0; // Clear/reset.
      students_t dist(degrees_of_freedom); // exception is thrown here if enabled.
      double p = cdf(dist, t);
      // Test for error reported by other means:
      if((boost::math::isnan)(p))
      {
         cout << "cdf returned a NaN!" << endl;
         if (errno != 0)
         { // So errno has been set.
           cout << "errno is set to: " << errno << endl;
         }
      }
      else
         cout << "Probability of Student's t is " << p << endl;
   }
   catch(const std::exception& e)
   {
      std::cout <<
         "\n""Message from thrown exception was:\n   " << e.what() << std::endl;
   }
   return 0;
} // int main()
```

Here's what the program output looks like with a default build (one that **does throw exceptions**):

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: throw_on_error

Message from thrown exception was:
   Error in function boost::math::students_t_distribution<double>::students_t_distribution:
   Degrees of freedom argument is -1, but must be > 0 !
```

Alternatively let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error
```

Now the program output is:

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: ignore_error
cdf returned a NaN!
```

And finally let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

Which gives the output show errno:

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: errno_on_error
cdf returned a NaN!
errno is set to: 33
```

Caution

If throwing of exceptions is enabled (the default) but you do **not** have try & catch block, then the program will terminate with an uncaught exception and probably abort.

Therefore to get the benefit of helpful error messages, enabling **all exceptions and using try & catch** is recommended for most applications.

However, for simplicity, the is not done for most examples.

Find Location and Scale Examples

Find Location (Mean) Example

First we need some includes to access the normal distribution, the algorithms to find location (and some std output of course).

```
#include <boost/math/distributions/normal.hpp> // for normal_distribution
  using boost::math::normal; // typedef provides default type is double.
#include <boost/math/distributions/cauchy.hpp> // for cauchy_distribution
  using boost::math::cauchy; // typedef provides default type is double.
#include <boost/math/distributions/find_location.hpp>
  using boost::math::find_location; // for mean
#include <boost/math/distributions/find_scale.hpp>
  using boost::math::find_scale; // for standard devation
  using boost::math::complement; // Needed if you want to use the complement version.
  using boost::math::policies::policy;

#include <iostream>
  using std::cout; using std::endl;
#include <iomanip>
  using std::setw; using std::setprecision;
#include <limits>
  using std::numeric_limits;
```

For this example, we will use the standard normal distribution, with mean (location) zero and standard deviation (scale) unity. This is also the default for this implementation.

```
normal N01;  // Default 'standard' normal distribution with zero mean and
double sd = 1.; // normal default standard deviation is 1.
```

Suppose we want to find a different normal distribution whose mean is shifted so that only fraction p (here 0.001 or 0.1%) are below a certain chosen limit (here -2, two standard deviations).

```
double z = -2.; // z to give prob p
double p = 0.001; // only 0.1% below z

cout << "Normal distribution with mean = " << N01.location()
  << ", standard deviation " << N01.scale()
  << ", has " << "fraction <= " << z
  << ", p = "  << cdf(N01, z) << endl;
cout << "Normal distribution with mean = " << N01.location()
  << ", standard deviation " << N01.scale()
  << ", has " << "fraction > " << z
  << ", p = "  << cdf(complement(N01, z)) << endl; // Note: uses complement.
```

```
Normal distribution with mean = 0, standard deviation 1, has fraction <= -2, p = 0.0227501
Normal distribution with mean = 0, standard deviation 1, has fraction > -2, p = 0.97725
```

We can now use "find_location" to give a new offset mean.

```
double l = find_location<normal>(z, p, sd);
cout << "offset location (mean) = " << l << endl;
```

that outputs:

```
offset location (mean) = 1.09023
```

showing that we need to shift the mean just over one standard deviation from its previous value of zero.

Then we can check that we have achieved our objective by constructing a new distribution with the offset mean (but same standard deviation):

```
normal np001pc(l, sd); // Same standard_deviation (scale) but with mean (location) shifted.
```

And re-calculating the fraction below our chosen limit.

```
cout << "Normal distribution with mean = " << l
    << " has " << "fraction <= " << z
    << ", p = "  << cdf(np001pc, z) << endl;
  cout << "Normal distribution with mean = " << l
    << " has " << "fraction > " << z
    << ", p = "  << cdf(complement(np001pc, z)) << endl;
```

```
Normal distribution with mean = 1.09023 has fraction <= -2, p = 0.001
Normal distribution with mean = 1.09023 has fraction > -2, p = 0.999
```

Controlling Error Handling from find_location

We can also control the policy for handling various errors. For example, we can define a new (possibly unwise) policy to ignore domain errors ('bad' arguments).

Unless we are using the boost::math namespace, we will need:

```
using boost::math::policies::policy;
using boost::math::policies::domain_error;
using boost::math::policies::ignore_error;
```

Using a typedef is often convenient, especially if it is re-used, although it is not required, as the various examples below show.

```
typedef policy<domain_error<ignore_error> > ignore_domain_policy;
// find_location with new policy, using typedef.
l = find_location<normal>(z, p, sd, ignore_domain_policy());
// Default policy policy<>, needs "using boost::math::policies::policy;"
l = find_location<normal>(z, p, sd, policy<>());
// Default policy, fully specified.
l = find_location<normal>(z, p, sd, boost::math::policies::policy<>());
// A new policy, ignoring domain errors, without using a typedef.
l = find_location<normal>(z, p, sd, policy<domain_error<ignore_error> >());
```

If we want to use a probability that is the complements of our probability, we should not even think of writing `find_location<normal>(z, 1 - p, sd)`, but use the complement version, see why complements?.

```
z = 2.;
double q = 0.95; // = 1 - p; // complement.
l = find_location<normal>(complement(z, q, sd));

normal np95pc(l, sd); // Same standard_deviation (scale) but with mean(location) shifted
cout << "Normal distribution with mean = " << l << " has "
  << "fraction <= " << z << " = "  << cdf(np95pc, z) << endl;
cout << "Normal distribution with mean = " << l << " has "
  << "fraction > " << z << " = "  << cdf(complement(np95pc, z)) << endl;
```

See find_location_example.cpp for full source code: the program output looks like this:

```
Example: Find location (mean).
Normal distribution with mean = 0, standard deviation 1, has fraction <= -2, p = 0.0227501
Normal distribution with mean = 0, standard deviation 1, has fraction > -2, p = 0.97725
offset location (mean) = 1.09023
Normal distribution with mean = 1.09023 has fraction <= -2, p = 0.001
Normal distribution with mean = 1.09023 has fraction > -2, p = 0.999
Normal distribution with mean = 0.355146 has fraction <= 2 = 0.95
Normal distribution with mean = 0.355146 has fraction > 2 = 0.05
```

Find Scale (Standard Deviation) Example

First we need some includes to access the Normal Distribution, the algorithms to find scale (and some std output of course).

```
#include <boost/math/distributions/normal.hpp> // for normal_distribution
  using boost::math::normal; // typedef provides default type is double.
#include <boost/math/distributions/find_scale.hpp>
  using boost::math::find_scale;
  using boost::math::complement; // Needed if you want to use the complement version.
  using boost::math::policies::policy; // Needed to specify the error handling policy.

#include <iostream>
  using std::cout; using std::endl;
#include <iomanip>
  using std::setw; using std::setprecision;
#include <limits>
  using std::numeric_limits;
```

For this example, we will use the standard Normal Distribution, with location (mean) zero and standard deviation (scale) unity. Conveniently, this is also the default for this implementation's constructor.

```
normal N01;  // Default 'standard' normal distribution with zero mean
double sd = 1.; // and standard deviation is 1.
```

Suppose we want to find a different normal distribution with standard deviation so that only fraction p (here 0.001 or 0.1%) are below a certain chosen limit (here -2. standard deviations).

```
double z = -2.; // z to give prob p
double p = 0.001; // only 0.1% below z = -2

cout << "Normal distribution with mean = " << N01.location()  // aka N01.mean()
  << ", standard deviation " << N01.scale() // aka N01.standard_deviation()
  << ", has " << "fraction <= " << z
  << ", p = "  << cdf(N01, z) << endl;
cout << "Normal distribution with mean = " << N01.location()
  << ", standard deviation " << N01.scale()
  << ", has " << "fraction > " << z
  << ", p = "  << cdf(complement(N01, z)) << endl; // Note: uses complement.
```

```
Normal distribution with mean = 0 has fraction <= -2, p = 0.0227501
Normal distribution with mean = 0 has fraction > -2, p = 0.97725
```

Noting that p = 0.02 instead of our target of 0.001, we can now use `find_scale` to give a new standard deviation.

```
double l = N01.location();
double s = find_scale<normal>(z, p, l);
cout << "scale (standard deviation) = " << s << endl;
```

that outputs:

```
scale (standard deviation) = 0.647201
```

showing that we need to reduce the standard deviation from 1. to 0.65.

Then we can check that we have achieved our objective by constructing a new distribution with the new standard deviation (but same zero mean):

```
normal np001pc(N01.location(), s);
```

And re-calculating the fraction below (and above) our chosen limit.

```
cout << "Normal distribution with mean = " << l
  << " has " << "fraction <= " << z
  << ", p = "  << cdf(np001pc, z) << endl;
cout << "Normal distribution with mean = " << l
  << " has " << "fraction > " << z
  << ", p = "  << cdf(complement(np001pc, z)) << endl;
```

```
Normal distribution with mean = 0 has fraction <= -2, p = 0.001
Normal distribution with mean = 0 has fraction > -2, p = 0.999
```

Controlling how Errors from find_scale are handled

We can also control the policy for handling various errors. For example, we can define a new (possibly unwise) policy to ignore domain errors ('bad' arguments).

Unless we are using the boost::math namespace, we will need:

```
using boost::math::policies::policy;
using boost::math::policies::domain_error;
using boost::math::policies::ignore_error;
```

Using a typedef is convenient, especially if it is re-used, although it is not required, as the various examples below show.

```
typedef policy<domain_error<ignore_error> > ignore_domain_policy;
// find_scale with new policy, using typedef.
l = find_scale<normal>(z, p, l, ignore_domain_policy());
// Default policy policy<>, needs using boost::math::policies::policy;

l = find_scale<normal>(z, p, l, policy<>());
// Default policy, fully specified.
l = find_scale<normal>(z, p, l, boost::math::policies::policy<>());
// New policy, without typedef.
l = find_scale<normal>(z, p, l, policy<domain_error<ignore_error> >());
```

If we want to express a probability, say 0.999, that is a complement, `1 - p` we should not even think of writing `find_scale<normal>(z, 1 - p, l)`, but use the complements version (see why complements?).

```
z = -2.;
double q = 0.999; // = 1 - p; // complement of 0.001.
sd = find_scale<normal>(complement(z, q, l));

normal np95pc(l, sd); // Same standard_deviation (scale) but with mean(scale) shifted
cout << "Normal distribution with mean = " << l << " has "
  << "fraction <= " << z << " = "  << cdf(np95pc, z) << endl;
cout << "Normal distribution with mean = " << l << " has "
  << "fraction > " << z << " = "  << cdf(complement(np95pc, z)) << endl;
```

Sadly, it is all too easy to get probabilities the wrong way round, when you may get a warning like this:

```
Message from thrown exception was:
   Error in function boost::math::find_scale<Dist, Policy>(complement(double, double, double, ↵
Policy)):
    Computed scale (-0.48043523852179076) is <= 0! Was the complement intended?
```

The default error handling policy is to throw an exception with this message, but if you chose a policy to ignore the error, the (impossible) negative scale is quietly returned.

See find_scale_example.cpp for full source code: the program output looks like this:

```
Example: Find scale (standard deviation).
Normal distribution with mean = 0, standard deviation 1, has fraction <= -2, p = 0.0227501
Normal distribution with mean = 0, standard deviation 1, has fraction > -2, p = 0.97725
scale (standard deviation) = 0.647201
Normal distribution with mean = 0 has fraction <= -2, p = 0.001
Normal distribution with mean = 0 has fraction > -2, p = 0.999
Normal distribution with mean = 0.946339 has fraction <= -2 = 0.001
Normal distribution with mean = 0.946339 has fraction > -2 = 0.999
```

Find mean and standard deviation example

First we need some includes to access the normal distribution, the algorithms to find location and scale (and some std output of course).

```
#include <boost/math/distributions/normal.hpp> // for normal_distribution
  using boost::math::normal; // typedef provides default type is double.
#include <boost/math/distributions/cauchy.hpp> // for cauchy_distribution
  using boost::math::cauchy; // typedef provides default type is double.
#include <boost/math/distributions/find_location.hpp>
  using boost::math::find_location;
#include <boost/math/distributions/find_scale.hpp>
  using boost::math::find_scale;
  using boost::math::complement;
  using boost::math::policies::policy;

#include <iostream>
  using std::cout; using std::endl; using std::left; using std::showpoint; using std::noshow↵
point;
#include <iomanip>
  using std::setw; using std::setprecision;
#include <limits>
  using std::numeric_limits;
#include <stdexcept>
```

Using find_location and find_scale to meet dispensing and measurement specifications

Consider an example from K Krishnamoorthy, Handbook of Statistical Distributions with Applications, ISBN 1-58488-635-8, (2006) p 126, example 10.3.7.

"A machine is set to pack 3 kg of ground beef per pack. Over a long period of time it is found that the average packed was 3 kg with a standard deviation of 0.1 kg. Assume the packing is normally distributed."

We start by constructing a normal distribution with the given parameters:

```
double mean = 3.; // kg
double standard_deviation = 0.1; // kg
normal packs(mean, standard_deviation);
```

We can then find the fraction (or %) of packages that weigh more than 3.1 kg.

```
double max_weight = 3.1; // kg
cout << "Percentage of packs > " << max_weight << " is "
<< cdf(complement(packs, max_weight)) * 100. << endl; // P(X > 3.1)
```

We might want to ensure that 95% of packs are over a minimum weight specification, then we want the value of the mean such that P(X < 2.9) = 0.05.

Using the mean of 3 kg, we can estimate the fraction of packs that fail to meet the specification of 2.9 kg.

```
double minimum_weight = 2.9;
cout <<"Fraction of packs <= " << minimum_weight << " with a mean of " << mean
  << " is " << cdf(complement(packs, minimum_weight)) << endl;
// fraction of packs <= 2.9 with a mean of 3 is 0.841345
```

This is 0.84 - more than the target fraction of 0.95. If we want 95% to be over the minimum weight, what should we set the mean weight to be?

Using the KK StatCalc program supplied with the book and the method given on page 126 gives 3.06449.

We can confirm this by constructing a new distribution which we call 'xpacks' with a safety margin mean of 3.06449 thus:

```
double over_mean = 3.06449;
normal xpacks(over_mean, standard_deviation);
cout << "Fraction of packs >= " << minimum_weight
<< " with a mean of " << xpacks.mean()
  << " is " << cdf(complement(xpacks, minimum_weight)) << endl;
// fraction of packs >= 2.9 with a mean of 3.06449 is 0.950005
```

Using this Math Toolkit, we can calculate the required mean directly thus:

```
double under_fraction = 0.05;  // so 95% are above the minimum weight mean - sd = 2.9
double low_limit = standard_deviation;
double offset = mean - low_limit - quantile(packs, under_fraction);
double nominal_mean = mean + offset;
// mean + (mean - low_limit - quantile(packs, under_fraction));

normal nominal_packs(nominal_mean, standard_deviation);
cout << "Setting the packer to " << nominal_mean << " will mean that "
  << "fraction of packs >= " << minimum_weight
  << " is " << cdf(complement(nominal_packs, minimum_weight)) << endl;
// Setting the packer to 3.06449 will mean that fraction of packs >= 2.9 is 0.95
```

This calculation is generalized as the free function called `find_location`, see algorithms.

To use this we will need to

```
#include <boost/math/distributions/find_location.hpp>
  using boost::math::find_location;
```

and then use find_location function to find safe_mean, & construct a new normal distribution called 'goodpacks'.

```
double safe_mean = find_location<normal>(minimum_weight, under_fraction, standard_deviation);
normal good_packs(safe_mean, standard_deviation);
```

with the same confirmation as before:

```
cout << "Setting the packer to " << nominal_mean << " will mean that "
  << "fraction of packs >= " << minimum_weight
  << " is " << cdf(complement(good_packs, minimum_weight)) << endl;
// Setting the packer to 3.06449 will mean that fraction of packs >= 2.9 is 0.95
```

Using Cauchy-Lorentz instead of normal distribution

After examining the weight distribution of a large number of packs, we might decide that, after all, the assumption of a normal distribution is not really justified. We might find that the fit is better to a Cauchy Distribution. This distribution has wider 'wings', so that whereas most of the values are closer to the mean than the normal, there are also more values than 'normal' that lie further from the mean than the normal.

This might happen because a larger than normal lump of meat is either included or excluded.

We first create a Cauchy Distribution with the original mean and standard deviation, and estimate the fraction that lie below our minimum weight specification.

```
cauchy cpacks(mean, standard_deviation);
cout << "Cauchy Setting the packer to " << mean << " will mean that "
  << "fraction of packs >= " << minimum_weight
  << " is " << cdf(complement(cpacks, minimum_weight)) << endl;
// Cauchy Setting the packer to 3 will mean that fraction of packs >= 2.9 is 0.75
```

Note that far fewer of the packs meet the specification, only 75% instead of 95%. Now we can repeat the find_location, using the cauchy distribution as template parameter, in place of the normal used above.

```
double lc = find_location<cauchy>(minimum_weight, under_fraction, standard_deviation);
cout << "find_location<cauchy>(minimum_weight, over fraction, standard_deviation); ↵
" << lc << endl;
// find_location<cauchy>(minimum_weight, over fraction, packs.standard_deviation()); 3.53138
```

Note that the safe_mean setting needs to be much higher, 3.53138 instead of 3.06449, so we will make rather less profit.

And again confirm that the fraction meeting specification is as expected.

```
cauchy goodcpacks(lc, standard_deviation);
cout << "Cauchy Setting the packer to " << lc << " will mean that "
  << "fraction of packs >= " << minimum_weight
  << " is " << cdf(complement(goodcpacks, minimum_weight)) << endl;
// Cauchy Setting the packer to 3.53138 will mean that fraction of packs >= 2.9 is 0.95
```

Finally we could estimate the effect of a much tighter specification, that 99% of packs met the specification.

```
cout << "Cauchy Setting the packer to "
  << find_location<cauchy>(minimum_weight, 0.99, standard_deviation)
  << " will mean that "
  << "fraction of packs >= " << minimum_weight
  << " is " << cdf(complement(goodcpacks, minimum_weight)) << endl;
```

Setting the packer to 3.13263 will mean that fraction of packs >= 2.9 is 0.99, but will more than double the mean loss from 0.0644 to 0.133 kg per pack.

Of course, this calculation is not limited to packs of meat, it applies to dispensing anything, and it also applies to a 'virtual' material like any measurement.

The only caveat is that the calculation assumes that the standard deviation (scale) is known with a reasonably low uncertainty, something that is not so easy to ensure in practice. And that the distribution is well defined, Normal Distribution or Cauchy Distribution, or some other.

If one is simply dispensing a very large number of packs, then it may be feasible to measure the weight of hundreds or thousands of packs. With a healthy 'degrees of freedom', the confidence intervals for the standard deviation are not too wide, typically about + and - 10% for hundreds of observations.

For other applications, where it is more difficult or expensive to make many observations, the confidence intervals are depressingly wide.

See Confidence Intervals on the standard deviation for a worked example chi_square_std_dev_test.cpp of estimating these intervals.

Changing the scale or standard deviation

Alternatively, we could invest in a better (more precise) packer (or measuring device) with a lower standard deviation, or scale.

This might cost more, but would reduce the amount we have to 'give away' in order to meet the specification.

To estimate how much better (how much smaller standard deviation) it would have to be, we need to get the 5% quantile to be located at the under_weight limit, 2.9

```
double p = 0.05; // wanted p th quantile.
cout << "Quantile of " << p << " = " << quantile(packs, p)
  << ", mean = " << packs.mean() << ", sd = " << packs.standard_deviation() << endl;
```

Quantile of 0.05 = 2.83551, mean = 3, sd = 0.1

With the current packer (mean = 3, sd = 0.1), the 5% quantile is at 2.8551 kg, a little below our target of 2.9 kg. So we know that the standard deviation is going to have to be smaller.

Let's start by guessing that it (now 0.1) needs to be halved, to a standard deviation of 0.05 kg.

```
normal pack05(mean, 0.05);
cout << "Quantile of " << p << " = " << quantile(pack05, p)
  << ", mean = " << pack05.mean() << ", sd = " << pack05.standard_deviation() << endl;
// Quantile of 0.05 = 2.91776, mean = 3, sd = 0.05

cout <<"Fraction of packs >= " << minimum_weight << " with a mean of " << mean
  << " and standard deviation of " << pack05.standard_deviation()
  << " is " << cdf(complement(pack05, minimum_weight)) << endl;
// Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.05 is 0.97725
```

So 0.05 was quite a good guess, but we are a little over the 2.9 target, so the standard deviation could be a tiny bit more. So we could do some more guessing to get closer, say by increasing standard deviation to 0.06 kg, constructing another new distribution called pack06.

```
normal pack06(mean, 0.06);
cout << "Quantile of " << p << " = " << quantile(pack06, p)
  << ", mean = " << pack06.mean() << ", sd = " << pack06.standard_deviation() << endl;
// Quantile of 0.05 = 2.90131, mean = 3, sd = 0.06

cout <<"Fraction of packs >= " << minimum_weight << " with a mean of " << mean
  << " and standard deviation of " << pack06.standard_deviation()
  << " is " << cdf(complement(pack06, minimum_weight)) << endl;
// Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.06 is 0.95221
```

Now we are getting really close, but to do the job properly, we might need to use root finding method, for example the tools provided, and used elsewhere, in the Math Toolkit, see root-finding without derivatives

But in this (normal) distribution case, we can and should be even smarter and make a direct calculation.

Our required limit is minimum_weight = 2.9 kg, often called the random variate z. For a standard normal distribution, then probability p = N((minimum_weight - mean) / sd).

We want to find the standard deviation that would be required to meet this limit, so that the p th quantile is located at z (minimum_weight). In this case, the 0.05 (5%) quantile is at 2.9 kg pack weight, when the mean is 3 kg, ensuring that 0.95 (95%) of packs are above the minimum weight.

Rearranging, we can directly calculate the required standard deviation:

```
normal N01; // standard normal distribution with mean zero and unit standard deviation.
p = 0.05;
double qp = quantile(N01, p);
double sd95 = (minimum_weight - mean) / qp;

cout << "For the "<< p << "th quantile to be located at "
  << minimum_weight << ", would need a standard deviation of " << sd95 << endl;
// For the 0.05th quantile to be located at 2.9, would need a standard deviation of 0.0607957
```

We can now construct a new (normal) distribution pack95 for the 'better' packer, and check that our distribution will meet the specification.

```
normal pack95(mean, sd95);
cout <<"Fraction of packs >= " << minimum_weight << " with a mean of " << mean
  << " and standard deviation of " << pack95.standard_deviation()
  << " is " << cdf(complement(pack95, minimum_weight)) << endl;
// Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.0607957 is 0.95
```

This calculation is generalized in the free function find_scale, as shown below, giving the same standard deviation.

```
double ss = find_scale<normal>(minimum_weight, under_fraction, packs.mean());
cout << "find_scale<normal>(minimum_weight, under_fraction, packs.mean()); " << ss << endl;
// find_scale<normal>(minimum_weight, under_fraction, packs.mean()); 0.0607957
```

If we had defined an over_fraction, or percentage that must pass specification

```
double over_fraction = 0.95;
```

And (wrongly) written

```
double sso = find_scale<normal>(minimum_weight, over_fraction, packs.mean());
```

With the default policy, we would get a message like

```
Message from thrown exception was:
   Error in function boost::math::find_scale<Dist, Policy>(double, double, double, Policy):
   Computed scale (-0.060795683191176959) is <= 0! Was the complement intended?
```

But this would return a **negative** standard deviation - obviously impossible. The probability should be 1 - over_fraction, not over_fraction, thus:

```
double ss1o = find_scale<normal>(minimum_weight, 1 - over_fraction, packs.mean());
cout << "find_scale<normal>(minimum_weight, under_fraction, packs.mean()); " << ss1o << endl;
// find_scale<normal>(minimum_weight, under_fraction, packs.mean()); 0.0607957
```

But notice that using '1 - over_fraction' - will lead to a loss of accuracy, especially if over_fraction was close to unity. (See why complements?). In this (very common) case, we should instead use the complements, giving the most accurate result.

```
double ssc = find_scale<normal>(complement(minimum_weight, over_fraction, packs.mean()));
cout << "find_scale<normal>(complement(minimum_weight, over_fraction, packs.mean())); ↵
" << ssc << endl;
// find_scale<normal>(complement(minimum_weight, over_fraction, packs.mean())); 0.0607957
```

Note that our guess of 0.06 was close to the accurate value of 0.060795683191176959.

We can again confirm our prediction thus:

```
normal pack95c(mean, ssc);
cout <<"Fraction of packs >= " << minimum_weight << " with a mean of " << mean
  << " and standard deviation of " << pack95c.standard_deviation()
  << " is " << cdf(complement(pack95c, minimum_weight)) << endl;
// Fraction of packs >= 2.9 with a mean of 3 and standard deviation of 0.0607957 is 0.95
```

Notice that these two deceptively simple questions:

- Do we over-fill to make sure we meet a minimum specification (or under-fill to avoid an overdose)?

and/or

- Do we measure better?

are actually extremely common.

The weight of beef might be replaced by a measurement of more or less anything, from drug tablet content, Apollo landing rocket firing, X-ray treatment doses...

The scale can be variation in dispensing or uncertainty in measurement.

See find_mean_and_sd_normal.cpp for full source code & appended program output.

Comparison with C, R, FORTRAN-style Free Functions

You are probably familiar with a statistics library that has free functions, for example the classic NAG C library and matching NAG FORTRAN Library, Microsoft Excel BINOMDIST(number_s,trials,probability_s,cumulative), R, MathCAD pbinom and many others.

If so, you may find 'Distributions as Objects' unfamiliar, if not alien.

However, **do not panic**, both definition and usage are not really very different.

A very simple example of generating the same values as the NAG C library for the binomial distribution follows. (If you find slightly different values, the Boost C++ version, using double or better, is very likely to be the more accurate. Of course, accuracy is not usually a concern for most applications of this function).

The NAG function specification is

```
void nag_binomial_dist(Integer n, double p, Integer k,
double *plek, double *pgtk, double *peqk, NagError *fail)
```

and is called

```
g01bjc(n, p, k, &plek, &pgtk, &peqk, NAGERR_DEFAULT);
```

The equivalent using this Boost C++ library is:

```
using namespace boost::math;  // Using declaration avoids very long names.
binomial my_dist(4, 0.5); // c.f. NAG n = 4, p = 0.5
```

and values can be output thus:

```
cout
  << my_dist.trials() << " "              // Echo the NAG input n = 4 trials.
  << my_dist.success_fraction() << " "    // Echo the NAG input p = 0.5
  << cdf(my_dist, 2) << "  "              // NAG plek with k = 2
  << cdf(complement(my_dist, 2)) << "  "  // NAG pgtk with k = 2
  << pdf(my_dist, 2) << endl;             // NAG peqk with k = 2
```

`cdf(dist, k)` is equivalent to NAG library `plek`, lower tail probability of <= k

`cdf(complement(dist, k))` is equivalent to NAG library `pgtk`, upper tail probability of > k

`pdf(dist, k)` is equivalent to NAG library `peqk`, point probability of == k

See binomial_example_nag.cpp for details.

Using the Distributions from Within C#

The distributions in this library can be used from the C# programming language when they are built using Microsoft's Common Language Runtime (CLR) option.

An example of this kind of usage is given in the Distribution Explorer example. See `boost-root/libs/math/dot_net_example` for the source code: the application consists of a C++ .dll that contains the actual distributions, and a C# GUI that allows you to explore their properties.

Random Variates and Distribution Parameters

Random variates and distribution parameters are conventionally distinguished (for example in Wikipedia and Wolfram MathWorld by placing a semi-colon after the random variate (whose value you 'choose'), to separate the variate from the parameter(s) that defines the shape of the distribution.

For example, the binomial distribution has two parameters: n (the number of trials) and p (the probability of success on one trial). It also has the random variate *k*: the number of successes observed. This means the probability density/mass function (pdf) is written as *f(k; n, p)*.

Translating this into code the `binomial_distribution` constructor therefore has two parameters:

```
binomial_distribution(RealType n, RealType p);
```

While the function `pdf` has one argument specifying the distribution type (which includes its parameters, if any), and a second argument for the random variate. So taking our binomial distribution example, we would write:

```
pdf(binomial_distribution<RealType>(n, p), k);
```

Discrete Probability Distributions

Note that the discrete distributions, including the binomial, negative binomial, Poisson & Bernoulli, are all mathematically defined as discrete functions: only integral values of the random variate are envisaged and the functions are only defined at these integral values. However because the method of calculation often uses continuous functions, it is convenient to treat them as if they were continuous functions, and permit non-integral values of their parameters.

To enforce a strict mathematical model, users may use floor or ceil functions on the random variate, prior to calling the distribution function, to enforce integral values.

For similar reasons, in continuous distributions, parameters like degrees of freedom that might appear to be integral, are treated as real values (and are promoted from integer to floating-point if necessary). In this case however, that there are a small number of situations where non-integral degrees of freedom do have a genuine meaning.

Generally speaking there is no loss of performance from allowing real-values parameters: the underlying special functions contain optimizations for integer-valued arguments when applicable.

Caution

The quantile function of a discrete distribution will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on a discrete distribution. The reference docs describe how to change the rounding policy for these distributions.

Statistical Distributions Reference

Non-Member Properties

Properties that are common to all distributions are accessed via non-member getter functions: non-membership allows more of these functions to be added over time, as the need arises. Unfortunately the literature uses many different and confusing names to refer to a rather small number of actual concepts; refer to the concept index to find the property you want by the name you are most familiar with. Or use the function index to go straight to the function you want if you already know its name.

Function Index

Conceptual Index

- kurtosis.
- kurtosis_excess
- mean.
- median.
- mode.
- P.
- Percent Point Function.
- Probability Density Function.
- Probability Mass Function.
- range.
- Q.
- Quantile.
- Quantile from the complement of the probability.
- skewness.
- standard deviation
- Survival Function.
- support.
- Upper Critical Value.
- variance.

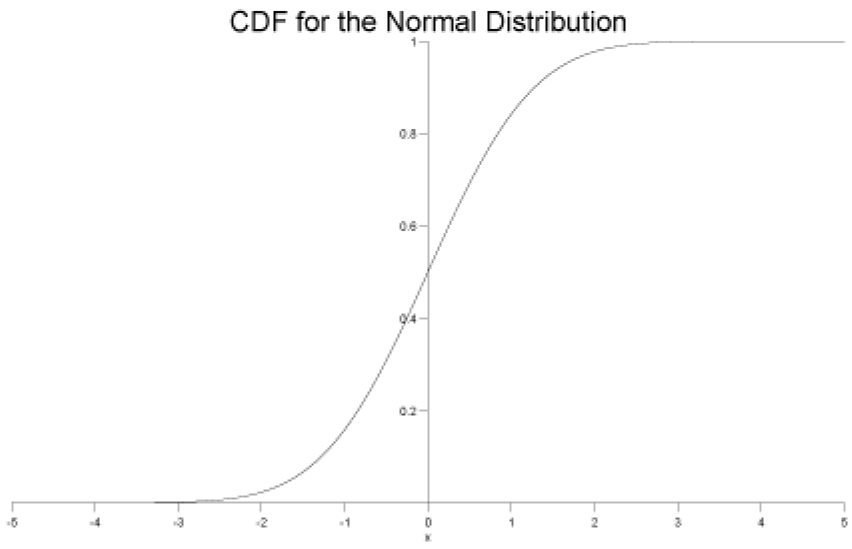
Cumulative Distribution Function

```
template <class RealType, class Policy>
RealType cdf(const Distribution-Type<RealType, Policy>& dist, const RealType& x);
```

The Cumulative Distribution Function is the probability that the variable takes a value less than or equal to x. It is equivalent to the integral from -infinity to x of the Probability Density Function.

This function may return a domain_error if the random variable is outside the defined range for the distribution.

For example, the following graph shows the cdf for the normal distribution:



Complement of the Cumulative Distribution Function

```
template <class Distribution, class RealType>
RealType cdf(const Unspecified-Complement-Type<Distribution, RealType>& comp);
```

The complement of the Cumulative Distribution Function is the probability that the variable takes a value greater than x. It is equivalent to the integral from x to infinity of the Probability Density Function, or 1 minus the Cumulative Distribution Function of x.

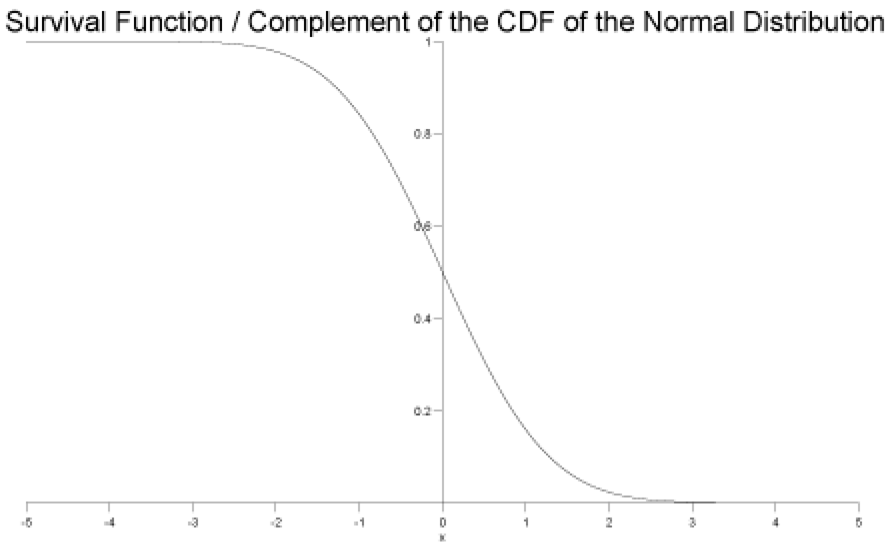
This is also known as the survival function.

This function may return a domain_error if the random variable is outside the defined range for the distribution.

In this library, it is obtained by wrapping the arguments to the `cdf` function in a call to `complement`, for example:

```
// standard normal distribution object:
boost::math::normal norm;
// print survival function for x=2.0:
std::cout << cdf(complement(norm, 2.0)) << std::endl;
```

For example, the following graph shows the __complement of the cdf for the normal distribution:



See why complements? for why the complement is useful and when it should be used.

Hazard Function

```
template <class RealType, class Policy>
RealType hazard(const Distribution-Type<RealType, Policy>& dist, const RealType& x);
```

Returns the Hazard Function of *x* and distibution *dist*.

This function may return a domain_error if the random variable is outside the defined range for the distribution.

$$\operatorname{hazard}(x) = h(x) = \frac{\operatorname{pdf}(x)}{1 - \operatorname{cdf}(x)}$$

Caution

Some authors refer to this as the conditional failure density function rather than the hazard function.

Cumulative Hazard Function

```
template <class RealType, class Policy>
RealType chf(const Distribution-Type<RealType, Policy>& dist, const RealType& x);
```

Returns the Cumulative Hazard Function of *x* and distibution *dist*.

This function may return a domain_error if the random variable is outside the defined range for the distribution.

$$\operatorname{chf}(dist, x) = H(x) = \int_{-\infty}^{x} h(\mu) d\mu$$

Caution

Some authors refer to this as simply the "Hazard Function".

mean

```
template<class RealType, class Policy>
RealType mean(const Distribution-Type<RealType, Policy>& dist);
```

Returns the mean of the distribution *dist*.

This function may return a domain_error if the distribution does not have a defined mean (for example the Cauchy distribution).

median

```
template<class RealType, class Policy>
RealType median(const Distribution-Type<RealType, Policy>& dist);
```

Returns the median of the distribution *dist*.

mode

```
template<class RealType, Policy>
RealType mode(const Distribution-Type<RealType, Policy>& dist);
```

Returns the mode of the distribution *dist*.

This function may return a domain_error if the distribution does not have a defined mode.

Probability Density Function

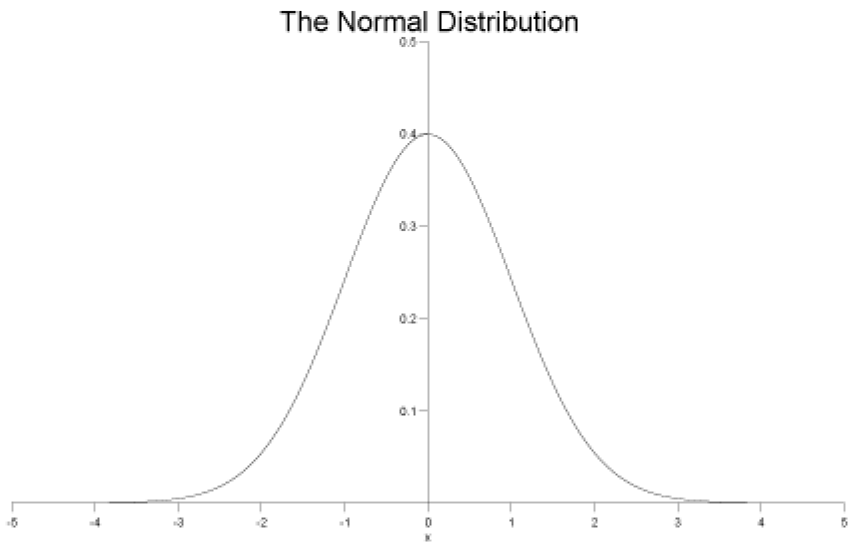
```
template <class RealType, class Policy>
RealType pdf(const Distribution-Type<RealType, Policy>& dist, const RealType& x);
```

For a continuous function, the probability density function (pdf) returns the probability that the variate has the value x. Since for continuous distributions the probability at a single point is actually zero, the probability is better expressed as the integral of the pdf between two points: see the Cumulative Distribution Function.

For a discrete distribution, the pdf is the probability that the variate takes the value x.

This function may return a domain_error if the random variable is outside the defined range for the distribution.

For example, for a standard normal distribution the pdf looks like this:



Range

```
template<class RealType, class Policy>
std::pair<RealType, RealType> range(const Distribution-Type<RealType, Policy>& dist);
```

Returns the valid range of the random variable over distribution *dist*.

Quantile

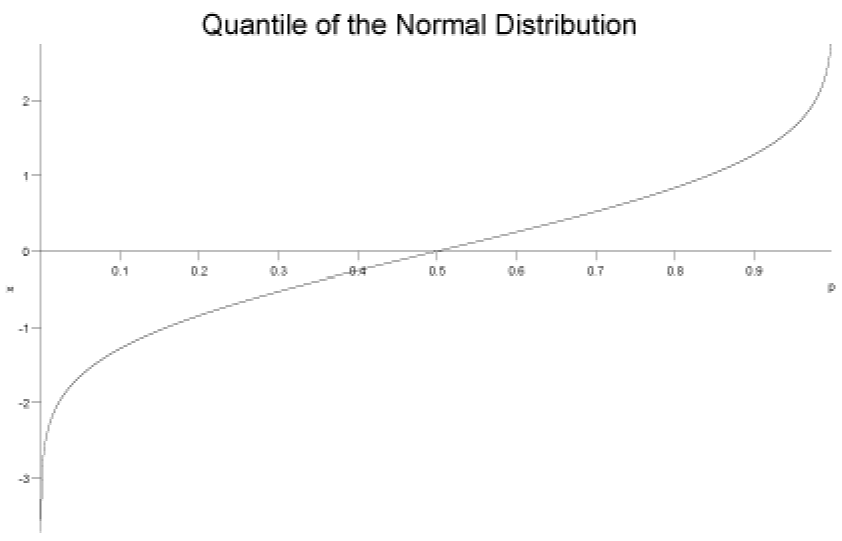
```
template <class RealType, class Policy>
RealType quantile(const Distribution-Type<RealType, Policy>& dist, const RealType& p);
```

The quantile is best viewed as the inverse of the Cumulative Distribution Function, it returns a value *x* such that `cdf(dist, x) == p`.

This is also known as the *percent point function*, or *percentile*, or *fractile*, it is also the same as calculating the *lower critical value* of a distribution.

This function returns a domain_error if the probability lies outside [0,1]. The function may return an overflow_error if there is no finite value that has the specified probability.

The following graph shows the quantile function for a standard normal distribution:



Quantile from the complement of the probability.

See also complements.

```
template <class Distribution, class RealType>
RealType quantile(const Unspecified-Complement-Type<Distribution, RealType>& comp);
```

This is the inverse of the Complement of the Cumulative Distribution Function. It is calculated by wrapping the arguments in a call to the quantile function in a call to *complement*. For example:

```
// define a standard normal distribution:
boost::math::normal norm;
// print the value of x for which the complement
// of the probability is 0.05:
std::cout << quantile(complement(norm, 0.05)) << std::endl;
```

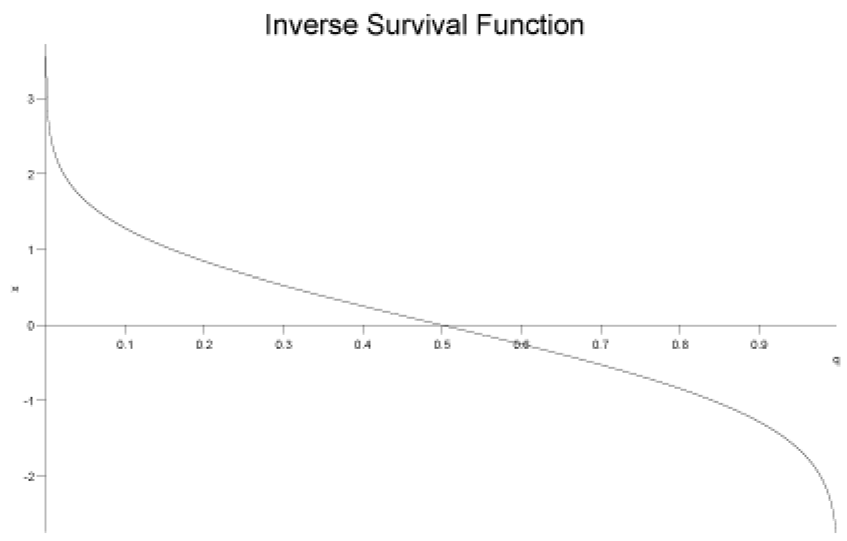
The function computes a value *x* such that `cdf(complement(dist, x)) == q` where *q* is complement of the probability.

Why complements?

This function is also called the inverse survival function, and is the same as calculating the *upper critical value* of a distribution.

This function returns a domain_error if the probablity lies outside [0,1]. The function may return an overflow_error if there is no finite value that has the specified probability.

The following graph show the inverse survival function for the normal distribution:



Standard Deviation

```
template <class RealType, class Policy>
RealType standard_deviation(const Distribution-Type<RealType, Policy>& dist);
```

Returns the standard deviation of distribution *dist*.

This function may return a domain_error if the distribution does not have a defined standard deviation.

support

```
template<class RealType, class Policy>
std::pair<RealType, RealType> support(const Distribution-Type<RealType, Policy>& dist);
```

Returns the supported range of random variable over the distribution *dist*.

The distribution is said to be 'supported' over a range that is "the smallest closed set whose complement has probability zero". Non-mathematicians might say it means the 'interesting' smallest range of random variate x that has the cdf going from zero to unity. Outside are uninteresting zones where the pdf is zero, and the cdf zero or unity.

Variance

```
template <class RealType, class Policy>
RealType variance(const Distribution-Type<RealType, Policy>& dist);
```

Returns the variance of the distribution *dist*.

This function may return a domain_error if the distribution does not have a defined variance.

Skewness

```
template <class RealType, class Policy>
RealType skewness(const Distribution-Type<RealType, Policy>& dist);
```

Returns the skewness of the distribution *dist*.

This function may return a domain_error if the distribution does not have a defined skewness.

Kurtosis

```
template <class RealType, class Policy>
RealType kurtosis(const Distribution-Type<RealType, Policy>& dist);
```

Returns the 'proper' kurtosis (normalized fourth moment) of the distribution *dist*.

kertosis = $\beta_2 = \mu_4 / \mu_2^2$

Where μ_i is the i'th central moment of the distribution, and in particular μ_2 is the variance of the distribution.

The kurtosis is a measure of the "peakedness" of a distribution.

Note that the literature definition of kurtosis is confusing. The definition used here is that used by for example Wolfram MathWorld (that includes a table of formulae for kurtosis excess for various distributions) but NOT the definition of kurtosis used by Wikipedia which treats "kurtosis" and "kurtosis excess" as the same quantity.

```
kurtosis_excess = 'proper' kurtosis - 3
```

This subtraction of 3 is convenient so that the *kurtosis excess* of a normal distribution is zero.

This function may return a domain_error if the distribution does not have a defined kurtosis.

'Proper' kurtosis can have a value from zero to + infinity.

Kurtosis excess

```
template <class RealType, Policy>
RealType kurtosis_excess(const Distribution-Type<RealType, Policy>& dist);
```

Returns the kurtosis excess of the distribution *dist*.

kurtosis excess = $\gamma_2 = \mu_4 / \mu_2^2 - 3$ = kurtosis - 3

Where μ_i is the i'th central moment of the distribution, and in particular μ_2 is the variance of the distribution.

The kurtosis excess is a measure of the "peakedness" of a distribution, and is more widely used than the "kurtosis proper". It is defined so that the kurtosis excess of a normal distribution is zero.

This function may return a domain_error if the distribution does not have a defined kurtosis excess.

Kurtosis excess can have a value from -2 to + infinity.

```
kurtosis = kurtosis_excess +3;
```

The kurtosis excess of a normal distribution is zero.

P and Q

The terms P and Q are sometimes used to refer to the Cumulative Distribution Function and its complement respectively. Lowercase p and q are sometimes used to refer to the values returned by these functions.

Percent Point Function or Percentile

The percent point function, also known as the percentile, is the same as the Quantile.

Inverse CDF Function.

The inverse of the cumulative distribution function, is the same as the Quantile.

Inverse Survival Function.

The inverse of the survival function, is the same as computing the quantile from the complement of the probability.

Probability Mass Function

The Probability Mass Function is the same as the Probability Density Function.

The term Mass Function is usually applied to discrete distributions, while the term Probability Density Function applies to continuous distributions.

Lower Critical Value.

The lower critical value calculates the value of the random variable given the area under the left tail of the distribution. It is equivalent to calculating the Quantile.

Upper Critical Value.

The upper critical value calculates the value of the random variable given the area under the right tail of the distribution. It is equivalent to calculating the quantile from the complement of the probability.

Survival Function

Refer to the Complement of the Cumulative Distribution Function.

Distributions

Arcsine Distribution

```
#include <boost/math/distributions/arcsine.hpp>
```

```
namespace boost{ namespace math{

 template <class RealType = double,
           class Policy   = policies::policy<> >
class arcsine_distribution;

typedef arcsine_distribution<double> arcsine; // double precision standard arcsine distribu↵
tion [0,1].

template <class RealType, class Policy>
class arcsine_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor from two range parameters, x_min and x_max:
   arcsine_distribution(RealType x_min, RealType x_max);

   // Range Parameter accessors:
   RealType x_min() const;
   RealType x_max() const;
};
}} // namespaces
```

The class type `arcsine_distribution` represents an arcsine probability distribution function. The arcsine distribution is named because its CDF uses the inverse sin^{-1} or arcsine.

This is implemented as a generalized version with support from *x_min* to *x_max* providing the 'standard arcsine distribution' as default with *x_min = 0* and *x_max = 1*. (A few make other choices for 'standard').

The arcsine distribution is generalized to include any bounded support *a <= x <= b* by Wolfram and Wikipedia, but also using *location* and *scale* parameters by Virtual Laboratories in Probability and Statistics Arcsine distribution. The end-point version is simpler and more obvious, so we implement that. If desired, this outlines how the Beta Distribution can be used to add a shape factor.

The probability density function PDF for the arcsine distribution defined on the interval [*x_min, x_max*] is given by:

f(x; x_min, x_max) = 1 /(π·√((x - x_min)·(x_max - x_min))

For example, Wolfram Alpha arcsine distribution, from input of

```
N[PDF[arcsinedistribution[0, 1], 0.5], 50]
```

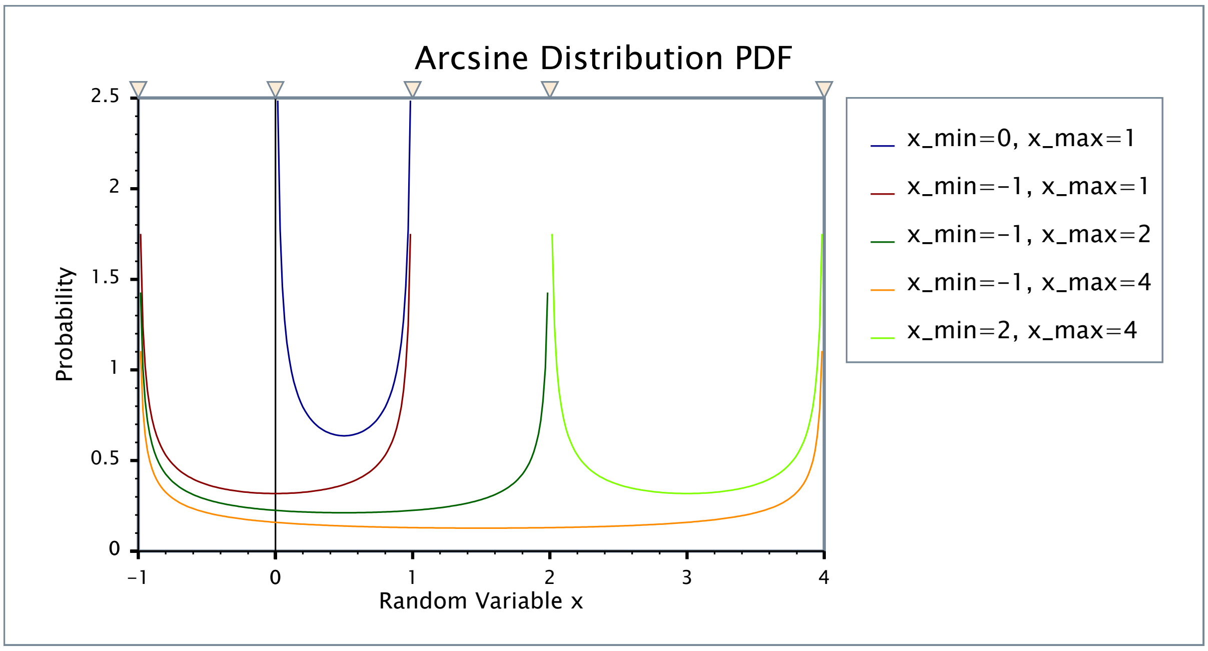
computes the PDF value

```
0.63661977236758134307553505349005744813783858296183
```

The Probability Density Functions (PDF) of generalized arcsine distributions are symmetric U-shaped curves, centered on *(x_max - x_min)/2*, highest (infinite) near the two extrema, and quite flat over the central region.

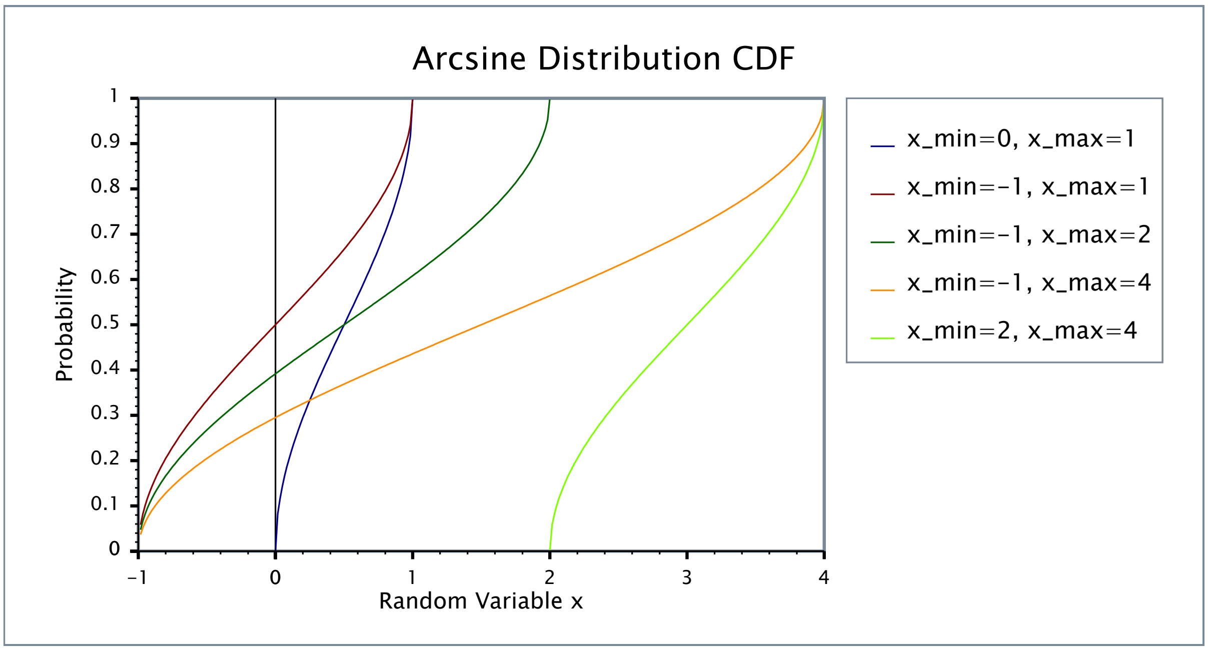
If random variate *x* is *x_min* or *x_max*, then the PDF is infinity. If random variate *x* is *x_min* then the CDF is zero. If random variate *x* is *x_max* then the CDF is unity.

The 'Standard' (0, 1) arcsine distribution is shown in blue and some generalized examples with other *x* ranges.



The Cumulative Distribution Function CDF is defined as

F(x) = 2·arcsin(√((x-x_min)/(x_max - x))) / π



Constructor

```
arcsine_distribution(RealType x_min, RealType x_max);
```

constructs an arcsine distribution with range parameters *x_min* and *x_max*.

Requires *x_min* < *x_max*, otherwise domain_error is called.

For example:

```
arcsine_distribution<> myarcsine(-2, 4);
```

constructs an arcsine distribution with *x_min* = *-2* and *x_max* = *4*.

Default values of *x_min* = *0* and *x_max* = *1* and a `typedef arcsine_distribution<double> arcsine;` mean that

```
arcsine as;
```

constructs a 'Standard 01' arcsine distribution.

Parameter Accessors

```
RealType x_min() const;
RealType x_max() const;
```

Return the parameter *x_min* or *x_max* from which this distribution was constructed.

So, for example:

```
using boost::math::arcsine_distribution;

arcsine_distribution<> as(2, 5); // Cconstructs a double arcsine distribution.
assert(as.x_min() == 2.);  // as.x_min() returns 2.
assert(as.x_max() == 5.);   // as.x_max()  returns 5.
```

Non-member Accessor Functions

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The formulae for calculating these are shown in the table below, and at Wolfram Mathworld.

Note

There are always **two** values for the **mode**, at *x_min* and at *x_max*, default 0 and 1, so instead we raise the exception domain_error. At these extrema, the PDFs are infinite, and the CDFs zero or unity.

Applications

The arcsine distribution is useful to describe Random walks, (including drunken walks) Brownian motion, Weiner processes, Bernoulli trials, and their appplication to solve stock market and other ruinous gambling games.

The random variate *x* is constrained to *x_min* and *x_max*, (for our 'standard' distribution, 0 and 1), and is usually some fraction. For any other *x_min* and *x_max* a fraction can be obtained from *x* using

fraction = (x - x_min) / (x_max - x_min)

The simplest example is tossing heads and tails with a fair coin and modelling the risk of losing, or winning. Walkers (molecules, drunks...) moving left or right of a centre line are another common example.

The random variate *x* is the fraction of time spent on the 'winning' side. If half the time is spent on the 'winning' side (and so the other half on the 'losing' side) then *x* = *1/2*.

For large numbers of tosses, this is modelled by the (standard [0,1]) arcsine distribution, and the PDF can be calculated thus:

```
std::cout << pdf(as, 1. / 2) << std::endl; // 0.637
// pdf has a minimum at x = 0.5
```

From the plot of PDF, it is clear that *x* = ½ is the **minimum** of the curve, so this is the **least likely** scenario. (This is highly counter-intuitive, considering that fair tosses must **eventually** become equal. It turns out that *eventually* is not just very long, but **infinite**!).

The **most likely** scenarios are towards the extrema where *x* = 0 or *x* = 1.

If fraction of time on the left is a ¼, it is only slightly more likely because the curve is quite flat bottomed.

```
std::cout << pdf(as, 1. / 4) << std::endl; // 0.735
```

If we consider fair coin-tossing games being played for 100 days (hypothetically continuously to be 'at-limit') the person winning after day 5 will not change in fraction 0.144 of the cases.

We can easily compute this setting *x* = 5./100 = 0.05

```
std::cout << cdf(as, 0.05) << std::endl; // 0.144
```

Similarly, we can compute from a fraction of 0.05 /2 = 0.025 (halved because we are considering both winners and losers) corresponding to 1 - 0.025 or 97.5% of the gamblers, (walkers, particles...) on the **same side** of the origin

```
std::cout << 2 * cdf(as, 1 - 0.975) << std::endl; // 0.202
```

(use of the complement gives a bit more clarity, and avoids potential loss of accuracy when *x* is close to unity, see why complements?).

```
std::cout << 2 * cdf(complement(as, 0.975)) << std::endl; // 0.202
```

or we can reverse the calculation by assuming a fraction of time on one side, say fraction 0.2,

```
std::cout << quantile(as, 1 - 0.2 / 2) << std::endl; //  0.976

std::cout << quantile(complement(as, 0.2 / 2)) << std::endl; // 0.976
```

Summary: Every time we toss, the odds are equal, so on average we have the same change of winning and losing.

But this is **not true** for an an individual game where one will be **mostly in a bad or good patch**.

This is quite counter-intuitive to most people, but the mathematics is clear, and gamblers continue to provide proof.

Moral: if you in a losing patch, leave the game. (Because the odds to recover to a good patch are poor).

Corollary: Quit while you are ahead?

A working example is at arcsine_example.cpp including sample output .

Related distributions

The arcsine distribution with *x_min = 0* and *x_max = 1* is special case of the Beta Distribution with α = 1/2 and β = 1/2.

Accuracy

This distribution is implemented using sqrt, sine, cos and arc sine and cos trigonometric functions which are normally accurate to a few machine epsilon. But all values suffer from loss of significance or cancellation error for values of *x* close to *x_max*. For example, for a standard [0, 1] arcsine distribution *as*, the pdf is symmetric about random variate *x = 0.5* so that one would expect `pdf(as, 0.01) == pdf(as, 0.99)`. But as *x* nears unity, there is increasing loss of significance. To counteract this, the complement versions of CDF and quantile are implemented with alternative expressions using cos^{-1} instead of sin^{-1}. Users should see why complements? for guidance on when to avoid loss of accuracy by using complements.

Testing

The results were tested against a few accurate spot values computed by Wolfram Alpha, for example:

```
N[PDF[arcsinedistribution[0, 1], 0.5], 50]
  0.63661977236758134307553505349005744813783858296183
```

Implementation

In the following table *a* and *b* are the parameters *x_min* and *x_max*, *x* is the random variable, *p* is the probability and its complement *q = 1-p*.

Function	Implementation Notes
support	x ∈ [a, b], default x ∈ [0, 1]
pdf	f(x; a, b) = 1/(π·√(x - a)·(b - x))
cdf	$F(x) = 2/\pi \cdot \sin^{-1}(\sqrt{(x - a) / (b - a)})$
cdf of complement	$2/(\pi \cdot \cos^{-1}(\sqrt{(x - a) / (b - a)}))$
quantile	$-a \cdot \sin^2(½\pi \cdot p) + a + b \cdot \sin^2(½\pi \cdot p)$
quantile from the complement	$-a \cdot \cos^2(½\pi \cdot p) + a + b \cdot \cos^2(½\pi \cdot q)$
mean	½(a+b)
median	½(a+b)
mode	x ∈ [a, b], so raises domain_error (returning NaN).
variance	$(b - a)^2 / 8$
skewness	0
kurtosis excess	-3/2
kurtosis	kurtosis_excess + 3

The quantile was calculated using an expression obtained by using Wolfram Alpha to invert the formula for the CDF thus

```
solve [p - 2/pi sin^-1(sqrt((x-a)/(b-a))) = 0, x]
```

which was interpreted as

```
Solve[p - (2 ArcSin[Sqrt[(-a + x)/(-a + b)]])/Pi == 0, x, MaxExtraConditions -> Automatic]
```

and produced the resulting expression

```
x = -a sin^2((pi p)/2)+a+b sin^2((pi p)/2)
```

Thanks to Wolfram for providing this facility.

References

- Wikipedia arcsine distribution
- Wikipedia Beta distribution
- Wolfram MathWorld
- Wolfram Alpha

Sources

- The probability of going through a bad patch Esteban Moro's Blog.
- What soschumcks and the arc sine have in common Peter Haggstrom.
- arcsine distribution.
- Wolfram reference arcsine examples.
- Shlomo Sternberg slides.

Bernoulli Distribution

```
#include <boost/math/distributions/bernoulli.hpp>
```

```
namespace boost{ namespace math{
 template <class RealType = double,
           class Policy   = policies::policy<> >
 class bernoulli_distribution;

 typedef bernoulli_distribution<> bernoulli;

 template <class RealType, class Policy>
 class bernoulli_distribution
 {
 public:
    typedef RealType  value_type;
    typedef Policy    policy_type;

    bernoulli_distribution(RealType p); // Constructor.
    // Accessor function.
    RealType success_fraction() const
    // Probability of success (as a fraction).
 };
}} // namespaces
```

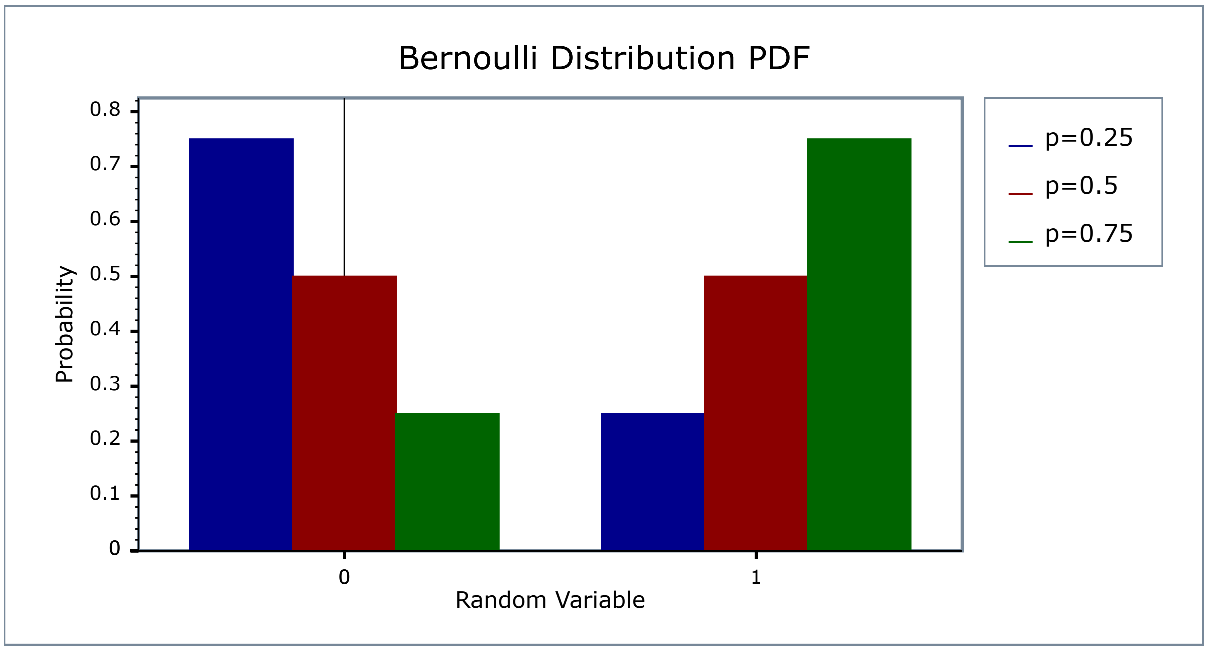
The Bernoulli distribution is a discrete distribution of the outcome of a single trial with only two results, 0 (failure) or 1 (success), with a probability of success p.

The Bernoulli distribution is the simplest building block on which other discrete distributions of sequences of independent Bernoulli trials can be based.

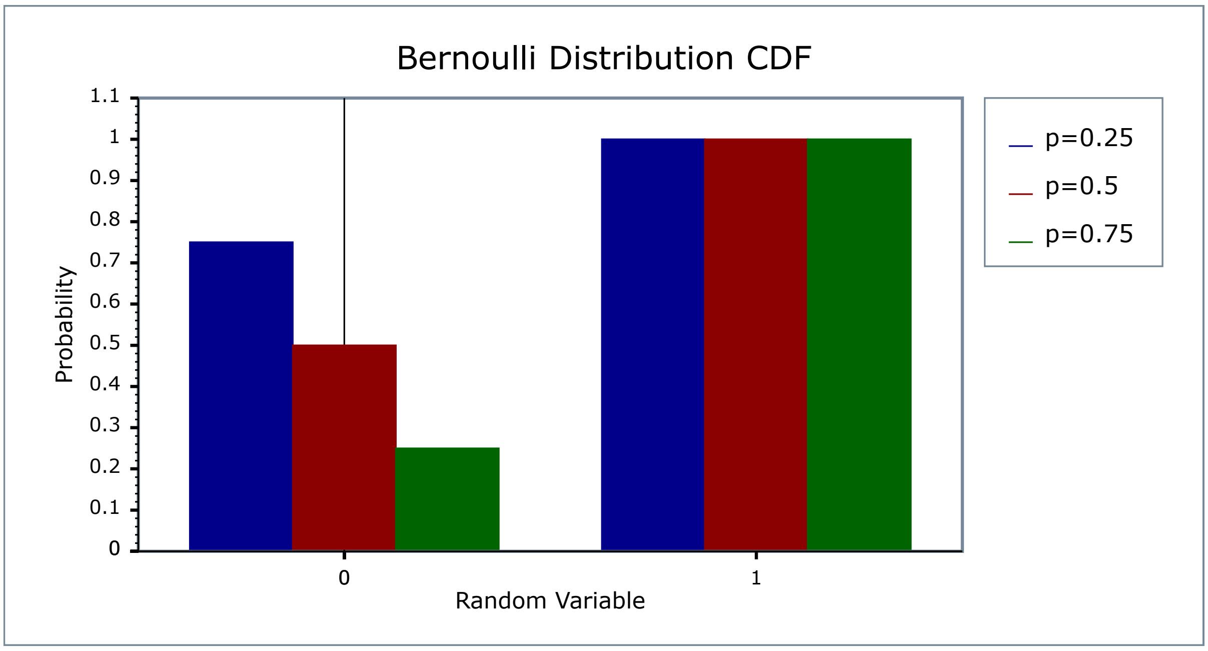
The Bernoulli is the binomial distribution (k = 1, p) with only one trial.

probability density function pdf f(0) = 1 - p, f(1) = p. Cumulative distribution function D(k) = if (k == 0) 1 - p else 1.

The following graph illustrates how the probability density function pdf varies with the outcome of the single trial:



and the Cumulative distribution function



Member Functions

```
bernoulli_distribution(RealType p);
```

Constructs a bernoulli distribution with success_fraction *p*.

```
RealType success_fraction() const
```

Returns the *success_fraction* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is 0 and 1, and the useful supported range is only 0 or 1.

Outside this range, functions are undefined, or may throw domain_error exception and make an error message available.

Accuracy

The Bernoulli distribution is implemented with simple arithmetic operators and so should have errors within an epsilon or two.

Implementation

In the following table *p* is the probability of success and *q = 1-p*. *k* is the random variate, either 0 or 1.

Note

The Bernoulli distribution is implemented here as a *strict discrete* distribution. If a generalised version, allowing k to be any real, is required then the binomial distribution with a single trial should be used, for example:

```
binomial_distribution(1, 0.25)
```

Function	Implementation Notes
Supported range	{0, 1}
pdf	Using the relation: pdf = 1 - p for k = 0, else p
cdf	Using the relation: cdf = 1 - p for k = 0, else 1
cdf complement	q = 1 - p
quantile	if x <= (1-p) 0 else 1
quantile from the complement	if x <= (1-p) 1 else 0
mean	p
variance	p * (1 - p)
mode	if (p < 0.5) 0 else 1
skewness	(1 - 2 * p) / sqrt(p * q)
kurtosis	6 * p * p - 6 * p +1/ p * q
kurtosis excess	kurtosis -3

References

- Wikpedia Bernoulli distribution
- Weisstein, Eric W. "Bernoulli Distribution." From MathWorld--A Wolfram Web Resource.

Beta Distribution

```
#include <boost/math/distributions/beta.hpp>
```

```
namespace boost{ namespace math{

 template <class RealType = double,
           class Policy   = policies::policy<> >
class beta_distribution;

// typedef beta_distribution<double> beta;
// Note that this is deliberately NOT provided,
// to avoid a clash with the function name beta.

template <class RealType, class Policy>
class beta_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;
   // Constructor from two shape parameters, alpha & beta:
   beta_distribution(RealType a, RealType b);

   // Parameter accessors:
   RealType alpha() const;
   RealType beta() const;

   // Parameter estimators of alpha or beta from mean and variance.
   static RealType find_alpha(
     RealType mean, // Expected value of mean.
     RealType variance); // Expected value of variance.

   static RealType find_beta(
     RealType mean, // Expected value of mean.
     RealType variance); // Expected value of variance.

   // Parameter estimators from
   // either alpha or beta, and x and probability.

   static RealType find_alpha(
     RealType beta, // from beta.
     RealType x, //  x.
     RealType probability); // cdf

   static RealType find_beta(
     RealType alpha, // alpha.
     RealType x, // probability x.
     RealType probability); // probability cdf.
};

}} // namespaces
```

The class type `beta_distribution` represents a beta probability distribution function.

The beta distribution is used as a prior distribution for binomial proportions in Bayesian analysis.

See also: beta distribution and Bayesian statistics.

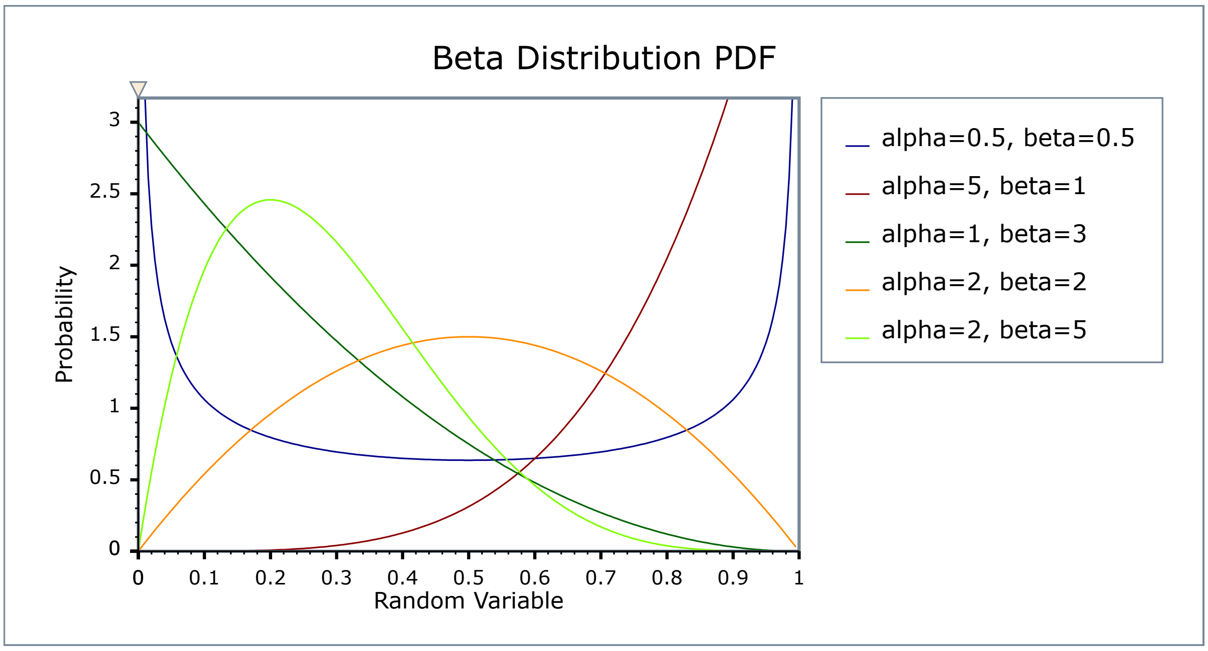
How the beta distribution is used for Bayesian analysis of one parameter models is discussed by Jeff Grynaviski.

The probability density function PDF for the beta distribution defined on the interval [0,1] is given by:

$f(x;\alpha,\beta) = x^{\alpha - 1} (1 - x)^{\beta -1} / B(\alpha, \beta)$

where B(α, β) is the beta function, implemented in this library as beta. Division by the beta function ensures that the pdf is normalized to the range zero to unity.

The following graph illustrates examples of the pdf for various values of the shape parameters. Note the α = β = 2 (blue line) is dome-shaped, and might be approximated by a symmetrical triangular distribution.



If α = β = 1, then it is a __space uniform distribution, equal to unity in the entire interval x = 0 to 1. If α __space and β __space are < 1, then the pdf is U-shaped. If α != β, then the shape is asymmetric and could be approximated by a triangle whose apex is away from the centre (where x = half).

Member Functions

Constructor

```
beta_distribution(RealType alpha, RealType beta);
```

Constructs a beta distribution with shape parameters *alpha* and *beta*.

Requires alpha,beta > 0,otherwise domain_error is called. Note that technically the beta distribution is defined for alpha,beta >= 0, but it's not clear whether any program can actually make use of that latitude or how many of the non-member functions can be usefully defined in that case. Therefore for now, we regard it as an error if alpha or beta is zero.

For example:

```
beta_distribution<> mybeta(2, 5);
```

Constructs a the beta distribution with alpha=2 and beta=5 (shown in yellow in the graph above).

Parameter Accessors

```
RealType alpha() const;
```

Returns the parameter *alpha* from which this distribution was constructed.

```
RealType beta() const;
```

Returns the parameter *beta* from which this distribution was constructed.

So for example:

```
beta_distribution<> mybeta(2, 5);
assert(mybeta.alpha() == 2.);  // mybeta.alpha() returns 2
assert(mybeta.beta() == 5.);   // mybeta.beta()  returns 5
```

Parameter Estimators

Two pairs of parameter estimators are provided.

One estimates either α __space or β __space from presumed-known mean and variance.

The other pair estimates either α __space or β __space from the cdf and x.

It is also possible to estimate α __space and β __space from 'known' mode & quantile. For example, calculators are provided by the Pooled Prevalence Calculator and Beta Buster but this is not yet implemented here.

```
static RealType find_alpha(
  RealType mean, // Expected value of mean.
  RealType variance); // Expected value of variance.
```

Returns the unique value of α that corresponds to a beta distribution with mean *mean* and variance *variance*.

```
static RealType find_beta(
  RealType mean, // Expected value of mean.
  RealType variance); // Expected value of variance.
```

Returns the unique value of β that corresponds to a beta distribution with mean *mean* and variance *variance*.

```
static RealType find_alpha(
  RealType beta, // from beta.
  RealType x, //  x.
  RealType probability); // probability cdf
```

Returns the value of α that gives: `cdf(beta_distribution<RealType>(alpha, beta), x) == probability`.

```
static RealType find_beta(
  RealType alpha, // alpha.
  RealType x, // probability x.
  RealType probability); // probability cdf.
```

Returns the value of β that gives: `cdf(beta_distribution<RealType>(alpha, beta), x) == probability`.

Non-member Accessor Functions

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The formulae for calculating these are shown in the table below, and at Wolfram Mathworld.

Applications

The beta distribution can be used to model events constrained to take place within an interval defined by a minimum and maximum value: so it is used in project management systems.

It is also widely used in Bayesian statistical inference.

Related distributions

The beta distribution with both α __space and β = 1 follows a uniform distribution.

The triangular is used when less precise information is available.

The binomial distribution is closely related when α __space and β __space are integers.

With integer values of α __space and β __space the distribution B(i, j) is that of the j-th highest of a sample of i + j + 1 independent random variables uniformly distributed between 0 and 1. The cumulative probability from 0 to x is thus the probability that the j-th highest value is less than x. Or it is the probability that at least i of the random variables are less than x, a probability given by summing over the Binomial Distribution with its p parameter set to x.

Accuracy

This distribution is implemented using the beta functions beta and incomplete beta functions ibeta and ibetac; please refer to these functions for information on accuracy.

Implementation

In the following table *a* and *b* are the parameters α and β, *x* is the random variable, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	$f(x;\alpha,\beta) = x^{\alpha - 1} (1 - x)^{\beta -1} / B(\alpha, \beta)$ Implemented using ibeta_derivative(a, b, x).
cdf	Using the incomplete beta function ibeta(a, b, x)
cdf complement	ibetac(a, b, x)
quantile	Using the inverse incomplete beta function ibeta_inv(a, b, p)
quantile from the complement	ibetac_inv(a, b, q)
mean	`a/(a+b)`
variance	`a * b / (a+b)^2 * (a + b + 1)`
mode	`(a-1) / (a + b - 2)`
skewness	`2 (b-a) sqrt(a+b+1)/(a+b+2) * sqrt(a * b)`
kurtosis excess	$6\frac{\alpha^3 - \alpha^2(2\beta - 1) + \beta^2(\beta - 1) - 2\alpha\beta(\beta + 2)}{\alpha\beta(\alpha + \beta + 2)(\alpha + \beta + 3)}$
kurtosis	`kurtosis + 3`
parameter estimation	
alpha from mean and variance	`mean * (( (mean * (1 - mean)) / variance)- 1)`
beta from mean and variance	`(1 - mean) * (((mean * (1 - mean)) /variance)-1)`
The member functions `find_alpha` and `find_beta` from cdf and probability x and **either** `alpha` or `beta`	Implemented in terms of the inverse incomplete beta functions ibeta_inva, and ibeta_invb respectively.
`find_alpha`	`ibeta_inva(beta, x, probability)`
`find_beta`	`ibeta_invb(alpha, x, probability)`

References

Wikipedia Beta distribution

NIST Exploratory Data Analysis

Wolfram MathWorld

Binomial Distribution

```
#include <boost/math/distributions/binomial.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class binomial_distribution;

typedef binomial_distribution<> binomial;

template <class RealType, class Policy>
class binomial_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   static const unspecified-type clopper_pearson_exact_interval;
   static const unspecified-type jeffreys_prior_interval;

   // construct:
   binomial_distribution(RealType n, RealType p);

   // parameter access::
   RealType success_fraction() const;
   RealType trials() const;

   // Bounds on success fraction:
   static RealType find_lower_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability,
      unspecified-type method = clopper_pearson_exact_interval);
   static RealType find_upper_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability,
      unspecified-type method = clopper_pearson_exact_interval);

   // estimate min/max number of trials:
   static RealType find_minimum_number_of_trials(
      RealType k,     // number of events
      RealType p,     // success fraction
      RealType alpha); // risk level

   static RealType find_maximum_number_of_trials(
      RealType k,     // number of events
      RealType p,     // success fraction
      RealType alpha); // risk level
};

}} // namespaces
```

The class type `binomial_distribution` represents a binomial distribution: it is used when there are exactly two mutually exclusive outcomes of a trial. These outcomes are labelled "success" and "failure". The Binomial Distribution is used to obtain the probability of observing k successes in N trials, with the probability of success on a single trial denoted by p. The binomial distribution assumes that p is fixed for all trials.

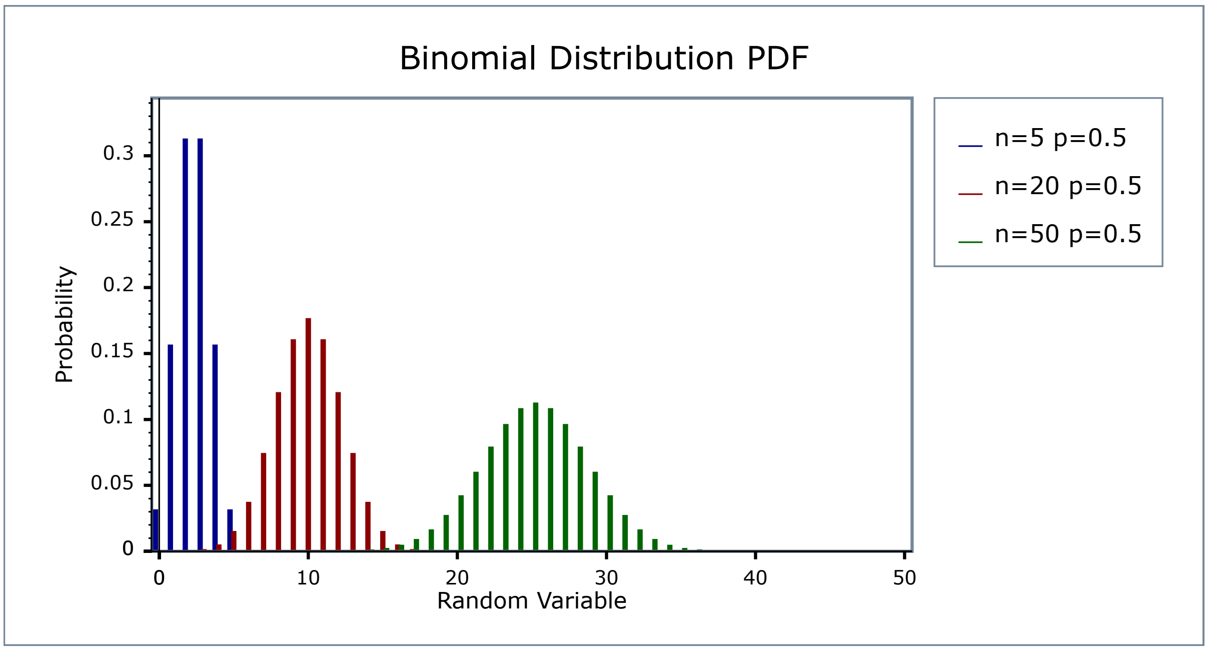
Note

The random variable for the binomial distribution is the number of successes, (the number of trials is a fixed property of the distribution) whereas for the negative binomial, the random variable is the number of trials, for a fixed number of successes.

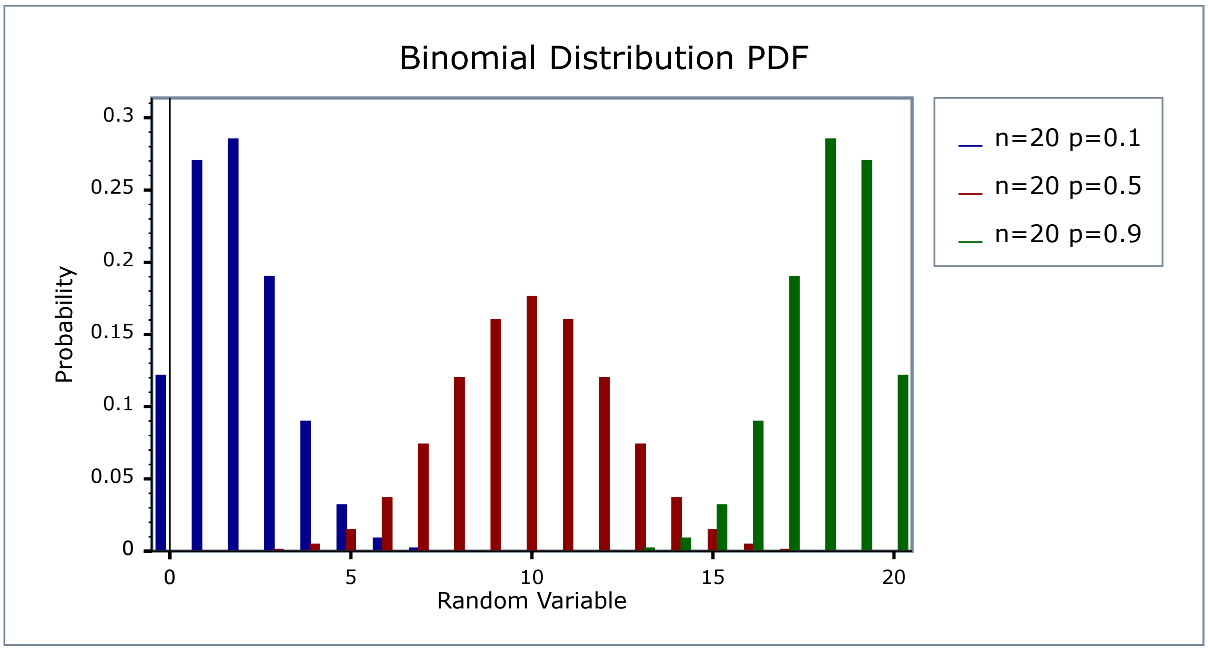
The PDF for the binomial distribution is given by:

$$\begin{aligned} f(k; n, p) &= {}_nC_k p^k (1-p)^{n-k} \\ &= \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} \end{aligned}$$

The following two graphs illustrate how the PDF changes depending upon the distributions parameters, first we'll keep the success fraction *p* fixed at 0.5, and vary the sample size:



Alternatively, we can keep the sample size fixed at N=20 and vary the success fraction *p*:



Caution

The Binomial distribution is a discrete distribution: internally, functions like the `cdf` and `pdf` are treated "as if" they are continuous functions, but in reality the results returned from these functions only have meaning if an integer value is provided for the random variate argument.

The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on the Binomial distribution. The reference docs describe how to change the rounding policy for these distributions.

Member Functions

Construct

```
binomial_distribution(RealType n, RealType p);
```

Constructor: *n* is the total number of trials, *p* is the probability of success of a single trial.

Requires `0 <= p <= 1`, and `n >= 0`, otherwise calls domain_error.

Accessors

```
RealType success_fraction() const;
```

Returns the parameter *p* from which this distribution was constructed.

```
RealType trials() const;
```

Returns the parameter *n* from which this distribution was constructed.

Lower Bound on the Success Fraction

```
static RealType find_lower_bound_on_p(
   RealType trials,
   RealType successes,
   RealType alpha,
   unspecified-type method = clopper_pearson_exact_interval);
```

Returns a lower bound on the success fraction:

trials	The total number of trials conducted.
successes	The number of successes that occurred.
alpha	The largest acceptable probability that the true value of the success fraction is **less than** the value returned.
method	An optional parameter that specifies the method to be used to compute the interval (See below).

For example, if you observe *k* successes from *n* trials the best estimate for the success fraction is simply *k/n*, but if you want to be 95% sure that the true value is **greater than** some value, p_{min}, then:

```
pmin = binomial_distribution<RealType>::find_lower_bound_on_p(
                    n, k, 0.05);
```

See worked example.

There are currently two possible values available for the *method* optional parameter: *clopper_pearson_exact_interval* or *jeffreys_prior_interval*. These constants are both members of class template `binomial_distribution`, so usage is for example:

```
p = binomial_distribution<RealType>::find_lower_bound_on_p(
    n, k, 0.05, binomial_distribution<RealType>::jeffreys_prior_interval);
```

The default method if this parameter is not specified is the Clopper Pearson "exact" interval. This produces an interval that guarantees at least `100(1-alpha)%` coverage, but which is known to be overly conservative, sometimes producing intervals with much greater than the requested coverage.

The alternative calculation method produces a non-informative Jeffreys Prior interval. It produces `100(1-alpha)%` coverage only *in the average case*, though is typically very close to the requested coverage level. It is one of the main methods of calculation recommended in the review by Brown, Cai and DasGupta.

Please note that the "textbook" calculation method using a normal approximation (the Wald interval) is deliberately not provided: it is known to produce consistently poor results, even when the sample size is surprisingly large. Refer to Brown, Cai and DasGupta for a full explanation. Many other methods of calculation are available, and may be more appropriate for specific situations. Unfortunately there appears to be no consensus amongst statisticians as to which is "best": refer to the discussion at the end of Brown, Cai and DasGupta for examples.

The two methods provided here were chosen principally because they can be used for both one and two sided intervals. See also:

Lawrence D. Brown, T. Tony Cai and Anirban DasGupta (2001), Interval Estimation for a Binomial Proportion, Statistical Science, Vol. 16, No. 2, 101-133.

T. Tony Cai (2005), One-sided confidence intervals in discrete distributions, Journal of Statistical Planning and Inference 131, 63-88.

Agresti, A. and Coull, B. A. (1998). Approximate is better than "exact" for interval estimation of binomial proportions. Amer. Statist. 52 119-126.

Clopper, C. J. and Pearson, E. S. (1934). The use of confidence or fiducial limits illustrated in the case of the binomial. Biometrika 26 404-413.

Upper Bound on the Success Fraction

```
static RealType find_upper_bound_on_p(
   RealType trials,
   RealType successes,
   RealType alpha,
   unspecified-type method = clopper_pearson_exact_interval);
```

Returns an upper bound on the success fraction:

trials	The total number of trials conducted.
successes	The number of successes that occurred.
alpha	The largest acceptable probability that the true value of the success fraction is **greater than** the value returned.
method	An optional parameter that specifies the method to be used to compute the interval. Refer to the documentation for `find_upper_bound_on_p` above for the meaning of the method options.

For example, if you observe *k* successes from *n* trials the best estimate for the success fraction is simply *k/n*, but if you want to be 95% sure that the true value is **less than** some value, p_{max}, then:

```
pmax = binomial_distribution<RealType>::find_upper_bound_on_p(
                    n, k, 0.05);
```

See worked example.

Note

In order to obtain a two sided bound on the success fraction, you call both `find_lower_bound_on_p` **and** `find_upper_bound_on_p` each with the same arguments.

If the desired risk level that the true success fraction lies outside the bounds is α, then you pass α/2 to these functions.

So for example a two sided 95% confidence interval would be obtained by passing α = 0.025 to each of the functions.

See worked example.

Estimating the Number of Trials Required for a Certain Number of Successes

```
static RealType find_minimum_number_of_trials(
   RealType k,     // number of events
   RealType p,     // success fraction
   RealType alpha); // probability threshold
```

This function estimates the minimum number of trials required to ensure that more than k events is observed with a level of risk *alpha* that k or fewer events occur.

k	The number of success observed.

p	The probability of success for each trial.
alpha	The maximum acceptable probability that k events or fewer will be observed.

For example:

```
binomial_distribution<RealType>::find_number_of_trials(10, 0.5, 0.05);
```

Returns the smallest number of trials we must conduct to be 95% sure of seeing 10 events that occur with frequency one half.

Estimating the Maximum Number of Trials to Ensure no more than a Certain Number of Successes

```
static RealType find_maximum_number_of_trials(
   RealType k,     // number of events
   RealType p,     // success fraction
   RealType alpha); // probability threshold
```

This function estimates the maximum number of trials we can conduct to ensure that k successes or fewer are observed, with a risk *alpha* that more than k occur.

k	The number of success observed.
p	The probability of success for each trial.
alpha	The maximum acceptable probability that more than k events will be observed.

For example:

```
binomial_distribution<RealType>::find_maximum_number_of_trials(0, 1e-6, 0.05);
```

Returns the largest number of trials we can conduct and still be 95% certain of not observing any events that occur with one in a million frequency. This is typically used in failure analysis.

See Worked Example.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain for the random variable *k* is `0 <= k <= N`, otherwise a domain_error is returned.

It's worth taking a moment to define what these accessors actually mean in the context of this distribution:

Table 15. Meaning of the non-member accessors

Function	Meaning
Probability Density Function	The probability of obtaining **exactly k successes** from n trials with success fraction p. For example: `pdf(binomial(n, p), k)`
Cumulative Distribution Function	The probability of obtaining **k successes or fewer** from n trials with success fraction p. For example: `cdf(binomial(n, p), k)`
Complement of the Cumulative Distribution Function	The probability of obtaining **more than k successes** from n trials with success fraction p. For example: `cdf(complement(binomial(n, p), k))`
Quantile	Given a binomial distribution with *n* trials, success fraction *p* and probability *P*, finds the largest number of successes *k* whose CDF is less than *P*. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function.
Quantile from the complement of the probability	Given a binomial distribution with *n* trials, success fraction *p* and probability *Q*, finds the smallest number of successes *k* whose CDF is greater than *1-Q*. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function.

Examples

Various worked examples are available illustrating the use of the binomial distribution.

Accuracy

This distribution is implemented using the incomplete beta functions ibeta and ibetac, please refer to these functions for information on accuracy.

Implementation

In the following table *p* is the probability that one trial will be successful (the success fraction), *n* is the number of trials, *k* is the number of successes, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Implementation is in terms of ibeta_derivative: if $_nC_k$ is the binomial coefficient of a and b, then we have: $$\begin{aligned} f(k; n, p) &= {}_nC_k p^k (1-p)^{n-k} \\ &= \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} \\ &= \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} p^k (1-p)^{n-k} \\ &= \frac{p^k (1-p)^{n-k}}{B(k+1, n-k+1)(n+1)} \end{aligned}$$ Which can be evaluated as `ibeta_derivative(k+1, n-k+1, p) / (n+1)` The function ibeta_derivative is used here, since it has already been optimised for the lowest possible error - indeed this is really just a thin wrapper around part of the internals of the incomplete beta function. There are also various special cases: refer to the code for details.

cdf	Using the relation:
	
	 `p = I[sub 1-p](n - k, k + 1)`
	 `  = 1 - I[sub p](k + 1, n - k)`
	 `  = ibetac(k + 1, n - k, p)`
	
	 There are also various special cases: refer to the code for details.

cdf complement	Using the relation: q = ibeta(k + 1, n - k, p) There are also various special cases: refer to the code for details.
quantile	Since the cdf is non-linear in variate *k* none of the inverse incomplete beta functions can be used here. Instead the quantile is found numerically using a derivative free method (TOMS 748 algorithm).
quantile from the complement	Found numerically as above.
mean	`p * n`
variance	`p * n * (1-p)`
mode	`floor(p * (n + 1))`
skewness	`(1 - 2 * p) / sqrt(n * p * (1 - p))`
kurtosis	`3 - (6 / n) + (1 / (n * p * (1 - p)))`
kurtosis excess	`(1 - 6 * p * q) / (n * p * q)`

Function	Implementation Notes
parameter estimation	The member functions `find_upper_bound_on_p` `find_lower_bound_on_p` and `find_number_of_trials` are implemented in terms of the inverse incomplete beta functions ibetac_inv, ibeta_inv, and ibetac_invb respectively

References

- Weisstein, Eric W. "Binomial Distribution." From MathWorld--A Wolfram Web Resource.
- Wikipedia binomial distribution.
- NIST Explorary Data Analysis.

Cauchy-Lorentz Distribution

```
#include <boost/math/distributions/cauchy.hpp>
```

```
template <class RealType = double,
          class Policy   = policies::policy<> >
class cauchy_distribution;

typedef cauchy_distribution<> cauchy;

template <class RealType, class Policy>
class cauchy_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   cauchy_distribution(RealType location = 0, RealType scale = 1);

   RealType location()const;
   RealType scale()const;
};
```

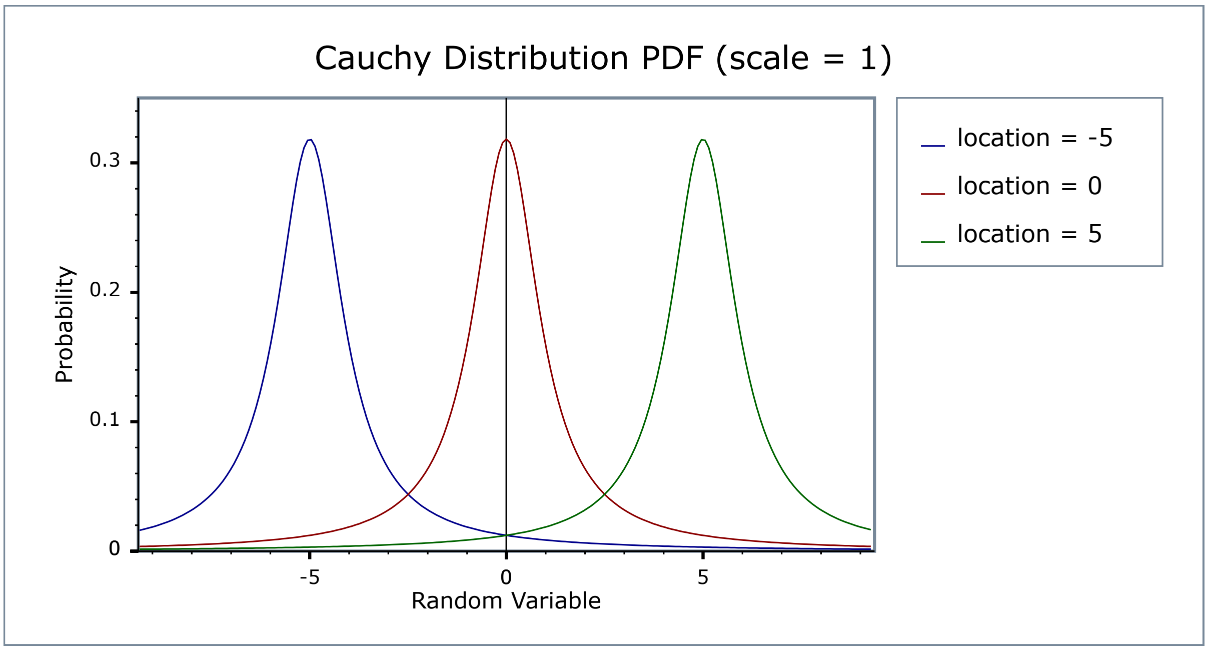
The Cauchy-Lorentz distribution is named after Augustin Cauchy and Hendrik Lorentz. It is a continuous probability distribution with probability distribution function PDF given by:

$$f(x; x_0, \gamma) = \frac{1}{\pi}\left(\frac{\gamma}{(x - x_0)^2 + \gamma^2}\right)$$

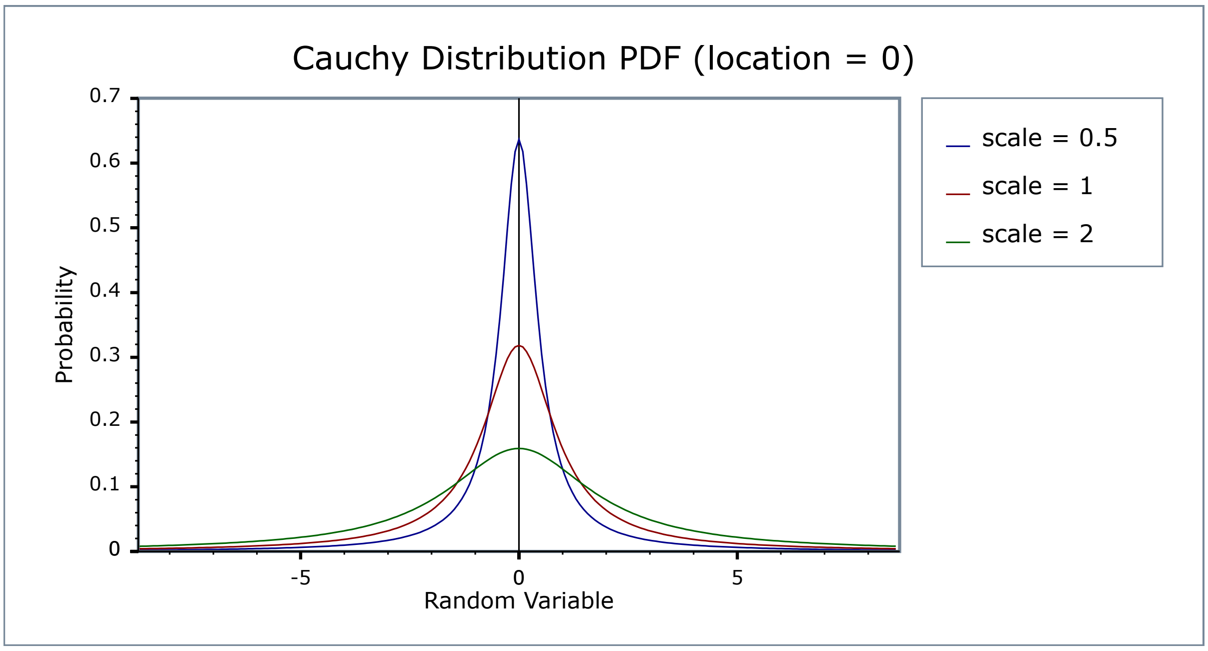
The location parameter x_0 is the location of the peak of the distribution (the mode of the distribution), while the scale parameter γ specifies half the width of the PDF at half the maximum height. If the location is zero, and the scale 1, then the result is a standard Cauchy distribution.

The distribution is important in physics as it is the solution to the differential equation describing forced resonance, while in spectroscopy it is the description of the line shape of spectral lines.

The following graph shows how the distributions moves as the location parameter changes:



While the following graph shows how the shape (scale) parameter alters the distribution:



Member Functions

```
cauchy_distribution(RealType location = 0, RealType scale = 1);
```

Constructs a Cauchy distribution, with location parameter *location* and scale parameter *scale*. When these parameters take their default values (location = 0, scale = 1) then the result is a Standard Cauchy Distribution.

Requires scale > 0, otherwise calls domain_error.

```
RealType location()const;
```

Returns the location parameter of the distribution.

```
RealType scale()const;
```

Returns the scale parameter of the distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

Note however that the Cauchy distribution does not have a mean, standard deviation, etc. See mathematically undefined function to control whether these should fail to compile with a BOOST_STATIC_ASSERTION_FAILURE, which is the default.

Alternately, the functions mean, standard deviation, variance, skewness, kurtosis and kurtosis_excess will all return a domain_error if called.

The domain of the random variable is [-[max_value], +[min_value]].

Accuracy

The Cauchy distribution is implemented in terms of the standard library `tan` and `atan` functions, and as such should have very low error rates.

Implementation

In the following table x_0 is the location parameter of the distribution, γ is its scale parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	Implementation Notes
pdf	Using the relation: pdf = 1 / (π * γ * (1 + ((x - x_0) / γ)^2)
cdf and its complement	The cdf is normally given by: p = 0.5 + atan(x)/π But that suffers from cancellation error as x -> -∞. So recall that for `x < 0`: atan(x) = -π/2 - atan(1/x) Substituting into the above we get: p = -atan(1/x) ; x < 0 So the procedure is to calculate the cdf for -fabs(x) using the above formula. Note that to factor in the location and scale parameters you must substitute (x - x_0) / γ for x in the above. This procedure yields the smaller of *p* and *q*, so the result may need subtracting from 1 depending on whether we want the complement or not, and whether *x* is less than x_0 or not.
quantile	The same procedure is used irrespective of whether we're starting from the probability or its complement. First the argument *p* is reduced to the range [-0.5, 0.5], then the relation x = x_0 ± γ / tan(π * p) is used to obtain the result. Whether we're adding or subtracting from x_0 is determined by whether we're starting from the complement or not.
mode	The location parameter.

References

- Cauchy-Lorentz distribution
- NIST Exploratory Data Analysis
- Weisstein, Eric W. "Cauchy Distribution." From MathWorld--A Wolfram Web Resource.

Chi Squared Distribution

```
#include <boost/math/distributions/chi_squared.hpp>
```

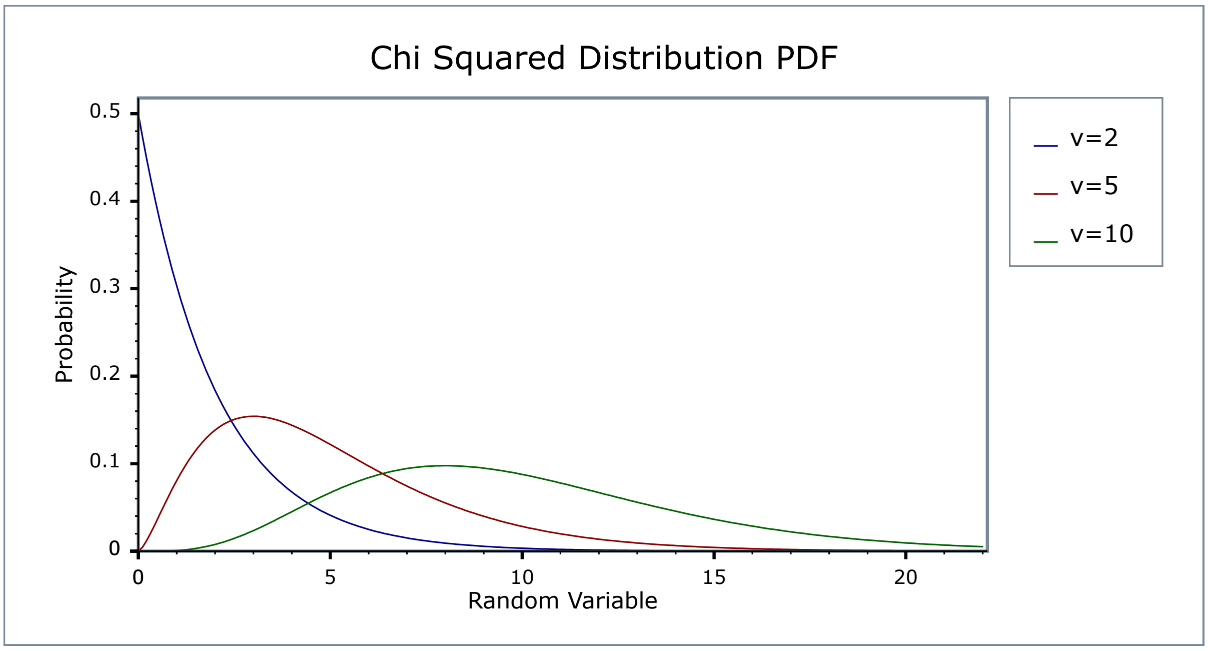
```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class chi_squared_distribution;

typedef chi_squared_distribution<> chi_squared;

template <class RealType, class Policy>
class chi_squared_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor:
   chi_squared_distribution(RealType i);

   // Accessor to parameter:
   RealType degrees_of_freedom()const;

   // Parameter estimation:
   static RealType find_degrees_of_freedom(
      RealType difference_from_mean,
      RealType alpha,
      RealType beta,
      RealType sd,
      RealType hint = 100);
};

}} // namespaces
```

The Chi-Squared distribution is one of the most widely used distributions in statistical tests. If χ_i are ν independent, normally distributed random variables with means μ_i and variances σ_i^2, then the random variable:

$$Q = \sum_{i=1}^{\nu} \left(\frac{\chi_i - \nu_i}{\sigma_i} \right)^2$$

is distributed according to the Chi-Squared distribution.

The Chi-Squared distribution is a special case of the gamma distribution and has a single parameter ν that specifies the number of degrees of freedom. The following graph illustrates how the distribution changes for different values of ν:



Member Functions

```
chi_squared_distribution(RealType v);
```

Constructs a Chi-Squared distribution with *v* degrees of freedom.

Requires v > 0, otherwise calls domain_error.

```
RealType degrees_of_freedom()const;
```

Returns the parameter *v* from which this object was constructed.

```
static RealType find_degrees_of_freedom(
   RealType difference_from_variance,
   RealType alpha,
   RealType beta,
   RealType variance,
   RealType hint = 100);
```

Estimates the sample size required to detect a difference from a nominal variance in a Chi-Squared test for equal standard deviations.

difference_from_variance	The difference from the assumed nominal variance that is to be detected: Note that the sign of this value is critical, see below.
alpha	The maximum acceptable risk of rejecting the null hypothesis when it is in fact true.
beta	The maximum acceptable risk of falsely failing to reject the null hypothesis.
variance	The nominal variance being tested against.
hint	An optional hint on where to start looking for a result: the current sample size would be a good choice.

Note that this calculation works with *variances* and not *standard deviations*.

The sign of the parameter *difference_from_variance* is important: the Chi Squared distribution is asymmetric, and the caller must decide in advance whether they are testing for a variance greater than a nominal value (positive *difference_from_variance*) or testing for a variance less than a nominal value (negative *difference_from_variance*). If the latter, then obviously it is a requirement that `variance + difference_from_variance > 0`, since no sample can have a negative variance!

This procedure uses the method in Diamond, W. J. (1989). Practical Experiment Designs, Van-Nostrand Reinhold, New York.

See also section on Sample sizes required in the NIST Engineering Statistics Handbook, Section 7.2.3.2.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

(We have followed the usual restriction of the mode to degrees of freedom >= 2, but note that the maximum of the pdf is actually zero for degrees of freedom from 2 down to 0, and provide an extended definition that would avoid a discontinuity in the mode as alternative code in a comment).

The domain of the random variable is [0, +∞].

Examples

Various worked examples are available illustrating the use of the Chi Squared Distribution.

Accuracy

The Chi-Squared distribution is implemented in terms of the incomplete gamma functions: please refer to the accuracy data for those functions.

Implementation

In the following table *v* is the number of degrees of freedom of the distribution, *x* is the random variate, *p* is the probability, and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = gamma_p_derivative(v / 2, x / 2) / 2
cdf	Using the relation: p = gamma_p(v / 2, x / 2)
cdf complement	Using the relation: q = gamma_q(v / 2, x / 2)
quantile	Using the relation: x = 2 * gamma_p_inv(v / 2, p)
quantile from the complement	Using the relation: x = 2 * gamma_q_inv(v / 2, p)
mean	v
variance	2v
mode	v - 2 (if v >= 2)
skewness	2 * sqrt(2 / v) == sqrt(8 / v)
kurtosis	3 + 12 / v
kurtosis excess	12 / v

References

- NIST Exploratory Data Analysis
- Chi-square distribution
- Weisstein, Eric W. "Chi-Squared Distribution." From MathWorld--A Wolfram Web Resource.

Exponential Distribution

```
#include <boost/math/distributions/exponential.hpp>
```

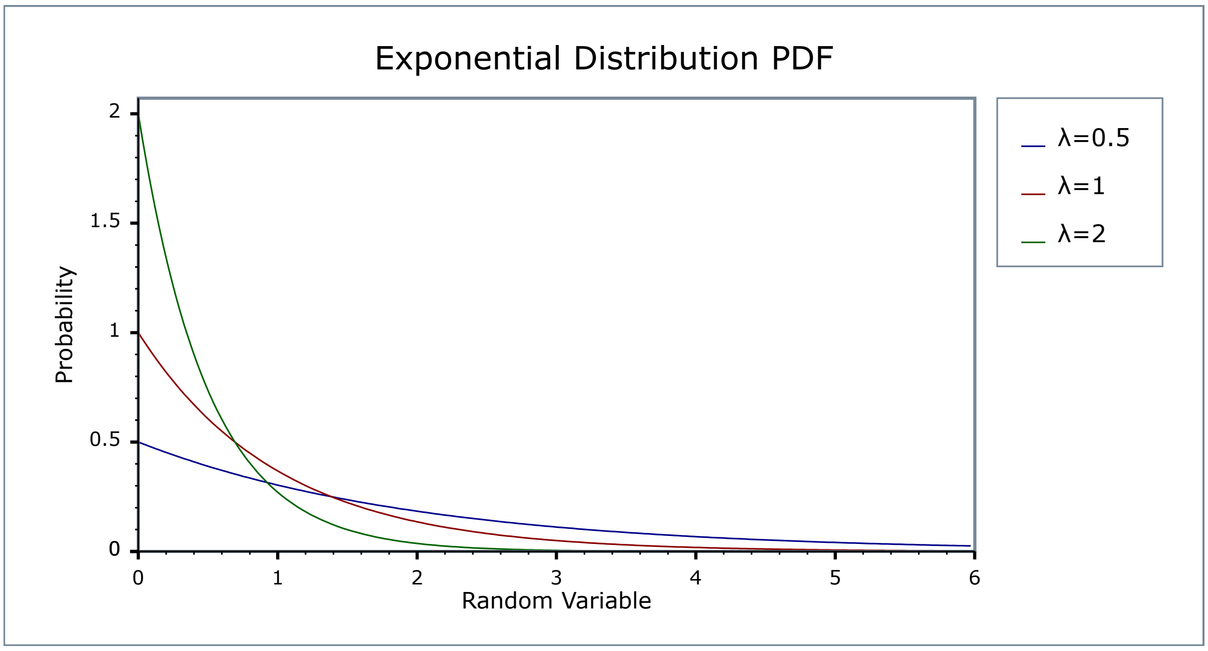
```
template <class RealType = double,
          class Policy   = policies::policy<> >
class exponential_distribution;

typedef exponential_distribution<> exponential;

template <class RealType, class Policy>
class exponential_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   exponential_distribution(RealType lambda = 1);

   RealType lambda()const;
};
```

The exponential distribution is a continuous probability distribution with PDF:

$$f(x) = \lambda e^{-\lambda x}$$

It is often used to model the time between independent events that happen at a constant average rate.

The following graph shows how the distribution changes for different values of the rate parameter lambda:



Member Functions

```
exponential_distribution(RealType lambda = 1);
```

Constructs an Exponential distribution with parameter *lambda*. Lambda is defined as the reciprocal of the scale parameter.

Requires lambda > 0, otherwise calls domain_error.

```
RealType lambda()const;
```

Accessor function returns the lambda parameter of the distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, +∞].

Accuracy

The exponential distribution is implemented in terms of the standard library functions `exp`, `log`, `log1p` and `expm1` and as such should have very low error rates.

Implementation

In the following table λ is the parameter lambda of the distribution, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = λ * exp(-λ * x)
cdf	Using the relation: p = 1 - exp(-x * λ) = -expm1(-x * λ)
cdf complement	Using the relation: q = exp(-x * λ)
quantile	Using the relation: x = -log(1-p) / λ = -log1p(-p) / λ
quantile from the complement	Using the relation: x = -log(q) / λ
mean	1/λ
standard deviation	1/λ
mode	0
skewness	2
kurtosis	9
kurtosis excess	6

references

- Weisstein, Eric W. "Exponential Distribution." From MathWorld--A Wolfram Web Resource

- Wolfram Mathematica calculator
- NIST Exploratory Data Analysis
- Wikipedia Exponential distribution

(See also the reference documentation for the related Extreme Distributions.)

- Extreme Value Distributions, Theory and Applications Samuel Kotz & Saralees Nadarajah discuss the relationship of the types of extreme value distributions.

Extreme Value Distribution

```
#include <boost/math/distributions/extreme.hpp>
```

```
template <class RealType = double,
          class Policy   = policies::policy<> >
class extreme_value_distribution;

typedef extreme_value_distribution<> extreme_value;

template <class RealType, class Policy>
class extreme_value_distribution
{
public:
   typedef RealType value_type;

   extreme_value_distribution(RealType location = 0, RealType scale = 1);

   RealType scale()const;
   RealType location()const;
};
```

There are various extreme value distributions : this implementation represents the maximum case, and is variously known as a Fisher-Tippett distribution, a log-Weibull distribution or a Gumbel distribution.

Extreme value theory is important for assessing risk for highly unusual events, such as 100-year floods.

More information can be found on the NIST, Wikipedia, Mathworld, and Extreme value theory websites.

The relationship of the types of extreme value distributions, of which this is but one, is discussed by Extreme Value Distributions, Theory and Applications Samuel Kotz & Saralees Nadarajah.

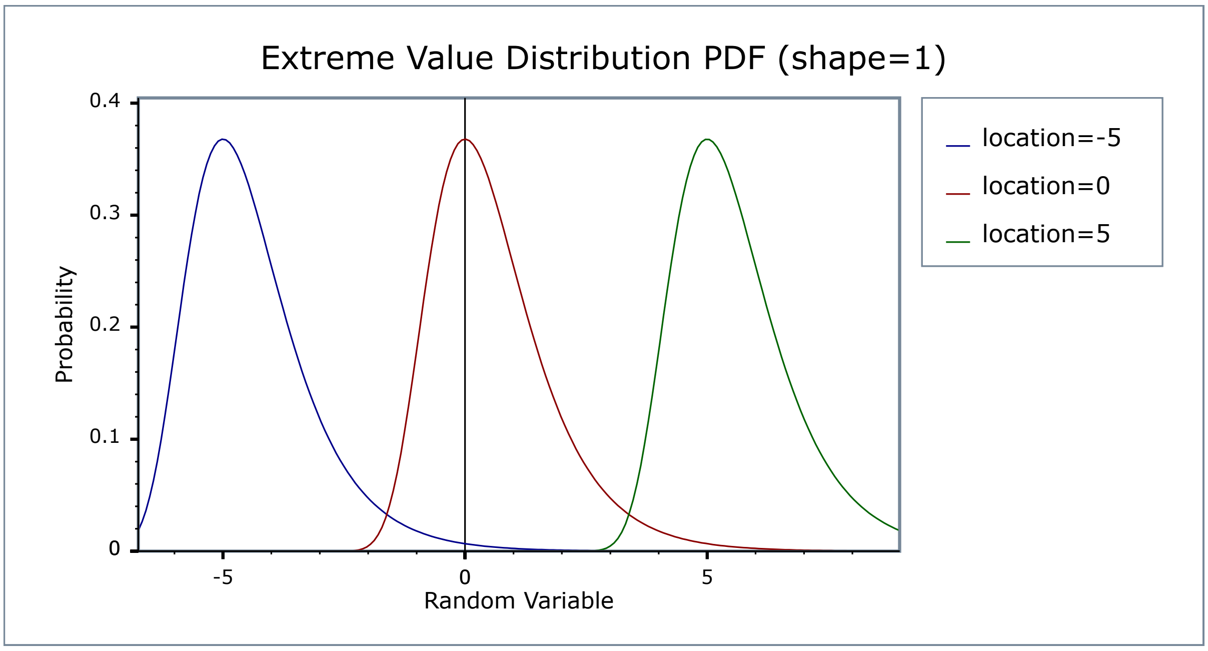
The distribution has a PDF given by:

$f(x) = (1/scale)\ e^{-(x-location)/scale}\ e^{-e^{-(x-location)/scale}}$

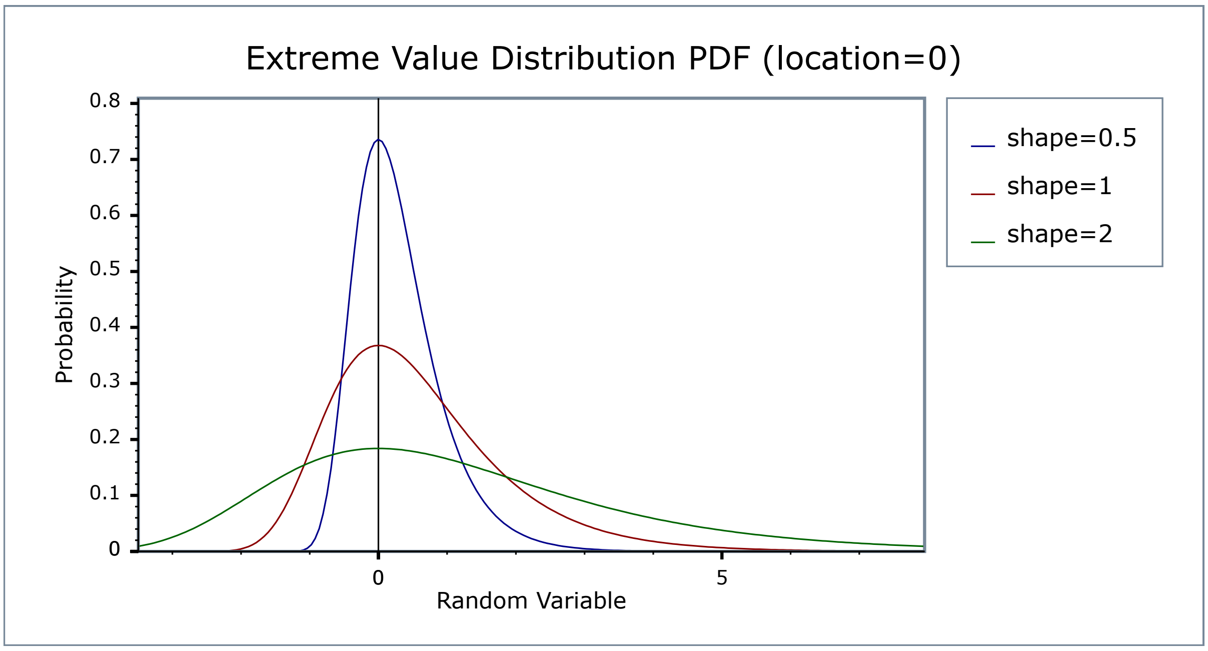
Which in the standard case (scale = 1, location = 0) reduces to:

$f(x) = e^{-x}e^{-e^{-x}}$

The following graph illustrates how the PDF varies with the location parameter:



And this graph illustrates how the PDF varies with the shape parameter:



Member Functions

```
extreme_value_distribution(RealType location = 0, RealType scale = 1);
```

Constructs an Extreme Value distribution with the specified location and scale parameters.

Requires `scale > 0`, otherwise calls domain_error.

```
RealType location()const;
```

Returns the location parameter of the distribution.

```
RealType scale()const;
```

Returns the scale parameter of the distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random parameter is [-∞, +∞].

Accuracy

The extreme value distribution is implemented in terms of the standard library `exp` and `log` functions and as such should have very low error rates.

Implementation

In the following table: *a* is the location parameter, *b* is the scale parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	Implementation Notes
pdf	Using the relation: pdf = exp((a-x)/b) * exp(-exp((a-x)/b)) / b
cdf	Using the relation: p = exp(-exp((a-x)/b))
cdf complement	Using the relation: q = -expm1(-exp((a-x)/b))
quantile	Using the relation: a - log(-log(p)) * b
quantile from the complement	Using the relation: a - log(-log1p(-q)) * b
mean	a + Euler-Mascheroni-constant * b
standard deviation	pi * b / sqrt(6)
mode	The same as the location parameter *a*.
skewness	12 * sqrt(6) * zeta(3) / pi^3
kurtosis	27 / 5
kurtosis excess	kurtosis - 3 or 12 / 5

F Distribution

```
#include <boost/math/distributions/fisher_f.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class fisher_f_distribution;

typedef fisher_f_distribution<> fisher_f;

template <class RealType, class Policy>
class fisher_f_distribution
{
public:
   typedef RealType value_type;

   // Construct:
   fisher_f_distribution(const RealType& i, const RealType& j);

   // Accessors:
   RealType degrees_of_freedom1()const;
   RealType degrees_of_freedom2()const;
};

}} //namespaces
```

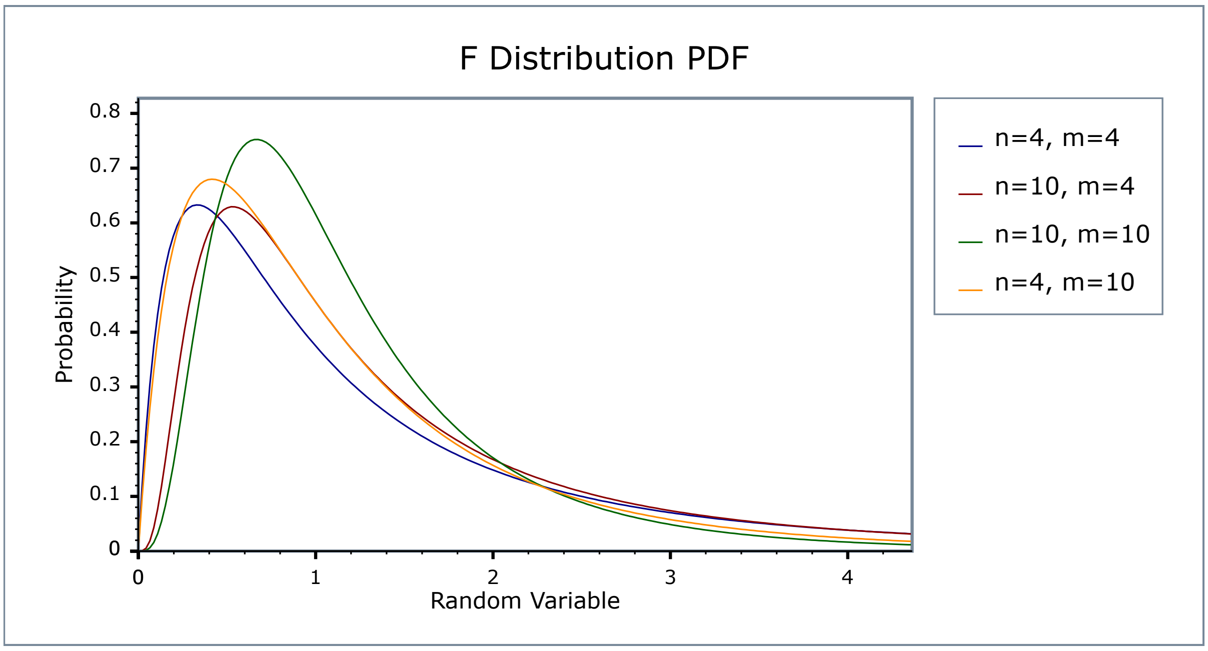
The F distribution is a continuous distribution that arises when testing whether two samples have the same variance. If χ^2_m and χ^2_n are independent variates each distributed as Chi-Squared with *m* and *n* degrees of freedom, then the test statistic:

$F_{n,m} = (\chi^2_n / n) / (\chi^2_m / m)$

Is distributed over the range [0, ∞] with an F distribution, and has the PDF:

$$f_{n,m}(x) = \frac{m^{\frac{m}{2}} n^{\frac{n}{2}} x^{\frac{n}{2}-1}}{(m+nx)^{\frac{(n+m)}{2}} B\left(\frac{n}{2}, \frac{m}{2}\right)}$$

The following graph illustrates how the PDF varies depending on the two degrees of freedom parameters.



Member Functions

```
fisher_f_distribution(const RealType& df1, const RealType& df2);
```

Constructs an F-distribution with numerator degrees of freedom *df1* and denominator degrees of freedom *df2*.

Requires that *df1* and *df2* are both greater than zero, otherwise domain_error is called.

```
RealType degrees_of_freedom1()const;
```

Returns the numerator degrees of freedom parameter of the distribution.

```
RealType degrees_of_freedom2()const;
```

Returns the denominator degrees of freedom parameter of the distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, +∞].

Examples

Various worked examples are available illustrating the use of the F Distribution.

Accuracy

The normal distribution is implemented in terms of the incomplete beta function and its inverses, refer to those functions for accuracy data.

Implementation

In the following table *v1* and *v2* are the first and second degrees of freedom parameters of the distribution, *x* is the random variate, *p* is the probability, and *q = 1-p*.

Function	Implementation Notes
pdf	The usual form of the PDF is given by: $$f_{n,m}(x) = \frac{m^{\frac{m}{2}} n^{\frac{n}{2}} x^{\frac{n}{2}-1}}{(m+nx)^{\frac{(n+m)}{2}} B\left(\frac{n}{2}, \frac{m}{2}\right)}$$ However, that form is hard to evaluate directly without incurring problems with either accuracy or numeric overflow. Direct differentiation of the CDF expressed in terms of the incomplete beta function led to the following two formulas: $f_{v1,v2}(x)$ = y * ibeta_derivative(v2 / 2, v1 / 2, v2 / (v2 + v1 * x)) with y = (v2 * v1) / ((v2 + v1 * x) * (v2 + v1 * x)) and $f_{v1,v2}(x)$ = y * ibeta_derivative(v1 / 2, v2 / 2, v1 * x / (v2 + v1 * x)) with y = (z * v1 - x * v1 * v1) / z^2 and z = v2 + v1 * x The first of these is used for v1 * x > v2, otherwise the second is used. The aim is to keep the *x* argument to ibeta_derivative away from 1 to avoid rounding error.
cdf	Using the relations: p = ibeta(v1 / 2, v2 / 2, v1 * x / (v2 + v1 * x)) and p = ibetac(v2 / 2, v1 / 2, v2 / (v2 + v1 * x)) The first is used for v1 * x > v2, otherwise the second is used. The aim is to keep the *x* argument to ibeta well away from 1 to avoid rounding error.

Function	**Implementation Notes**
cdf complement	Using the relations: p = ibetac(v1 / 2, v2 / 2, v1 * x / (v2 + v1 * x)) and p = ibeta(v2 / 2, v1 / 2, v2 / (v2 + v1 * x)) The first is used for v1 * x < v2, otherwise the second is used. The aim is to keep the *x* argument to ibeta well away from 1 to avoid rounding error.
quantile	Using the relation: x = v2 * a / (v1 * b) where: a = ibeta_inv(v1 / 2, v2 / 2, p) and b = 1 - a Quantities *a* and *b* are both computed by ibeta_inv without the subtraction implied above.
quantile from the complement	Using the relation: x = v2 * a / (v1 * b) where a = ibetac_inv(v1 / 2, v2 / 2, p) and b = 1 - a Quantities *a* and *b* are both computed by ibetac_inv without the subtraction implied above.
mean	v2 / (v2 - 2)
variance	2 * v2^2 * (v1 + v2 - 2) / (v1 * (v2 - 2) * (v2 - 2) * (v2 - 4))
mode	v2 * (v1 - 2) / (v1 * (v2 + 2))
skewness	2 * (v2 + 2 * v1 - 2) * sqrt((2 * v2 - 8) / (v1 * (v2 + v1 - 2))) / (v2 - 6)
kurtosis and kurtosis excess	Refer to, Weisstein, Eric W. "F-Distribution." From MathWorld--A Wolfram Web Resource.

Gamma (and Erlang) Distribution

```
#include <boost/math/distributions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class gamma_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   gamma_distribution(RealType shape, RealType scale = 1)

   RealType shape()const;
   RealType scale()const;
};

}} // namespaces
```

The gamma distribution is a continuous probability distribution. When the shape parameter is an integer then it is known as the Erlang Distribution. It is also closely related to the Poisson and Chi Squared Distributions.

When the shape parameter has an integer value, the distribution is the Erlang distribution. Since this can be produced by ensuring that the shape parameter has an integer value > 0, the Erlang distribution is not separately implemented.

Note

To avoid potential confusion with the gamma functions, this distribution does not provide the typedef:

```
typedef gamma_distribution<double> gamma;
```

Instead if you want a double precision gamma distribution you can write

```
boost::math::gamma_distribution<> my_gamma(1, 1);
```

For shape parameter *k* and scale parameter θ it is defined by the probability density function:

$$f(x; k, \theta) = x^{k-1}\frac{e^{-\frac{x}{\theta}}}{\theta^k \Gamma(k)}$$

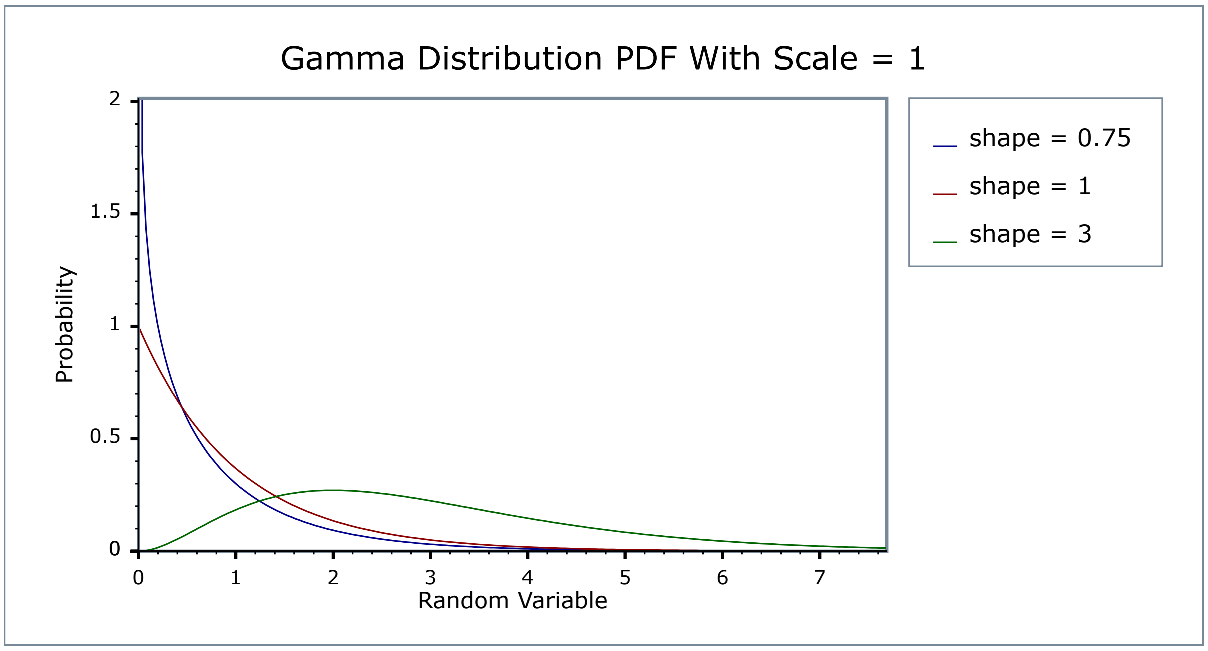
Sometimes an alternative formulation is used: given parameters α = k and β = 1 / θ, then the distribution can be defined by the PDF:

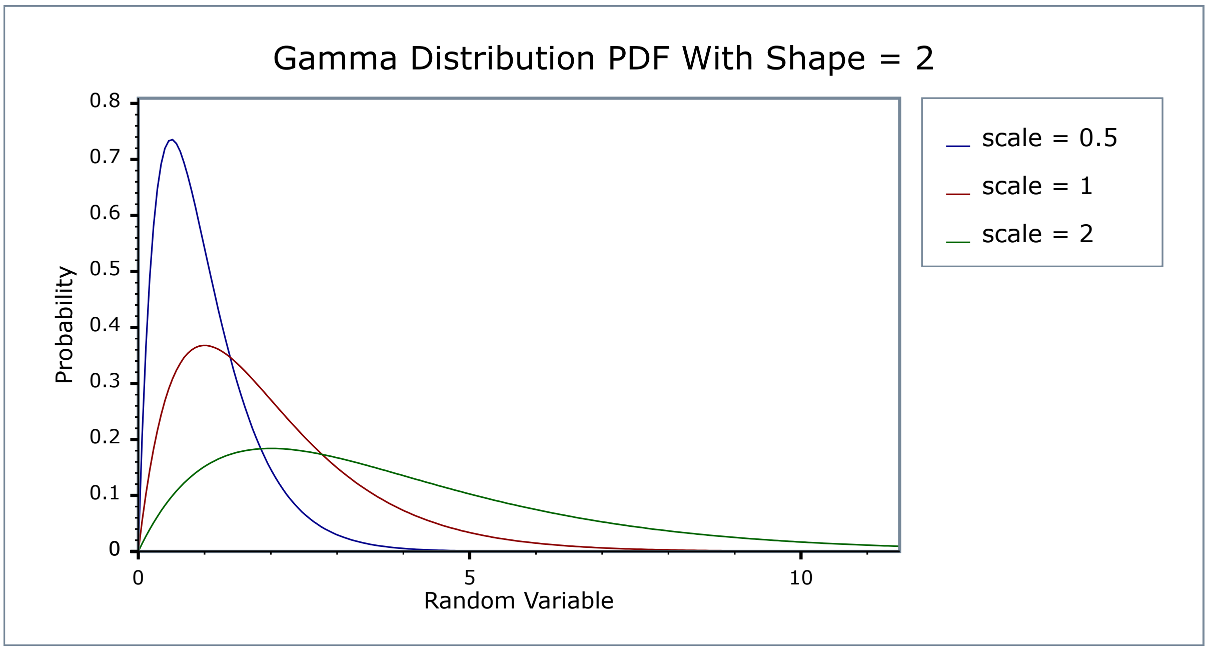
$$f(x; \alpha, \beta) = x^{\alpha-1}\frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

In this form the inverse scale parameter is called a *rate parameter*.

Both forms are in common usage: this library uses the first definition throughout. Therefore to construct a Gamma Distribution from a *rate parameter*, you should pass the reciprocal of the rate as the scale parameter.

The following two graphs illustrate how the PDF of the gamma distribution varies as the parameters vary:





The **Erlang Distribution** is the same as the Gamma, but with the shape parameter an integer. It is often expressed using a *rate* rather than a *scale* as the second parameter (remember that the rate is the reciprocal of the scale).

Internally the functions used to implement the Gamma Distribution are already optimised for small-integer arguments, so in general there should be no great loss of performance from using a Gamma Distribution rather than a dedicated Erlang Distribution.

Member Functions

```
gamma_distribution(RealType shape, RealType scale = 1);
```

Constructs a gamma distribution with shape *shape* and scale *scale*.

Requires that the shape and scale parameters are greater than zero, otherwise calls domain_error.

```
RealType shape()const;
```

Returns the *shape* parameter of this distribution.

```
RealType scale()const;
```

Returns the *scale* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0,+∞].

Accuracy

The gamma distribution is implemented in terms of the incomplete gamma functions gamma_p and gamma_q and their inverses gamma_p_inv and gamma_q_inv: refer to the accuracy data for those functions for more information.

Implementation

In the following table *k* is the shape parameter of the distribution, θ is its scale parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	Implementation Notes
pdf	Using the relation: pdf = gamma_p_derivative(k, x / θ) / θ
cdf	Using the relation: p = gamma_p(k, x / θ)
cdf complement	Using the relation: q = gamma_q(k, x / θ)
quantile	Using the relation: x = θ * gamma_p_inv(k, p)
quantile from the complement	Using the relation: x = θ * gamma_q_inv(k, p)
mean	kθ
variance	$k\theta^2$
mode	(k-1)θ for *k>1* otherwise a domain_error
skewness	2 / sqrt(k)
kurtosis	3 + 6 / k
kurtosis excess	6 / k

Geometric Distribution

```
#include <boost/math/distributions/geometric.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class geometric_distribution;

typedef geometric_distribution<> geometric;

template <class RealType, class Policy>
class geometric_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Constructor from success_fraction:
   geometric_distribution(RealType p);

   // Parameter accessors:
   RealType success_fraction() const;
   RealType successes() const;

   // Bounds on success fraction:
   static RealType find_lower_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability); // alpha
   static RealType find_upper_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability); // alpha

   // Estimate min/max number of trials:
   static RealType find_minimum_number_of_trials(
      RealType k,      // Number of failures.
      RealType p,      // Success fraction.
      RealType probability); // Probability threshold alpha.
   static RealType find_maximum_number_of_trials(
      RealType k,      // Number of failures.
      RealType p,      // Success fraction.
      RealType probability); // Probability threshold alpha.
};

}} // namespaces
```

The class type `geometric_distribution` represents a geometric distribution: it is used when there are exactly two mutually exclusive outcomes of a Bernoulli trial: these outcomes are labelled "success" and "failure".

For Bernoulli trials each with success fraction *p*, the geometric distribution gives the probability of observing *k* trials (failures, events, occurrences, or arrivals) before the first success.

Note

For this implementation, the set of trials **includes zero** (unlike another definition where the set of trials starts at one, sometimes named *shifted*).

The geometric distribution assumes that success_fraction *p* is fixed for all *k* trials.

The probability that there are *k* failures before the first success is

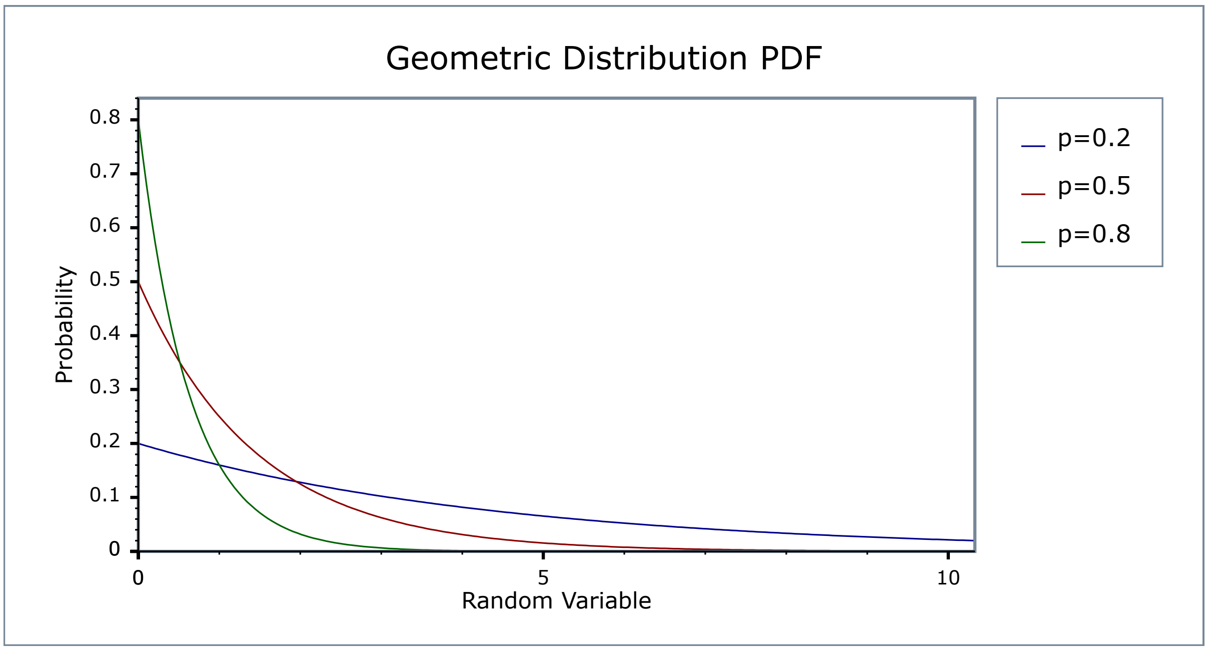
$$\Pr(Y=k) = (1-p)^k p$$

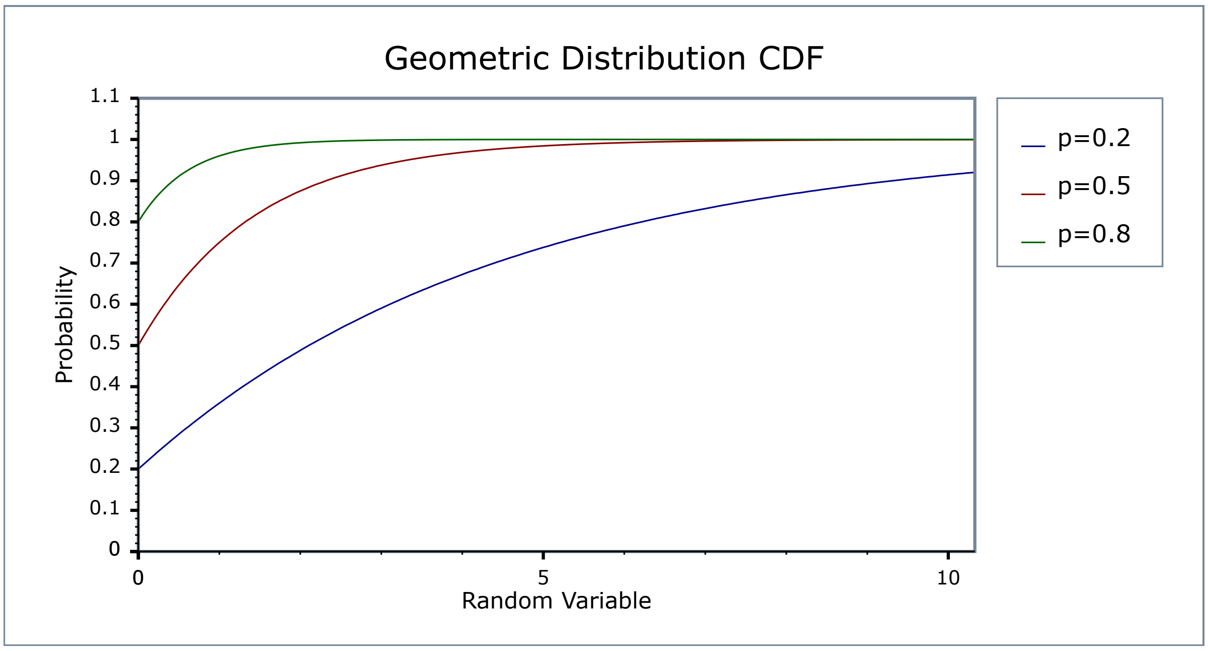
For example, when throwing a 6-face dice the success probability $p = 1/6 = 0.1666$. Throwing repeatedly until a *three* appears, the probability distribution of the number of times *not-a-three* is thrown is geometric.

Geometric distribution has the Probability Density Function PDF:

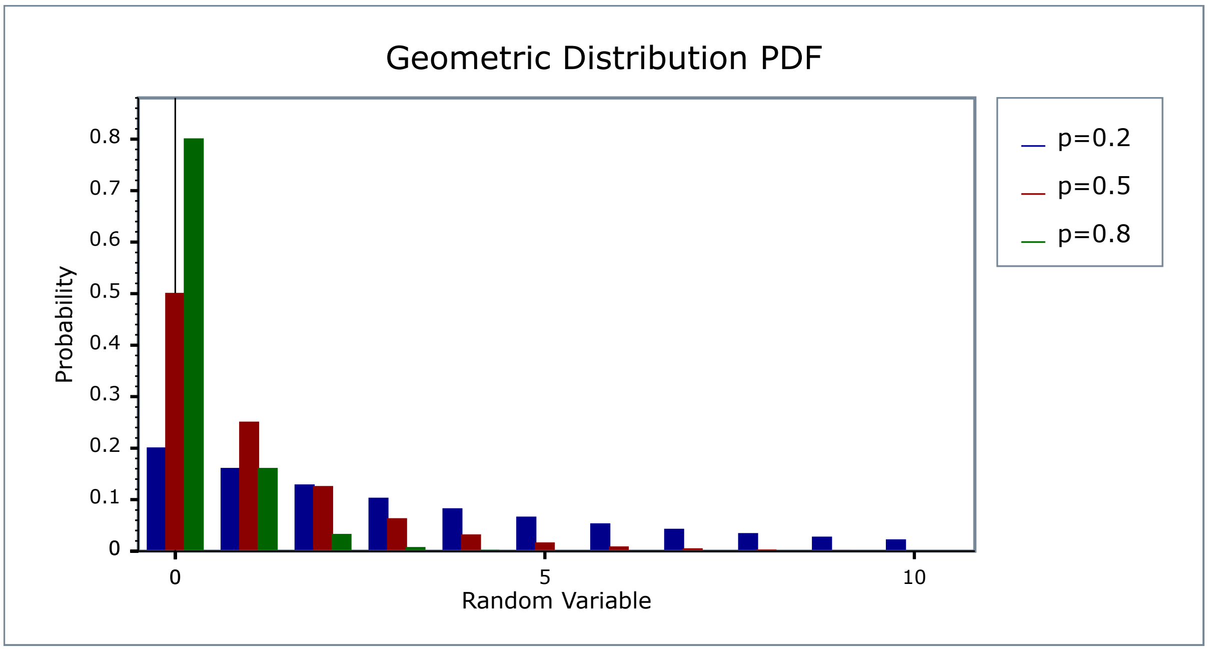
$(1-p)^k p$

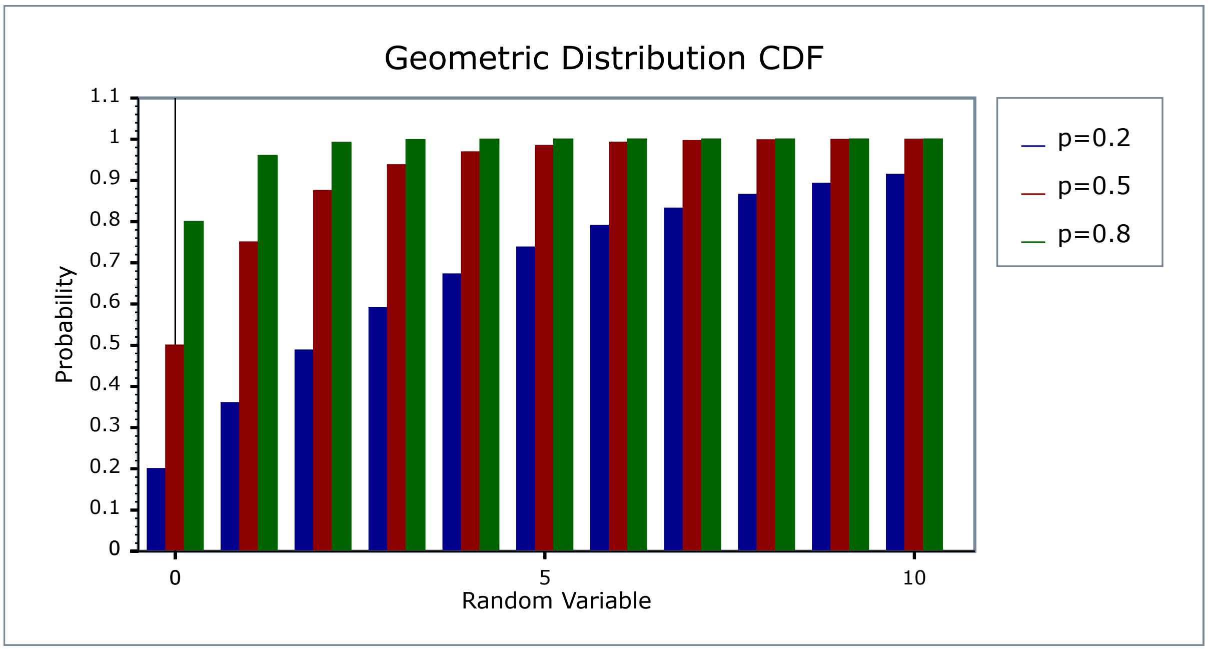
The following graph illustrates how the PDF and CDF vary for three examples of the success fraction p, (when considering the geometric distribution as a continuous function),





and as discrete.





Related Distributions

The geometric distribution is a special case of the Negative Binomial Distribution with successes parameter *r* = 1, so only one first and only success is required : thus by definition `geometric(p) == negative_binomial(1, p)`

```
negative_binomial_distribution(RealType r, RealType success_fraction);
negative_binomial nb(1, success_fraction);
geometric g(success_fraction);
ASSERT(pdf(nb, 1) == pdf(g, 1));
```

This implementation uses real numbers for the computation throughout (because it uses the **real-valued** power and exponential functions). So to obtain a conventional strictly-discrete geometric distribution you must ensure that an integer value is provided for the number of trials (random variable) *k*, and take integer values (floor or ceil functions) from functions that return a number of successes.

Caution

The geometric distribution is a discrete distribution: internally, functions like the `cdf` and `pdf` are treated "as if" they are continuous functions, but in reality the results returned from these functions only have meaning if an integer value is provided for the random variate argument.

The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on the geometric distribution. The reference docs describe how to change the rounding policy for these distributions.

Member Functions

Constructor

```
geometric_distribution(RealType p);
```

Constructor: *p* or success_fraction is the probability of success of a single trial.

Requires: `0 <= p <= 1`.

Accessors

```
RealType success_fraction() const; // successes / trials (0 <= p <= 1)
```

Returns the success_fraction parameter *p* from which this distribution was constructed.

```
RealType successes() const; // required successes always one,
// included for compatibility with negative binomial distribution
// with successes r == 1.
```

Returns unity.

The following functions are equivalent to those provided for the negative binomial, with successes = 1, but are provided here for completeness.

The best method of calculation for the following functions is disputed: see Binomial Distribution and Negative Binomial Distribution for more discussion.

Lower Bound on success_fraction Parameter *p*

```
static RealType find_lower_bound_on_p(
  RealType failures,
  RealType probability) // (0 <= alpha <= 1), 0.05 equivalent to 95% confidence.
```

Returns a **lower bound** on the success fraction:

failures	The total number of failures before the 1st success.
alpha	The largest acceptable probability that the true value of the success fraction is **less than** the value returned.

For example, if you observe *k* failures from *n* trials the best estimate for the success fraction is simply 1/*n*, but if you want to be 95% sure that the true value is **greater than** some value, p_{min}, then:

```
pmin = geometric_distribution<RealType>::
   find_lower_bound_on_p(failures, 0.05);
```

See negative_binomial confidence interval example.

This function uses the Clopper-Pearson method of computing the lower bound on the success fraction, whilst many texts refer to this method as giving an "exact" result in practice it produces an interval that guarantees *at least* the coverage required, and may produce pessimistic estimates for some combinations of *failures* and *successes*. See:

Yong Cai and K. Krishnamoorthy, A Simple Improved Inferential Method for Some Discrete Distributions. Computational statistics and data analysis, 2005, vol. 48, no3, 605-621.

Upper Bound on success_fraction Parameter p

```
static RealType find_upper_bound_on_p(
   RealType trials,
   RealType alpha); // (0 <= alpha <= 1), 0.05 equivalent to 95% confidence.
```

Returns an **upper bound** on the success fraction:

trials	The total number of trials conducted.
alpha	The largest acceptable probability that the true value of the success fraction is **greater than** the value returned.

For example, if you observe *k* successes from *n* trials the best estimate for the success fraction is simply *k/n*, but if you want to be 95% sure that the true value is **less than** some value, p_{max}, then:

```
pmax = geometric_distribution<RealType>::find_upper_bound_on_p(
                    k, 0.05);
```

See negative binomial confidence interval example.

This function uses the Clopper-Pearson method of computing the lower bound on the success fraction, whilst many texts refer to this method as giving an "exact" result in practice it produces an interval that guarantees *at least* the coverage required, and may produce pessimistic estimates for some combinations of *failures* and *successes*. See:

Yong Cai and K. Krishnamoorthy, A Simple Improved Inferential Method for Some Discrete Distributions. Computational statistics and data analysis, 2005, vol. 48, no3, 605-621.

Estimating Number of Trials to Ensure at Least a Certain Number of Failures

```
static RealType find_minimum_number_of_trials(
   RealType k,     // number of failures.
   RealType p,     // success fraction.
   RealType alpha); // probability threshold (0.05 equivalent to 95%).
```

This functions estimates the number of trials required to achieve a certain probability that **more than k failures will be observed**.

k	The target number of failures to be observed.
p	The probability of *success* for each trial.
alpha	The maximum acceptable *risk* that only *k* failures or fewer will be observed.

For example:

```
geometric_distribution<RealType>::find_minimum_number_of_trials(10, 0.5, 0.05);
```

Returns the smallest number of trials we must conduct to be 95% (1-0.05) sure of seeing 10 failures that occur with frequency one half.

Worked Example.

This function uses numeric inversion of the geometric distribution to obtain the result: another interpretation of the result is that it finds the number of trials (failures) that will lead to an *alpha* probability of observing *k* failures or fewer.

Estimating Number of Trials to Ensure a Maximum Number of Failures or Less

```
static RealType find_maximum_number_of_trials(
   RealType k,     // number of failures.
   RealType p,     // success fraction.
   RealType alpha); // probability threshold (0.05 equivalent to 95%).
```

This functions estimates the maximum number of trials we can conduct and achieve a certain probability that **k failures or fewer will be observed**.

k	The maximum number of failures to be observed.
p	The probability of *success* for each trial.
alpha	The maximum acceptable *risk* that more than *k* failures will be observed.

For example:

```
geometric_distribution<RealType>::find_maximum_number_of_trials(0, 1.0-1.0/1000000, 0.05);
```

Returns the largest number of trials we can conduct and still be 95% sure of seeing no failures that occur with frequency one in one million.

This function uses numeric inversion of the geometric distribution to obtain the result: another interpretation of the result, is that it finds the number of trials that will lead to an *alpha* probability of observing more than k failures.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

However it's worth taking a moment to define what these actually mean in the context of this distribution:

Table 16. Meaning of the non-member accessors.

Function	Meaning
Probability Density Function	The probability of obtaining **exactly k failures** from *k* trials with success fraction p. For example: `pdf(geometric(p), k)`
Cumulative Distribution Function	The probability of obtaining **k failures or fewer** from *k* trials with success fraction p and success on the last trial. For example: `cdf(geometric(p), k)`
Complement of the Cumulative Distribution Function	The probability of obtaining **more than k failures** from *k* trials with success fraction p and success on the last trial. For example: `cdf(complement(geometric(p), k))`
Quantile	The **greatest** number of failures *k* expected to be observed from *k* trials with success fraction *p*, at probability *P*. Note that the value returned is a real-number, and not an integer. Depending on the use case you may want to take either the floor or ceiling of the real result. For example: `quantile(geometric(p), P)`
Quantile from the complement of the probability	The **smallest** number of failures *k* expected to be observed from *k* trials with success fraction *p*, at probability *P*. Note that the value returned is a real-number, and not an integer. Depending on the use case you may want to take either the floor or ceiling of the real result. For example: `quantile(complement(geometric(p), P))`

Accuracy

This distribution is implemented using the pow and exp functions, so most results are accurate within a few epsilon for the RealType. For extreme values of `double` *p*, for example 0.9999999999, accuracy can fall significantly, for example to 10 decimal digits (from 16).

Implementation

In the following table, *p* is the probability that any one trial will be successful (the success fraction), *k* is the number of failures, *p* is the probability and *q = 1-p*, *x* is the given probability to estimate the expected number of failures using the quantile.

Function	Implementation Notes
pdf	pdf = p * pow(q, k)
cdf	cdf = 1 - $q^{k=1}$
cdf complement	exp(log1p(-p) * (k+1))
quantile	k = log1p(-x) / log1p(-p) -1
quantile from the complement	k = log(x) / log1p(-p) -1
mean	(1-p)/p
variance	(1-p)/p^2
mode	0
skewness	(2-p)/√q
kurtosis	9+p^2/q
kurtosis excess	6 +p^2/q
parameter estimation member functions	See Negative Binomial Distribution
`find_lower_bound_on_p`	See Negative Binomial Distribution
`find_upper_bound_on_p`	See Negative Binomial Distribution
`find_minimum_number_of_trials`	See Negative Binomial Distribution
`find_maximum_number_of_trials`	See Negative Binomial Distribution

Hyperexponential Distribution

```
#include <boost/math/distributions/hyperexponential.hpp>
```

```
namespace boost{ namespace math{

template <typename RealType = double,
          typename Policy   = policies::policy<> >
class hyperexponential_distribution;

typedef hyperexponential_distribution<> hyperexponential;

template <typename RealType, typename Policy>
class hyperexponential_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   // Constructors:
   hyperexponential_distribution();  // Default.

   template <typename RateIterT, typename RateIterT2>
   hyperexponential_distribution(  // Default equal probabilities.
                                 RateIterT const& rate_first,
                                 RateIterT2 const& rate_last);  // Rates using Iterators.

   template <typename ProbIterT, typename RateIterT>
   hyperexponential_distribution(ProbIterT prob_first, ProbIterT prob_last,
                                 RateIterT rate_first, RateIterT rate_last);   // Iterators.

   template <typename ProbRangeT, typename RateRangeT>
   hyperexponential_distribution(ProbRangeT const& prob_range,
                                 RateRangeT const& rate_range);  // Ranges.

   template <typename RateRangeT>
   hyperexponential_distribution(RateRangeT const& rate_range);

 #if !defined(BOOST_NO_CXX11_HDR_INITIALIZER_LIST)     // C++11 initializer lists supported.
   hyperexponential_distribution(std::initializer_list<RealType> l1, std::initializer_list<Re↵
alType> l2);
   hyperexponential_distribution(std::initializer_list<RealType> l1);
 #endif

   // Accessors:
   std::size_t num_phases() const;
   std::vector<RealType> probabilities() const;
   std::vector<RealType> rates() const;
};

}} // namespaces
```

Note

An implementation-defined mechanism is provided to avoid ambiguity between constructors accepting ranges, iterators and constants as parameters. This should be transparent to the user. See below and the header file hyperexponential.hpp for details and explanatory comments.

The class type `hyperexponential_distribution` represents a hyperexponential distribution.

A *k*-phase hyperexponential distribution is a continuous probability distribution obtained as a mixture of *k* Exponential Distributions. It is also referred to as *mixed exponential distribution* or parallel *k-phase exponential distribution*.

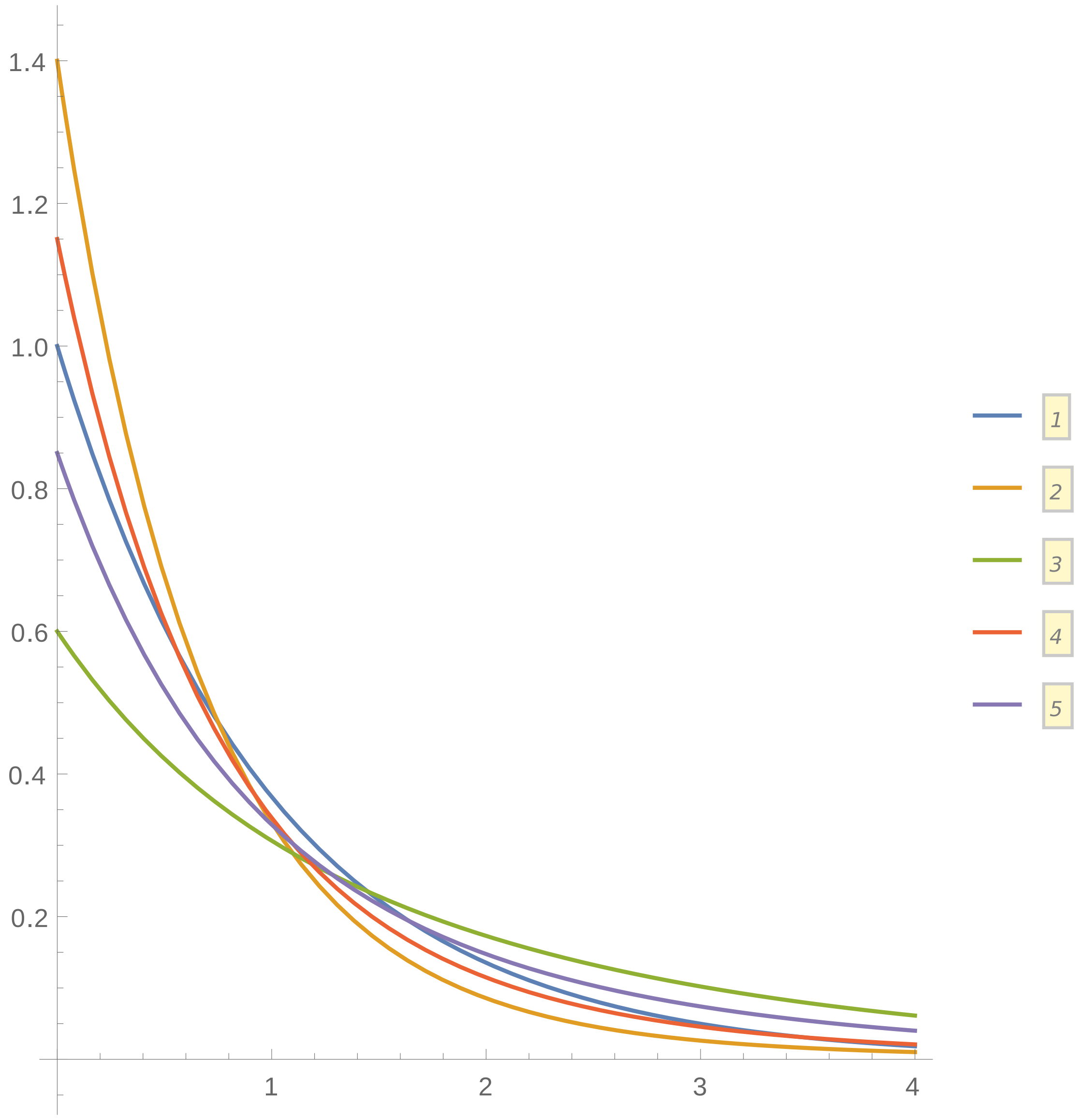
A *k*-phase hyperexponential distribution is characterized by two parameters, namely a *phase probability vector* $\alpha=(\alpha_1,...,\alpha_k)$ and a *rate vector* $\lambda=(\lambda_1,...,\lambda_k)$.

The probability density function for random variate *x* in a hyperexponential distribution is given by:

$$\sum_{i=1}^{k} \alpha_i \lambda_i e^{-\lambda_i x}$$

The following graph illustrates the PDF of the hyperexponential distribution with five different parameters, namely:

1. α=*(1.0)* and λ=*(1.0)* (which degenerates to a simple exponential distribution),
2. α=*(0.1, 0.9)* and λ=*(0.5, 1.5)*,
3. α=*(0.9, 0.1)* and λ=*(0.5, 1.5)*,
4. α=*(0.2, 0.3, 0.5)* and λ=*(0.5, 1.0, 1.5)*,
5. α=*(0.5, 0.3, 0.2)* and λ=*(0.5, 1.0, 1.5)*.



Also, the following graph illustrates the PDF of the hyperexponential distribution (solid lines) where only the *phase probability vector* changes together with the PDF of the two limiting exponential distributions (dashed lines):

1. α=*(0.1, 0.9)* and λ=*(0.5, 1.5)*,

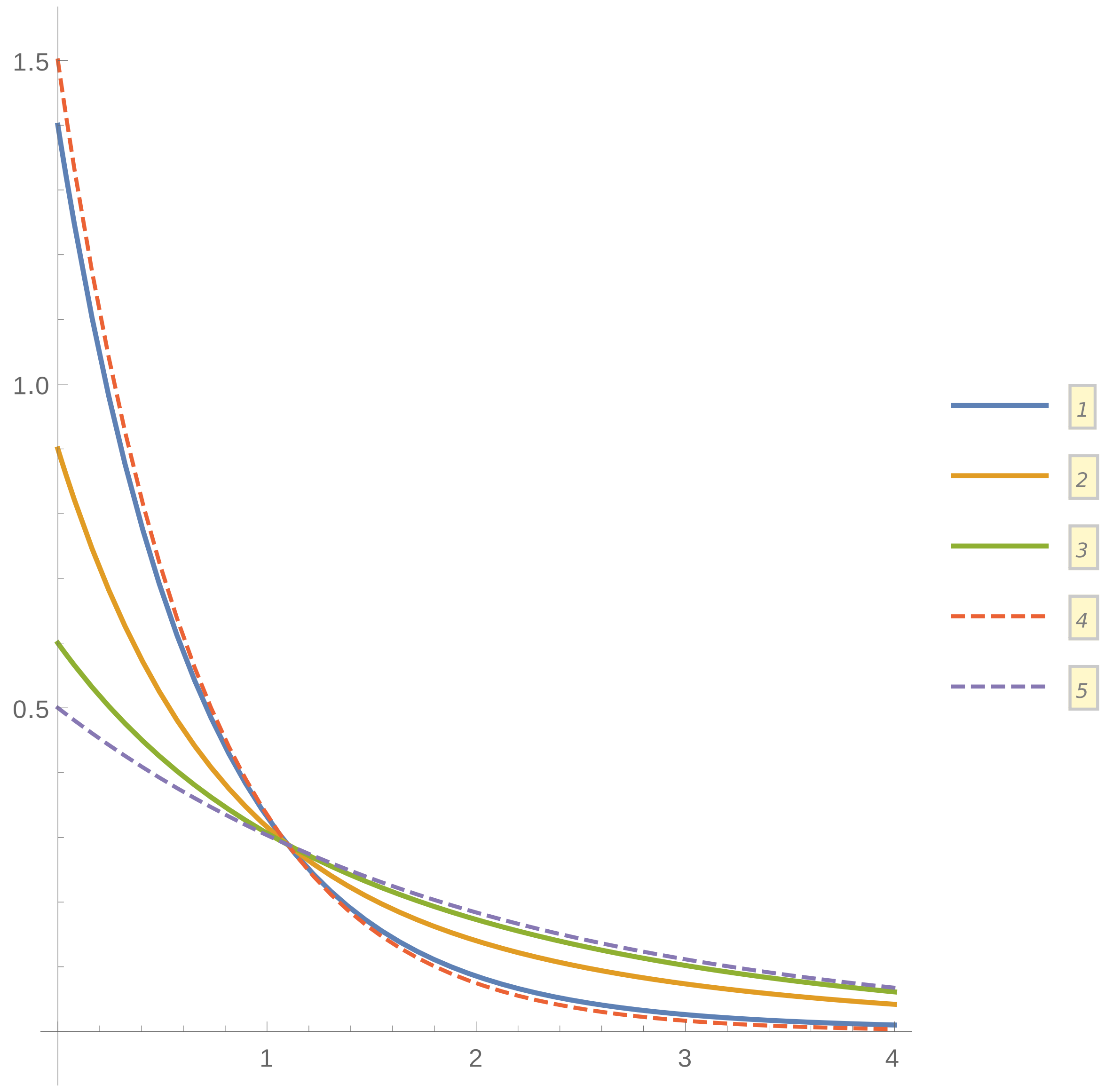
2. $\alpha=(0.6, 0.4)$ and $\lambda=(0.5, 1.5)$,

3. $\alpha=(0.9, 0.1)$ and $\lambda=(0.5, 1.5)$,

4. Exponential distribution with parameter $\lambda=0.5$,

5. Exponential distribution with parameter $\lambda=1.5$.

As expected, as the first element α_1 of the *phase probability vector* approaches to *1* (or, equivalently, α_2 approaches to *0*), the resulting hyperexponential distribution nears the exponential distribution with parameter $\lambda=0.5$. Conversely, as the first element α_2 of the *phase probability vector* approaches to *1* (or, equivalently, α_1 approaches to *0*), the resulting hyperexponential distribution nears the exponential distribution with parameter $\lambda=1.5$.

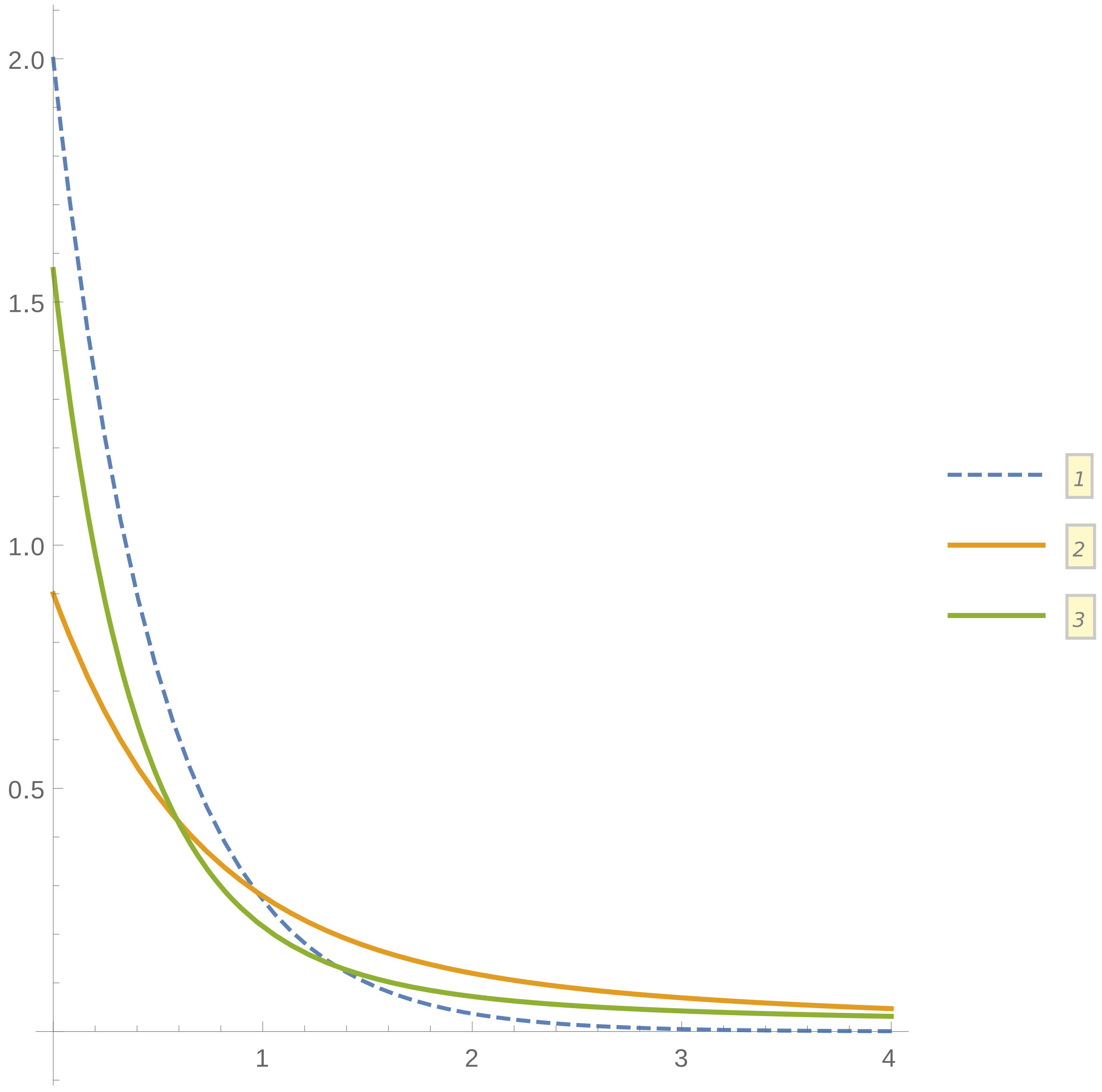


Finally, the following graph compares the PDF of the hyperexponential distribution with different number of phases but with the same mean value equal to *2*:

1. $\alpha=(1.0)$ and $\lambda=(2.0)$ (which degenerates to a simple exponential distribution),

2. $\alpha=(0.5, 0.5)$ and $\lambda=(0.3, 1.5)$,

3. $\alpha=(1.0/3.0, 1.0/3.0, 1.0/3.0)$ and $\lambda=(0.2, 1.5, 3.0)$,



As can be noted, even if the three distributions have the same mean value, the two hyperexponential distributions have a *longer* tail with respect to the one of the exponential distribution. Indeed, the hyperexponential distribution has a larger variability than the exponential distribution, thus resulting in a Coefficient of Variation greater than *1* (as opposed to the one of the exponential distribution which is exactly *1*).

Applications

A k-phase hyperexponential distribution is frequently used in queueing theory to model the distribution of the superposition of k independent events, like, for instance, the service time distribution of a queueing station with k servers in parallel where the i-th server is chosen with probability α_i and its service time distribution is an exponential distribution with rate λ_i (Allen,1990; Papadopolous et al.,1993; Trivedi,2002).

For instance, CPUs service-time distribution in a computing system has often been observed to possess such a distribution (Rosin,1965). Also, the arrival of different types of customer to a single queueing station is often modeled as a hyperexponential distribution (Papadopolous et al.,1993). Similarly, if a product manufactured in several parallel assembly lines and the outputs are merged, the failure density of the overall product is likely to be hyperexponential (Trivedi,2002).

Finally, since the hyperexponential distribution exhibits a high Coefficient of Variation (CoV), that is a CoV > 1, it is especially suited to fit empirical data with large CoV (Feitelson,2014; Wolski et al.,2013) and to approximate long-tail probability distributions (Feldmann et al.,1998).

Related distributions

- When the number of phases *k* is equal to `1`, the hyperexponential distribution is simply an Exponential Distribution.
- When the *k* rates are all equal to λ, the hyperexponential distribution is simple an Exponential Distribution with rate λ.

Examples

Lifetime of Appliances

Suppose a customer is buying an appliance and is choosing at random between an appliance with average lifetime of 10 years and an appliance with average lifetime of 12 years. Assuming the lifetime of this appliance follows an exponential distribution, the lifetime distribution of the purchased appliance can be modeled as a hyperexponential distribution with phase probability vector $\alpha=(1/2,1/2)$ and rate vector $\lambda=(1/10,1/12)$ (Wolfram,2014).

In the rest of this section, we provide an example C++ implementation for computing the average lifetime and the probability that the appliance will work for more than 15 years.

```
#include <boost/math/distributions/hyperexponential.hpp>
#include <iostream>
int main()
{
   const double rates[] = { 1.0 / 10.0, 1.0 / 12.0 };

   boost::math::hyperexponential he(rates);

   std::cout << "Average lifetime: "
      << boost::math::mean(he)
      << " years" << std::endl;
   std::cout << "Probability that the appliance will work for more than 15 years: "
      << boost::math::cdf(boost::math::complement(he, 15.0))
      << std::endl;
}
```

The resulting output is:

```
Average lifetime: 11 years
Probability that the appliance will work for more than 15 years: 0.254817
```

Workloads of Private Cloud Computing Systems

Cloud computing has become a popular metaphor for dynamic and secure self-service access to computational and storage capabilities. In (Wolski et al.,2013), the authors analyze and model workloads gathered from enterprise-operated commercial private clouds and show that 3-phase hyperexponential distributions (fitted using the Expectation Maximization algorithm) capture workload attributes accurately.

In this type of computing system, user requests consist in demanding the provisioning of one or more Virtual Machines (VMs). In particular, in (Wolski et al.,2013) the workload experienced by each cloud system is a function of four distributions, one for each of the following workload attributes:

- *Request Interarrival Time*: the amount of time until the next request,
- *VM Lifetime*: the time duration over which a VM is provisioned to a physical machine,
- *Request Size*: the number of VMs in the request, and
- *Core Count*: the CPU core count requested for each VM.

The authors assume that all VMs in a request have the same core count, but request sizes and core counts can vary from request to request. Moreover, all VMs within a request are assumed to have the same lifetime. Given these assumptions, the authors build a statistical model for the request interarrival time and VM lifetime attributes by fitting their respective data to a 3-phase hyperexponential distribution.

In the following table, we show the sample mean and standard deviation (SD), in seconds, of the request interarrival time and of the VM lifetime distributions of the three datasets collected by authors:

Dataset	**Mean Request Interarrival Time (SD)**	**Mean Multi-core VM Lifetime (SD)**	**Mean Single-core VM Lifetime (SD)**
DS1	2202.1 (2.2e+04)	257173 (4.6e+05)	28754.4 (1.6e+05)
DS2	41285.7 (1.1e+05)	144669.0 (7.9e+05)	599815.0 (1.7e+06)
DS3	11238.8 (3.0e+04)	30739.2 (1.6e+05)	44447.8 (2.2e+05)

Whereas in the following table we show the hyperexponential distribution parameters resulting from the fit:

Dataset	**Request Interarrival Time**	**Multi-core VM Lifetime**	**Single-core VM Lifetime**
DS1	*α=(0.34561,0.08648,0.56791), λ=(0.008,0.00005,0.02894)*	*α=(0.24667,0.37948,0.37385), λ=(0.00004,0.000002,0.00059)*	*α=(0.09325,0.22251,0.68424), λ=(0.000003,0.00109,0.00109)*
DS2	*α=(0.38881,0.18227,0.42892), λ=(0.000006,0.05228,0.00081)*	*α=(0.42093,0.43960,0.13947), λ=(0.00186,0.00008,0.0000008)*	*α=(0.44885,0.30675,0.2444), λ=(0.00143,0.00005,0.0000004)*
DS3	*α=(0.39442,0.24644,0.35914), λ=(0.00030,0.00003,0.00257)*	*α=(0.37621,0.14838,0.47541), λ=(0.00498,0.000005,0.00022)*	*α=(0.34131,0.12544,0.53325), λ=(0.000297,0.000003,0.00410)*

In the rest of this section, we provide an example C++ implementation for computing some statistical properties of the fitted distributions for each of the analyzed dataset.

```
#include <boost/math/distributions.hpp>
#include <iostream>
#include <string>

struct ds_info
{
   std::string name;
   double iat_sample_mean;
   double iat_sample_sd;
   boost::math::hyperexponential iat_he;
   double multi_lt_sample_mean;
   double multi_lt_sample_sd;
   boost::math::hyperexponential multi_lt_he;
   double single_lt_sample_mean;
   double single_lt_sample_sd;
   boost::math::hyperexponential single_lt_he;
};

// DS1 dataset
ds_info make_ds1()
{
   ds_info ds;

   ds.name = "DS1";

   // VM interarrival time distribution
   const double iat_fit_probs[] = { 0.34561,0.08648,0.56791 };
   const double iat_fit_rates[] = { 0.0008,0.00005,0.02894 };
   ds.iat_sample_mean = 2202.1;
   ds.iat_sample_sd = 2.2e+4;
   ds.iat_he = boost::math::hyperexponential(iat_fit_probs, iat_fit_rates);

   // Multi-core VM lifetime distribution
   const double multi_lt_fit_probs[] = { 0.24667,0.37948,0.37385 };
   const double multi_lt_fit_rates[] = { 0.00004,0.000002,0.00059 };
   ds.multi_lt_sample_mean = 257173;
   ds.multi_lt_sample_sd = 4.6e+5;
   ds.multi_lt_he = boost::math::hyperexponential(multi_lt_fit_probs, multi_lt_fit_rates);

   // Single-core VM lifetime distribution
   const double single_lt_fit_probs[] = { 0.09325,0.22251,0.68424 };
   const double single_lt_fit_rates[] = { 0.000003,0.00109,0.00109 };
   ds.single_lt_sample_mean = 28754.4;
   ds.single_lt_sample_sd = 1.6e+5;
   ds.single_lt_he = boost::math::hyperexponential(single_lt_fit_probs, single_lt_fit_rates);

   return ds;
}

// DS2 dataset
ds_info make_ds2()
{
   ds_info ds;

   ds.name = "DS2";

   // VM interarrival time distribution
   const double iat_fit_probs[] = { 0.38881,0.18227,0.42892 };
   const double iat_fit_rates[] = { 0.000006,0.05228,0.00081 };
   ds.iat_sample_mean = 41285.7;
   ds.iat_sample_sd = 1.1e+05;
   ds.iat_he = boost::math::hyperexponential(iat_fit_probs, iat_fit_rates);
   // Multi-core VM lifetime distribution
   const double multi_lt_fit_probs[] = { 0.42093,0.43960,0.13947 };
   const double multi_lt_fit_rates[] = { 0.00186,0.00008,0.0000008 };
   ds.multi_lt_sample_mean = 144669.0;
   ds.multi_lt_sample_sd = 7.9e+05;
   ds.multi_lt_he = boost::math::hyperexponential(multi_lt_fit_probs, multi_lt_fit_rates);

   // Single-core VM lifetime distribution
   const double single_lt_fit_probs[] = { 0.44885,0.30675,0.2444 };
   const double single_lt_fit_rates[] = { 0.00143,0.00005,0.0000004 };
   ds.single_lt_sample_mean = 599815.0;
   ds.single_lt_sample_sd = 1.7e+06;
   ds.single_lt_he = boost::math::hyperexponential(single_lt_fit_probs, single_lt_fit_rates);

   return ds;
}

// DS3 dataset
ds_info make_ds3()
{
   ds_info ds;

   ds.name = "DS3";

   // VM interarrival time distribution
   const double iat_fit_probs[] = { 0.39442,0.24644,0.35914 };
   const double iat_fit_rates[] = { 0.00030,0.00003,0.00257 };
   ds.iat_sample_mean = 11238.8;
   ds.iat_sample_sd = 3.0e+04;
   ds.iat_he = boost::math::hyperexponential(iat_fit_probs, iat_fit_rates);

   // Multi-core VM lifetime distribution
   const double multi_lt_fit_probs[] = { 0.37621,0.14838,0.47541 };
   const double multi_lt_fit_rates[] = { 0.00498,0.000005,0.00022 };
   ds.multi_lt_sample_mean = 30739.2;
   ds.multi_lt_sample_sd = 1.6e+05;
   ds.multi_lt_he = boost::math::hyperexponential(multi_lt_fit_probs, multi_lt_fit_rates);

   // Single-core VM lifetime distribution
   const double single_lt_fit_probs[] = { 0.34131,0.12544,0.53325 };
   const double single_lt_fit_rates[] = { 0.000297,0.000003,0.00410 };
   ds.single_lt_sample_mean = 44447.8;
   ds.single_lt_sample_sd = 2.2e+05;
   ds.single_lt_he = boost::math::hyperexponential(single_lt_fit_probs, single_lt_fit_rates);

   return ds;
}

void print_fitted(ds_info const& ds)
{
   const double secs_in_a_hour = 3600;
   const double secs_in_a_month = 30 * 24 * secs_in_a_hour;

   std::cout << "### " << ds.name << std::endl;
   std::cout << "* Fitted Request Interarrival Time" << std::endl;
  std::cout << " - Mean (SD): " << boost::math::mean(ds.iat_he) << " (" << boost::math::stand↵
ard_deviation(ds.iat_he) << ") seconds." << std::endl;
   std::cout << " - 99th Percentile: ↵
" << boost::math::quantile(ds.iat_he, 0.99) << " seconds." << std::endl;
   std::cout << " - Probability that a VM will arrive within 30 minutes: ↵
" << boost::math::cdf(ds.iat_he, secs_in_a_hour / 2.0) << std::endl;
   std::cout << " - Probability that a VM will arrive after 1 hour: ↵
" << boost::math::cdf(boost::math::complement(ds.iat_he, secs_in_a_hour)) << std::endl;
```

```
   std::cout << "* Fitted Multi-core VM Lifetime" << std::endl;
   std::cout << " - Mean (SD): " << boost::math::mean(ds.multi_lt_he) << " ↵
(" << boost::math::standard_deviation(ds.multi_lt_he) << ") seconds." << std::endl;
   std::cout << " - 99th Percentile: ↵
" << boost::math::quantile(ds.multi_lt_he, 0.99) << " seconds." << std::endl;
   std::cout << " - Probability that a VM will last for less than 1 month: ↵
" << boost::math::cdf(ds.multi_lt_he, secs_in_a_month) << std::endl;
   std::cout << " - Probability that a VM will last for more than 3 months: ↵
" << boost::math::cdf(boost::math::complement(ds.multi_lt_he, 3.0*secs_in_a_month)) << std::endl;
   std::cout << "* Fitted Single-core VM Lifetime" << std::endl;
   std::cout << " - Mean (SD): " << boost::math::mean(ds.single_lt_he) << " ↵
(" << boost::math::standard_deviation(ds.single_lt_he) << ") seconds." << std::endl;
   std::cout << " - 99th Percentile: ↵
" << boost::math::quantile(ds.single_lt_he, 0.99) << " seconds." << std::endl;
   std::cout << " - Probability that a VM will last for less than 1 month: ↵
" << boost::math::cdf(ds.single_lt_he, secs_in_a_month) << std::endl;
   std::cout << " - Probability that a VM will last for more than 3 months: ↵
" << boost::math::cdf(boost::math::comple↵
ment(ds.single_lt_he, 3.0*secs_in_a_month)) << std::endl;
}

int main()
{
   print_fitted(make_ds1());

   print_fitted(make_ds2());

   print_fitted(make_ds3());
}
```

The resulting output (with floating-point precision set to 2) is:

```
### DS1
* Fitted Request Interarrival Time
 - Mean (SD): 2.2e+03 (8.1e+03) seconds.
 - 99th Percentile: 4.3e+04 seconds.
 - Probability that a VM will arrive within 30 minutes: 0.84
 - Probability that a VM will arrive after 1 hour: 0.092
* Fitted Multi-core VM Lifetime
 - Mean (SD): 2e+05 (3.9e+05) seconds.
 - 99th Percentile: 1.8e+06 seconds.
 - Probability that a VM will last for less than 1 month: 1
 - Probability that a VM will last for more than 3 months: 6.7e-08
* Fitted Single-core VM Lifetime
 - Mean (SD): 3.2e+04 (1.4e+05) seconds.
 - 99th Percentile: 7.4e+05 seconds.
 - Probability that a VM will last for less than 1 month: 1
 - Probability that a VM will last for more than 3 months: 6.9e-12
### DS2
* Fitted Request Interarrival Time
 - Mean (SD): 6.5e+04 (1.3e+05) seconds.
 - 99th Percentile: 6.1e+05 seconds.
 - Probability that a VM will arrive within 30 minutes: 0.52
 - Probability that a VM will arrive after 1 hour: 0.4
* Fitted Multi-core VM Lifetime
 - Mean (SD): 1.8e+05 (6.4e+05) seconds.
 - 99th Percentile: 3.3e+06 seconds.
 - Probability that a VM will last for less than 1 month: 0.98
 - Probability that a VM will last for more than 3 months: 0.00028
* Fitted Single-core VM Lifetime
 - Mean (SD): 6.2e+05 (1.6e+06) seconds.
 - 99th Percentile: 8e+06 seconds.
 - Probability that a VM will last for less than 1 month: 0.91
 - Probability that a VM will last for more than 3 months: 0.011
### DS3
* Fitted Request Interarrival Time
 - Mean (SD): 9.7e+03 (2.2e+04) seconds.
 - 99th Percentile: 1.1e+05 seconds.
 - Probability that a VM will arrive within 30 minutes: 0.53
 - Probability that a VM will arrive after 1 hour: 0.36
* Fitted Multi-core VM Lifetime
 - Mean (SD): 3.2e+04 (1e+05) seconds.
 - 99th Percentile: 5.4e+05 seconds.
 - Probability that a VM will last for less than 1 month: 1
 - Probability that a VM will last for more than 3 months: 1.9e-18
* Fitted Single-core VM Lifetime
 - Mean (SD): 4.3e+04 (1.6e+05) seconds.
 - 99th Percentile: 8.4e+05 seconds.
 - Probability that a VM will last for less than 1 month: 1
 - Probability that a VM will last for more than 3 months: 9.3e-12
```

Note

The above results differ from the ones shown in Tables III, V, and VII of (Wolski et al.,2013). We carefully double-checked them with Wolfram Mathematica 10, which confirmed our results.

Member Functions

Default Constructor

```
hyperexponential_distribution();
```

Constructs a *1*-phase hyperexponential distribution (i.e., an exponential distribution) with rate `1`.

Constructor from Iterators

```
template <typename ProbIterT, typename RateIterT>
hyperexponential_distribution(ProbIterT prob_first, ProbIterT prob_last,
                              RateIterT rate_first, RateIterT rate_last);
```

Constructs a hyperexponential distribution with *phase probability vector* parameter given by the range defined by [`prob_first`, `prob_last`) iterator pair, and *rate vector* parameter given by the range defined by the [`rate_first`, `rate_last`) iterator pair.

Parameters

- `prob_first`, `prob_last`: the range of non-negative real elements representing the phase probabilities; elements are normalized to sum to unity.
- `rate_first`, `rate_last`: the range of positive elements representing the rates.

Type Requirements

- `ProbIterT`, `RateIterT`: must meet the requirements of the InputIterator concept.

Example

```
std::array<double, 2> phase_prob = { 0.5, 0.5 };
std::array<double, 2> rates = { 1.0 / 10, 1.0 / 12 };

hyperexponential he(phase_prob.begin(), phase_prob.end(), rates.begin(), rates.end());
```

Construction from Ranges/Containers

```
template <typename ProbRangeT, typename RateRangeT>
hyperexponential_distribution(ProbRangeT const& prob_range,
                              RateRangeT const& rate_range);
```

Constructs a hyperexponential distribution with *phase probability vector* parameter given by the range defined by `prob_range`, and *rate vector* parameter given by the range defined by `rate_range`.

Note

As an implementation detail, this constructor uses Boost's enable_if/disable_if mechanism to disambiguate between this and other 2-argument constructors. Refer to the source code for more details.

Parameters

- `prob_range`: the range of non-negative real elements representing the phase probabilities; elements are normalized to sum to unity.
- `rate_range`: the range of positive real elements representing the rates.

Type Requirements

- `ProbRangeT`, `RateRangeT`: must meet the requirements of the Range concept: that includes native C++ arrays, standard library containers, or a std::pair or iterators.

Examples

```
// We could be using any standard library container here... vector, deque, array, list etc:
std::array<double, 2> phase_prob = { 0.5, 0.5 };
std::array<double, 2> rates      = { 1.0 / 10, 1.0 / 12 };

hyperexponential he1(phase_prob, rates);    // Construct from standard library container.

double phase_probs2[] = { 0.5, 0.5 };
double rates2[]       = { 1.0 / 10, 1.0 / 12 };

hyperexponential he2(phase_probs2, rates2);  // Construct from native C++ array.
```

Construction with rates-iterators (and all phase probabilities equal)

```
template <typename RateIterT, typename RateIterT2>
hyperexponential_distribution(RateIterT const& rate_first,
                              RateIterT2 const& rate_last);
```

Constructs a hyperexponential distribution with *rate vector* parameter given by the range defined by the [`rate_first`, `rate_last`) iterator pair, and *phase probability vector* set to the equal phase probabilities (i.e., to a vector of the same length `n` of the *rate vector* and with each element set to `1.0/n`).

Note

As an implementation detail, this constructor uses Boost's enable_if/disable_if mechanism to disambiguate between this and other 2-argument constructors. Refer to the source code for more details.

Parameters

- `rate_first`, `rate_last`: the range of positive elements representing the rates.

Type Requirements

- `RateIterT`, `RateIterT2`: must meet the requirements of the InputIterator concept.

Example

```
// We could be using any standard library container here... vector, deque, array, list etc:
std::array<double, 2> rates = { 1.0 / 10, 1.0 / 12 };

hyperexponential he(rates.begin(), rates.end());

assert(he.probabilities()[0] == 0.5); // Phase probabilities will be equal and normalised to ↵
unity.
```

Construction from a single range of rates (all phase probabilities will be equal)

```
template <typename RateRangeT>
hyperexponential_distribution(RateRangeT const& rate_range);
```

Constructs a hyperexponential distribution with *rate vector* parameter given by the range defined by `rate_range`, and *phase probability vector* set to the equal phase probabilities (i.e., to a vector of the same length `n` of the *rate vector* and with each element set to `1.0/n`).

Parameters

- `rate_range`: the range of positive real elements representing the rates.

Type Requirements

- `RateRangeT`: must meet the requirements of the Range concept: this includes native C++ array, standard library containers, and a `std::pair` of iterators.

Examples

```
std::array<double, 2> rates = { 1.0 / 10, 1.0 / 12 };

hyperexponential he(rates);

assert(he.probabilities()[0] == 0.5); // Phase probabilities will be equal and normalised to ↵
unity.
```

Construction from Initializer lists

```
hyperexponential_distribution(std::initializer_list<RealType> l1, std::initializer_list<Real↵
Type> l2);
```

Constructs a hyperexponential distribution with *phase probability vector* parameter given by the brace-init-list defined by `l1`, and *rate vector* parameter given by the brace-init-list defined by `l2`.

Parameters

- `l1`: the brace-init-list of non-negative real elements representing the phase probabilities; elements are normalized to sum to unity.
- `l2`: the brace-init-list of positive real elements representing the rates.

The number of elements of the phase probabilities list and the rates list must be the same.

Example

```
hyperexponential he = { { 0.5, 0.5 }, { 1.0 / 10, 1.0 / 12 } };
```

Construction from a single initializer list (all phase probabilities will be equal)

```
hyperexponential_distribution(std::initializer_list<RealType> l1);
```

Constructs a hyperexponential distribution with *rate vector* parameter given by the brace-init-list defined by `l1`, and *phase probability vector* set to the equal phase probabilities (i.e., to a vector of the same length `n` of the *rate vector* and with each element set to `1.0/n`).

Parameters

- `l1`: the brace-init-list of non-negative real elements representing the phase probabilities; they are normalized to ensure that they sum to unity.

Example

```
hyperexponential he = { 1.0 / 10, 1.0 / 12 };

assert(he.probabilities()[0] == 0.5);
```

Accessors

```
std::size_t num_phases() const;
```

Gets the number of phases of this distribution (the size of both the rate and probability vectors).

Return Value

An non-negative integer number representing the number of phases of this distribution.

```
std::vector<RealType> probabilities() const;
```

Gets the *phase probability vector* parameter of this distribution.

Note

The returned probabilities are the **normalized** versions of the probability parameter values passed at construction time.

Return Value

A vector of non-negative real numbers representing the *phase probability vector* parameter of this distribution.

```
std::vector<RealType> rates() const;
```

Gets the *rate vector* parameter of this distribution.

Return Value

A vector of positive real numbers representing the *rate vector* parameter of this distribution.

Warning

The return type of these functions is a vector-by-value. This is deliberate as we wish to hide the actual container used internally which may be subject to future changes (for example to facilitate vectorization of the cdf code etc). Users should note that some code that might otherwise have been expected to work does not. For example, an attempt to output the (normalized) probabilities:

```
std::copy(he.probabilities().begin(), he.probabilities().end(), std::ostream_iter↲
ator<double>(std::cout, " "));
```

fails at compile or runtime because iterator types are incompatible, but, for example,

```
std::cout << he.probabilities()[0] << ' ' << he.probabilities()[1] << std::endl;
```

outputs the expected values.

In general if you want to access a member of the returned container, then assign to a variable first, and then access those members:

```
std::vector<double> t = he.probabilities();
std::copy(t.begin(), t.end(), std::ostream_iterator<double>(std::cout, " "));
```

Non-member Accessor Functions

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The formulae for calculating these are shown in the table below.

Accuracy

The hyperexponential distribution is implemented in terms of the Exponential Distribution and as such should have very small errors, usually an epsilon or few.

Implementation

In the following table:

- $\alpha=(\alpha_1,\ldots,\alpha_k)$ is the *phase probability vector* parameter of the *k*-phase hyperexponential distribution,
- $\lambda=(\lambda_1,\ldots,\lambda_k)$ is the *rate vector* parameter of the *k*-phase hyperexponential distribution,
- x is the random variate.

Function	Implementation Notes
support	$x \in [0,\infty)$
pdf	$\sum_{i=1}^{k} \alpha_i \lambda_i e^{-\lambda_i x}$
cdf	$1 - \sum_{i=1}^{k} \alpha_i e^{-\lambda_i x}$
cdf complement	$\sum_{i=1}^{k} \alpha_i e^{-\lambda_i x}$
quantile	No closed form available. Computed numerically.
quantile from the complement	No closed form available. Computed numerically.
mean	$\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i}$
variance	$2\left(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^2}\right) - \left(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i}\right)^2$
mode	`0`
skewness	$\frac{6(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^3}) - (3(2(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^2}) - (\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i})^2) + (\sum}{(2(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^2}) - (\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i})^2)^{\frac{3}{2}}}$
kurtosis	$\frac{24(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^4}) - 24(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^3})(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i}) + 3(2(2(\sum}{(2(\sum_{i=1}^{k} \frac{\alpha_i}{\lambda_i^2}) - (}$
kurtosis excess	kurtosis - `3`

References

- A.O. Allen, *Probability, Statistics, and Queuing Theory with Computer Science Applications, Second Edition*, Academic Press, 1990.
- D.G. Feitelson, *Workload Modeling for Computer Systems Performance Evaluation*, Cambridge University Press, 2014
- A. Feldmann and W. Whitt, *Fitting mixtures of exponentials to long-tail distributions to analyze network performance models*, Performance Evaluation 31(3-4):245, doi:10.1016/S0166-5316(97)00003-5, 1998.
- H.T. Papadopolous, C. Heavey and J. Browne, *Queueing Theory in Manufacturing Systems Analysis and Design*, Chapman & Hall/CRC, 1993, p. 35.
- R.F. Rosin, *Determining a computing center environment*, Communications of the ACM 8(7):463-468, 1965.
- K.S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, John Wiley & Sons, Inc., 2002.
- Wikipedia, *Hyperexponential Distribution*, Online: http://en.wikipedia.org/wiki/Hyperexponential_distribution, 2014
- R. Wolski and J. Brevik, *Using Parametric Models to Represent Private Cloud Workloads*, IEEE TSC, PrePrint, DOI: 10.1109/TSC.2013.48, 2013.
- Wolfram Mathematica, *Hyperexponential Distribution*, Online: http://reference.wolfram.com/language/ref/HyperexponentialDistribution.html, 2014.

Hypergeometric Distribution

```
#include <boost/math/distributions/hypergeometric.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class hypergeometric_distribution;

template <class RealType, class Policy>
class hypergeometric_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   hypergeometric_distribution(unsigned r, unsigned n, unsigned N);
   // Accessors:
   unsigned total()const;
   unsigned defective()const;
   unsigned sample_count()const;
};

typedef hypergeometric_distribution<> hypergeometric;

}} // namespaces
```

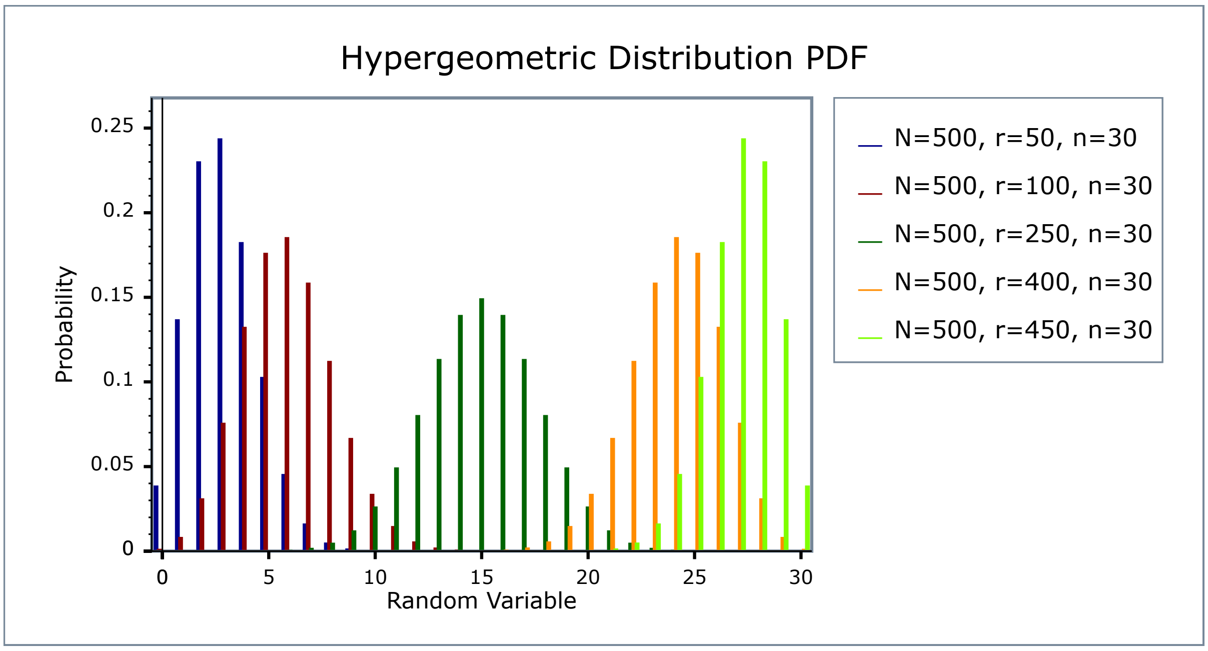
The hypergeometric distribution describes the number of "events" *k* from a sample *n* drawn from a total population *N without replacement*.

Imagine we have a sample of *N* objects of which *r* are "defective" and N-r are "not defective" (the terms "success/failure" or "red/blue" are also used). If we sample *n* items *without replacement* then what is the probability that exactly *k* items in the sample are defective? The answer is given by the pdf of the hypergeometric distribution `f(k; r, n, N)`, whilst the probability of *k* defectives or fewer is given by F(k; r, n, N), where F(k) is the CDF of the hypergeometric distribution.

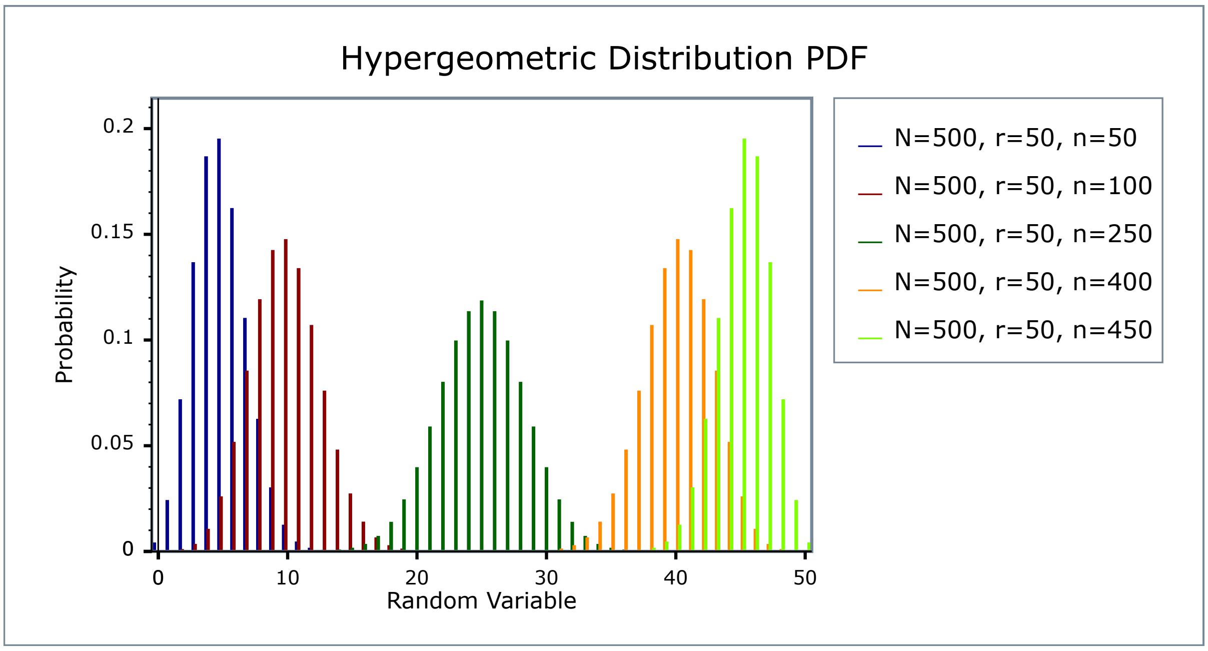
Note

Unlike almost all of the other distributions in this library, the hypergeometric distribution is strictly discrete: it can not be extended to real valued arguments of its parameters or random variable.

The following graph shows how the distribution changes as the proportion of "defective" items changes, while keeping the population and sample sizes constant:



Note that since the distribution is symmetrical in parameters *n* and *r*, if we change the sample size and keep the population and proportion "defective" the same then we obtain basically the same graphs:



Member Functions

```
hypergeometric_distribution(unsigned r, unsigned n, unsigned N);
```

Constructs a hypergeometric distribution with a population of *N* objects, of which *r* are defective, and from which *n* are sampled.

```
unsigned total()const;
```

Returns the total number of objects *N*.

```
unsigned defective()const;
```

Returns the number of objects *r* in population *N* which are defective.

```
unsigned sample_count()const;
```

Returns the number of objects *n* which are sampled from the population *N*.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is the unsigned integers in the range [max(0, n + r - N), min(n, r)]. A domain_error is raised if the random variable is outside this range, or is not an integral value.

Caution

The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on the Hypergeometric distribution. The reference docs describe how to change the rounding policy for these distributions.

However, note that the implementation method of the quantile function always returns an integral value, therefore attempting to use a Policy that requires (or produces) a real valued result will result in a compile time error.

Accuracy

For small N such that `N < boost::math::max_factorial<RealType>::value` then table based lookup of the results gives an accuracy to a few epsilon. `boost::math::max_factorial<RealType>::value` is 170 at double or long double precision.

For larger N such that `N < boost::math::prime(boost::math::max_prime)` then only basic arithmetic is required for the calculation and the accuracy is typically < 20 epsilon. This takes care of N up to 104729.

For `N > boost::math::prime(boost::math::max_prime)` then accuracy quickly degrades, with 5 or 6 decimal digits being lost for N = 110000.

In general for very large N, the user should expect to lose $\log_{10}N$ decimal digits of precision during the calculation, with the results becoming meaningless for N >= 10^{15}.

Testing

There are three sets of tests: our implementation is tested against a table of values produced by Mathematica's implementation of this distribution. We also sanity check our implementation against some spot values computed using the online calculator here http://stattrek.com/Tables/Hypergeometric.aspx. Finally we test accuracy against some high precision test data using this implementation and NTL::RR.

Implementation

The PDF can be calculated directly using the formula:

$$f(k; r, n, N) = \frac{n!r!(N-n)!(N-r)!}{N!k!(n-k)!(r-k)!(N-n-r+k)!} \quad ; \quad \max(0, N-r-n+k) <= k <= \min(r, n)$$

However, this can only be used directly when the largest of the factorials is guaranteed not to overflow the floating point representation used. This formula is used directly when `N < max_factorial<RealType>::value` in which case table lookup of the factorials gives a rapid and accurate implementation method.

For larger *N* the method described in "An Accurate Computation of the Hypergeometric Distribution Function", Trong Wu, ACM Transactions on Mathematical Software, Vol. 19, No. 1, March 1993, Pages 33-43 is used. The method relies on the fact that there is an easy method for factorising a factorial into the product of prime numbers:

$$N! = \prod_{i=0}^{p_i<N} p_i^{e_i}$$

Where p_i is the i'th prime number, and e_i is a small positive integer or zero, which can be calculated via:

$$e_i = \sum_{j=1}^{p_i^j \le N} \text{floor}\left(\frac{N}{p_i^j}\right)$$

Further we can combine the factorials in the expression for the PDF to yield the PDF directly as the product of prime numbers:

$$f(k; r, n, N) = \prod_{i=0}^{p_i<N} p_i^{e_i}$$

With this time the exponents e_i being either positive, negative or zero. Indeed such a degree of cancellation occurs in the calculation of the e_i that many are zero, and typically most have a magnitude or no more than 1 or 2.

Calculation of the product of the primes requires some care to prevent numerical overflow, we use a novel recursive method which splits the calculation into a series of sub-products, with a new sub-product started each time the next multiplication would cause either overflow or underflow. The sub-products are stored in a linked list on the program stack, and combined in an order that will guarantee no overflow or unnecessary-underflow once the last sub-product has been calculated.

This method can be used as long as N is smaller than the largest prime number we have stored in our table of primes (currently 104729). The method is relatively slow (calculating the exponents requires the most time), but requires only a small number of arithmetic operations to calculate the result (indeed there is no shorter method involving only basic arithmetic once the exponents have been found), the method is therefore much more accurate than the alternatives.

For much larger N, we can calculate the PDF from the factorials using either lgamma, or by directly combining lanczos approximations to avoid calculating via logarithms. We use the latter method, as it is usually 1 or 2 decimal digits more accurate than computing via logarithms with lgamma. However, in this area where N > 104729, the user should expect to lose around $\log_{10}N$ decimal digits during the calculation in the worst case.

The CDF and its complement is calculated by directly summing the PDF's. We start by deciding whether the CDF, or its complement, is likely to be the smaller of the two and then calculate the PDF at *k* (or *k+1* if we're calculating the complement) and calculate successive PDF values via the recurrence relations:

$$\begin{aligned} f(k+1; r, n, N) &= \frac{(n-k)(r-k)}{(k+1)(N-n-r+k+1)} f(k; r, n, N) \\ f(k-1; r, n, N) &= \frac{x(N-n-r+k)}{(n-k+1)(r-k+1)} f(k; r, n, N) \end{aligned}$$

Until we either reach the end of the distributions domain, or the next PDF value to be summed would be too small to affect the result.

The quantile is calculated in a similar manner to the CDF: we first guess which end of the distribution we're nearer to, and then sum PDFs starting from the end of the distribution this time, until we have some value *k* that gives the required CDF.

The median is simply the quantile at 0.5, and the remaining properties are calculated via:

$$\begin{aligned} \text{mean} &= \frac{rn}{N} \\ \text{mode} &= \text{floor}\left(\frac{(r+1)(n+1)}{N+2}\right) \\ \text{variance} &= \frac{r\left(\frac{n}{N}\right)\left(1-\frac{n}{N}\right)(N-r)}{(N-1)} \\ \text{skewness} &= \frac{(N-2n)\sqrt{(N-1)}(N-2r)}{\sqrt{rn(N-n)(N-r)}(N-2)} \\ \text{kurtosis excess} &= \left(\frac{N^2(N-1)}{r(N-2)(N-3)(N-r)}\right)\left(\frac{N(N+1)-6N(N-r)}{n(N-n)} + \frac{3r(N-r)(N+6)}{N^2} - 6\right) \end{aligned}$$

Inverse Chi Squared Distribution

```
#include <boost/math/distributions/inverse_chi_squared.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class inverse_chi_squared_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   inverse_chi_squared_distribution(RealType df = 1); // Not explicitly scaled, default 1/df.
   inverse_chi_squared_distribution(RealType df, RealType scale = 1/df);  // Scaled.

   RealType degrees_of_freedom()const; // Default 1.
   RealType scale()const; // Optional scale [xi] (variance), default 1/degrees_of_freedom.
};

}} // namespace boost // namespace math
```

The inverse chi squared distribution is a continuous probability distribution of the **reciprocal** of a variable distributed according to the chi squared distribution.

The sources below give confusingly different formulae using different symbols for the distribution pdf, but they are all the same, or related by a change of variable, or choice of scale.

Two constructors are available to implement both the scaled and (implicitly) unscaled versions.

The main version has an explicit scale parameter which implements the scaled inverse chi_squared distribution.

A second version has an implicit scale = 1/degrees of freedom and gives the 1st definition in the Wikipedia inverse chi_squared distribution. The 2nd Wikipedia inverse chi_squared distribution definition can be implemented by explicitly specifying a scale = 1.

Both definitions are also available in Wolfram Mathematica and in The R Project for Statistical Computing (geoR) with default scale = 1/degrees of freedom.

See

- Inverse chi_squared distribution http://en.wikipedia.org/wiki/Inverse-chi-square_distribution
- Scaled inverse chi_squared distributionhttp://en.wikipedia.org/wiki/Scaled-inverse-chi-square_distribution
- R inverse chi_squared distribution functions R
- Inverse chi_squared distribution functions Weisstein, Eric W. "Inverse Chi-Squared Distribution." From MathWorld--A Wolfram Web Resource.
- Inverse chi_squared distribution reference Weisstein, Eric W. "Inverse Chi-Squared Distribution reference." From Wolfram Mathematica.

The inverse_chi_squared distribution is used in Bayesian statistics: the scaled inverse chi-square is conjugate prior for the normal distribution with known mean, model parameter σ^2 (variance).

See conjugate priors including a table of distributions and their priors.

See also Inverse Gamma Distribution and Chi Squared Distribution.

The inverse_chi_squared distribution is a special case of a inverse_gamma distribution with ν (degrees_of_freedom) shape (α) and scale (β) where

$\alpha = \nu/2$ and $\beta = ½$.

Note

This distribution **does** provide the typedef:

```
typedef inverse_chi_squared_distribution<double> inverse_chi_squared;
```

If you want a `double` precision inverse_chi_squared distribution you can use

```
boost::math::inverse_chi_squared_distribution<>
```

or you can write `inverse_chi_squared my_invchisqr(2, 3);`

For degrees of freedom parameter ν, the (**unscaled**) inverse chi_squared distribution is defined by the probability density function (PDF):

$$f(x;\nu) = 2^{-\nu/2}\ x^{-\nu/2-1}\ e^{-1/2x}\ /\ \Gamma(\nu/2)$$

and Cumulative Density Function (CDF)

$$F(x;\nu) = \Gamma(\nu/2, 1/2x)\ /\ \Gamma(\nu/2)$$

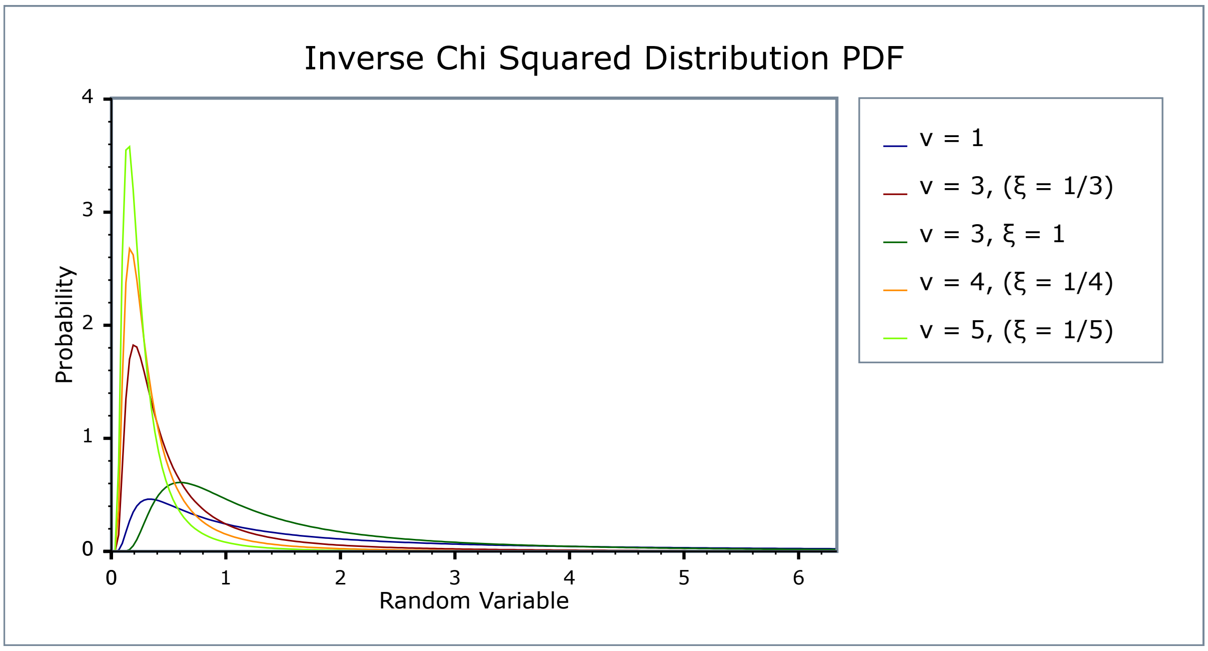
For degrees of freedom parameter ν and scale parameter ξ, the **scaled** inverse chi_squared distribution is defined by the probability density function (PDF):

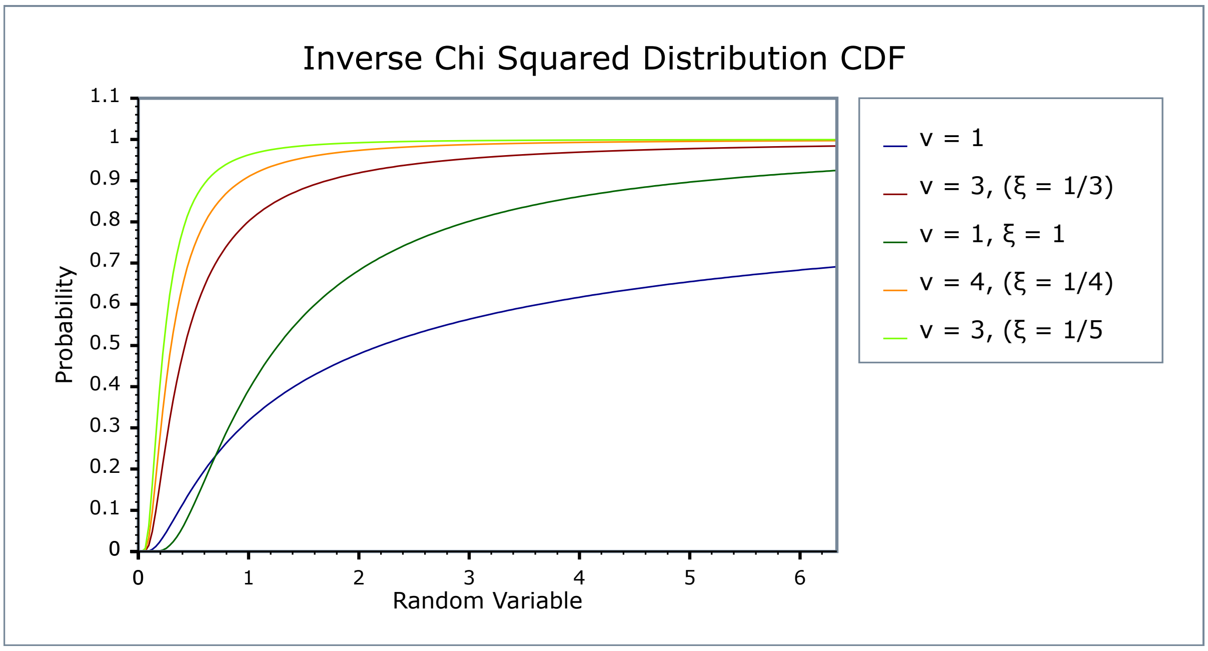
$$f(x;\nu, \xi) = (\xi\nu/2)^{\nu/2} e^{-\nu\xi/2x} x^{-1-\nu/2} / \Gamma(\nu/2)$$

and Cumulative Density Function (CDF)

$$F(x;\nu, \xi) = \Gamma(\nu/2, \nu\xi/2x) / \Gamma(\nu/2)$$

The following graphs illustrate how the PDF and CDF of the inverse chi_squared distribution varies for a few values of parameters ν and ξ:





Member Functions

```
inverse_chi_squared_distribution(RealType df = 1); // Implicitly scaled 1/df.
inverse_chi_squared_distribution(RealType df = 1, RealType scale); // Explicitly scaled.
```

Constructs an inverse chi_squared distribution with ν degrees of freedom *df*, and scale *scale* with default value 1/df.

Requires that the degrees of freedom ν parameter is greater than zero, otherwise calls domain_error.

```
RealType degrees_of_freedom()const;
```

Returns the degrees_of_freedom ν parameter of this distribution.

```
RealType scale()const;
```

Returns the scale ξ parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variate is [0,+∞].

Note

Unlike some definitions, this implementation supports a random variate equal to zero as a special case, returning zero for both pdf and cdf.

Accuracy

The inverse gamma distribution is implemented in terms of the incomplete gamma functions like the Inverse Gamma Distribution that use gamma_p and gamma_q and their inverses gamma_p_inv and gamma_q_inv: refer to the accuracy data for those functions for more information. But in general, gamma (and thus inverse gamma) results are often accurate to a few epsilon, >14 decimal digits accuracy for 64-bit double. unless iteration is involved, as for the estimation of degrees of freedom.

Implementation

In the following table ν is the degrees of freedom parameter and ξ is the scale parameter of the distribution, *x* is the random variate, *p* is the probability and *q = 1-p* its complement. Parameters α for shape and β for scale are used for the inverse gamma function: α = ν/2 and β = ν * ξ/2.

Function	**Implementation Notes**
pdf	Using the relation: pdf = gamma_p_derivative(α, β/ x, β) / x * x
cdf	Using the relation: p = gamma_q(α, β / x)
cdf complement	Using the relation: q = gamma_p(α, β / x)
quantile	Using the relation: x = β / gamma_q_inv(α, p)
quantile from the complement	Using the relation: x = α / gamma_p_inv(α, q)
mode	ν * ξ / (ν + 2)
median	no closed form analytic equation is known, but is evaluated as quantile(0.5)
mean	νξ / (ν - 2) for ν > 2, else a domain_error
variance	2 $ν^2$ $ξ^2$ / ((ν -2)^2 (ν -4)) for ν >4, else a domain_error
skewness	4 √2 √(ν-4) /(ν-6) for ν >6, else a domain_error
kurtosis_excess	12 * (5ν - 22) / ((ν - 6) * (ν - 8)) for ν >8, else a domain_error
kurtosis	3 + 12 * (5ν - 22) / ((ν - 6) * (ν-8)) for ν >8, else a domain_error

References

1. Bayesian Data Analysis, Andrew Gelman, John B. Carlin, Hal S. Stern, Donald B. Rubin, ISBN-13: 978-1584883883, Chapman & Hall; 2 edition (29 July 2003).
2. Bayesian Computation with R, Jim Albert, ISBN-13: 978-0387922973, Springer; 2nd ed. edition (10 Jun 2009)

Inverse Gamma Distribution

```
#include <boost/math/distributions/inverse_gamma.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class inverse_gamma_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   inverse_gamma_distribution(RealType shape, RealType scale = 1)

   RealType shape()const;
   RealType scale()const;
};

}} // namespaces
```

The inverse_gamma distribution is a continuous probability distribution of the reciprocal of a variable distributed according to the gamma distribution.

The inverse_gamma distribution is used in Bayesian statistics.

See inverse gamma distribution.

R inverse gamma distribution functions.

Wolfram inverse gamma distribution.

See also Gamma Distribution.

Note

In spite of potential confusion with the inverse gamma function, this distribution **does** provide the typedef:

```
typedef inverse_gamma_distribution<double> gamma;
```

If you want a `double` precision gamma distribution you can use

```
boost::math::inverse_gamma_distribution<>
```

or you can write `inverse_gamma my_ig(2, 3);`

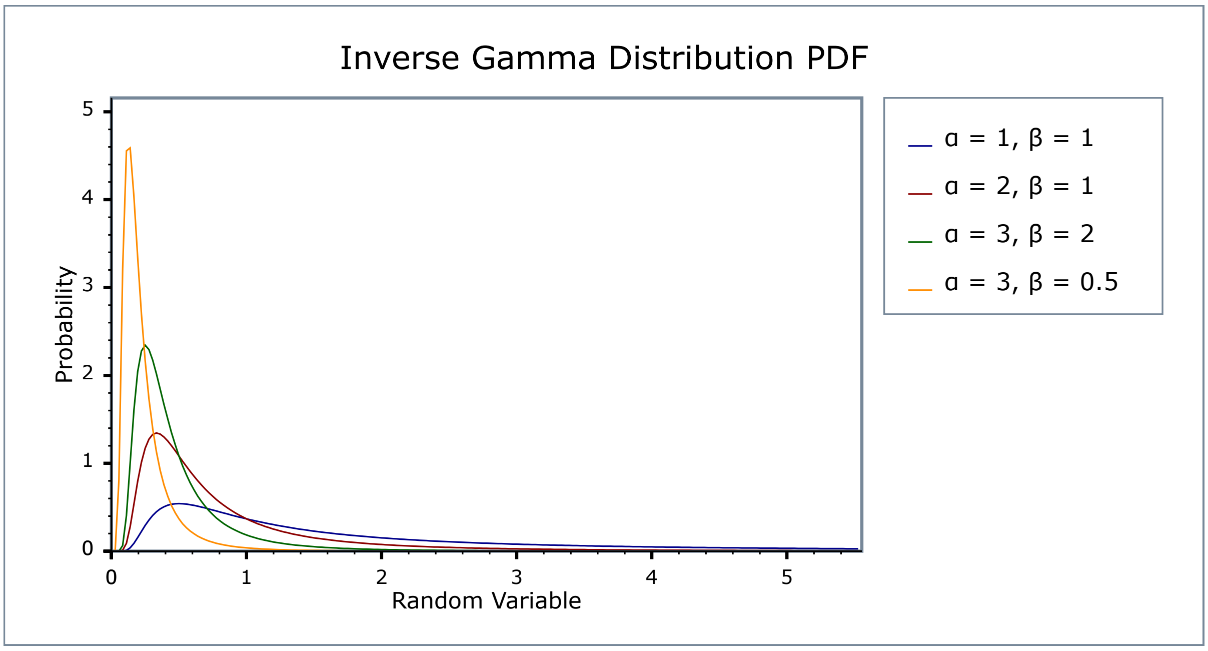
For shape parameter α and scale parameter β, it is defined by the probability density function (PDF):

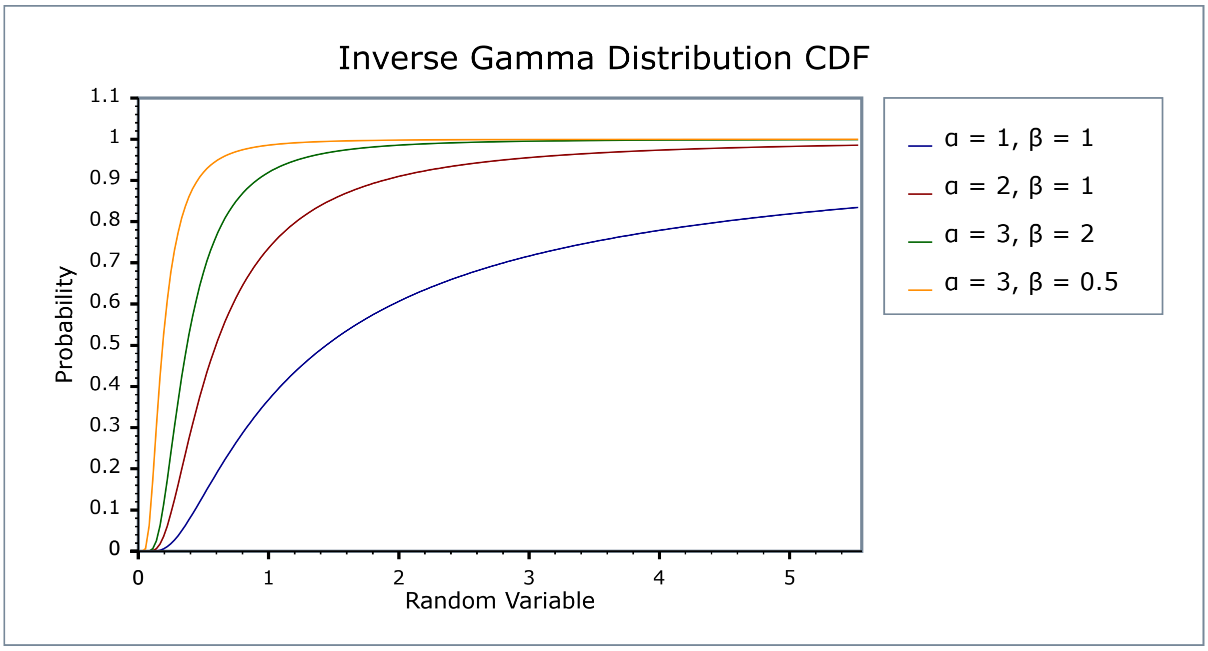
$$f(x;\alpha, \beta) = \beta^{\alpha} * (1/x)^{\alpha+1} \exp(-\beta/x) / \Gamma(\alpha)$$

and cumulative density function (CDF)

$$F(x;\alpha, \beta) = \Gamma(\alpha, \beta/x) / \Gamma(\alpha)$$

The following graphs illustrate how the PDF and CDF of the inverse gamma distribution varies as the parameters vary:





Member Functions

```
inverse_gamma_distribution(RealType shape = 1, RealType scale = 1);
```

Constructs an inverse gamma distribution with shape α and scale β.

Requires that the shape and scale parameters are greater than zero, otherwise calls domain_error.

```
RealType shape()const;
```

Returns the α shape parameter of this inverse gamma distribution.

```
RealType scale()const;
```

Returns the β scale parameter of this inverse gamma distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variate is [0,+∞].

Note

Unlike some definitions, this implementation supports a random variate equal to zero as a special case, returning zero for pdf and cdf.

Accuracy

The inverse gamma distribution is implemented in terms of the incomplete gamma functions gamma_p and gamma_q and their inverses gamma_p_inv and gamma_q_inv: refer to the accuracy data for those functions for more information. But in general, inverse_gamma results are accurate to a few epsilon, >14 decimal digits accuracy for 64-bit double.

Implementation

In the following table α is the shape parameter of the distribution, α is its scale parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = gamma_p_derivative(α, β/ x, β) / x * x
cdf	Using the relation: p = gamma_q(α, β / x)
cdf complement	Using the relation: q = gamma_p(α, β / x)
quantile	Using the relation: x = β / gamma_q_inv(α, p)
quantile from the complement	Using the relation: x = α / gamma_p_inv(α, q)
mode	β / (α + 1)
median	no analytic equation is known, but is evaluated as quantile(0.5)
mean	β / (α - 1) for α > 1, else a domain_error
variance	(β * β) / ((α - 1) * (α - 1) * (α - 2)) for α >2, else a domain_error
skewness	4 * sqrt (α -2) / (α -3) for α >3, else a domain_error
kurtosis_excess	(30 * α - 66) / ((α-3)*(α - 4)) for α >4, else a domain_error

Inverse Gaussian (or Inverse Normal) Distribution

```
#include <boost/math/distributions/inverse_gaussian.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class inverse_gaussian_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;

   inverse_gaussian_distribution(RealType mean = 1, RealType scale = 1);

   RealType mean()const; // mean default 1.
   RealType scale()const; // Optional scale, default 1 (unscaled).
   RealType shape()const; // Shape = scale/mean.
};
typedef inverse_gaussian_distribution<double> inverse_gaussian;

}} // namespace boost // namespace math
```

The Inverse Gaussian distribution distribution is a continuous probability distribution.

The distribution is also called 'normal-inverse Gaussian distribution', and 'normal Inverse' distribution.

It is also convenient to provide unity as default for both mean and scale. This is the Standard form for all distributions. The Inverse Gaussian distribution was first studied in relation to Brownian motion. In 1956 M.C.K. Tweedie used the name Inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time. The inverse Gaussian is one of family of distributions that have been called the Tweedie distributions.

(So *inverse* in the name may mislead: it does **not** relate to the inverse of a distribution).

The tails of the distribution decrease more slowly than the normal distribution. It is therefore suitable to model phenomena where numerically large values are more probable than is the case for the normal distribution. For stock market returns and prices, a key characteristic is that it models that extremely large variations from typical (crashes) can occur even when almost all (normal) variations are small.

Examples are returns from financial assets and turbulent wind speeds.

The normal-inverse Gaussian distributions form a subclass of the generalised hyperbolic distributions.

See distribution. Weisstein, Eric W. "Inverse Gaussian Distribution." From MathWorld--A Wolfram Web Resource.

If you want a `double` precision inverse_gaussian distribution you can use

```
boost::math::inverse_gaussian_distribution<>
```

or, more conveniently, you can write

```
using boost::math::inverse_gaussian;
inverse_gaussian my_ig(2, 3);
```

For mean parameters μ and scale (also called precision) parameter λ, and random variate x, the inverse_gaussian distribution is defined by the probability density function (PDF):

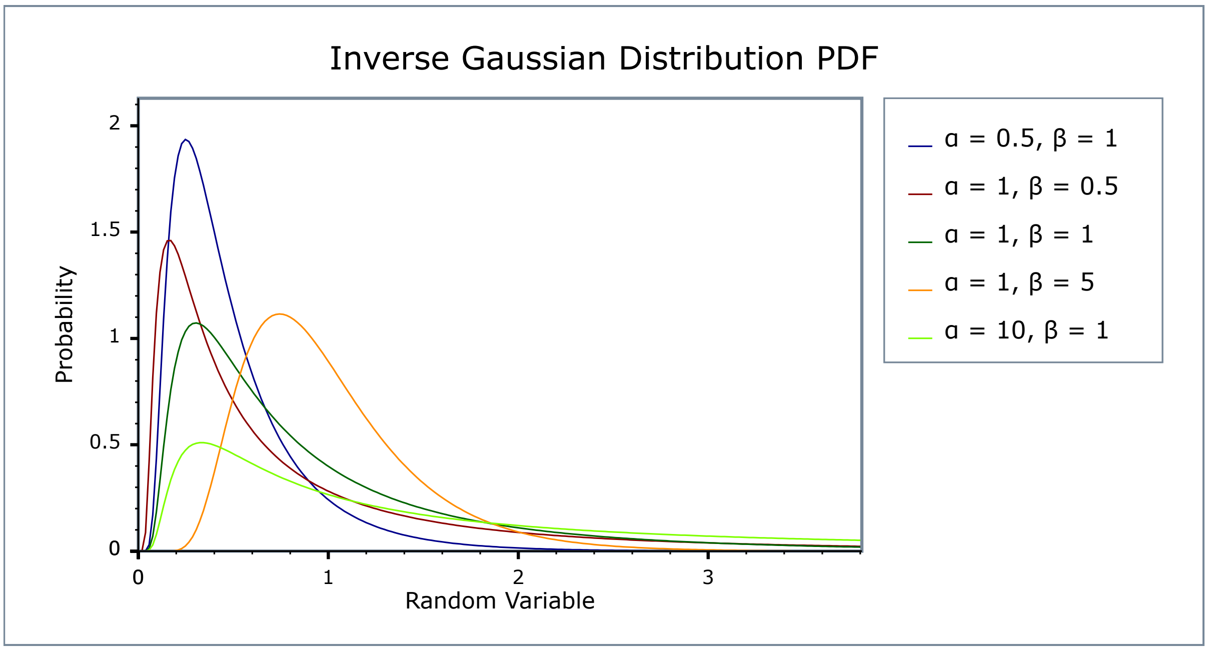
$f(x;\mu, \lambda) = \sqrt{(\lambda/2\pi x^3)}\ e^{-\lambda(x-\mu)^2/2\mu^2 x}$

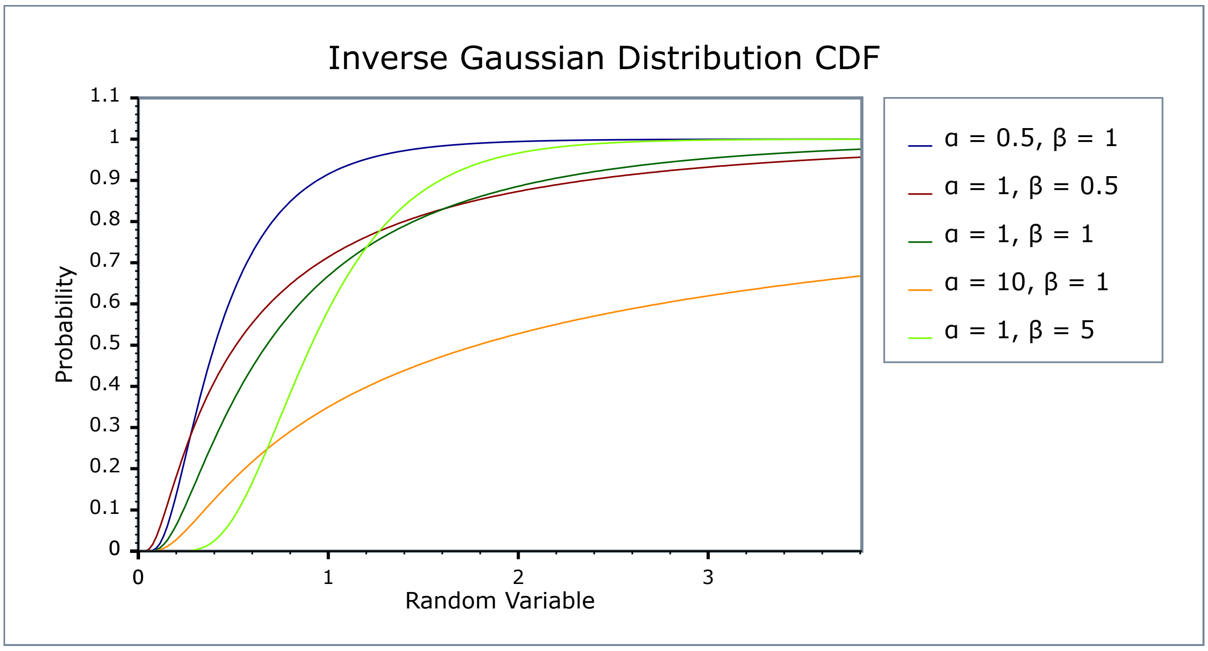
and Cumulative Density Function (CDF):

$F(x;\mu, \lambda) = \Phi\{\sqrt{(\lambda x)}\ (x\mu\text{-}1)\} + e^{2\mu/\lambda}\ \Phi\{-\sqrt{(\lambda/\mu)}\ (1+x/\mu)\}$

where Φ is the standard normal distribution CDF.

The following graphs illustrate how the PDF and CDF of the inverse_gaussian distribution varies for a few values of parameters μ and λ:





Tweedie also provided 3 other parameterisations where (μ and λ) are replaced by their ratio ϕ = λ/μ and by 1/μ: these forms may be more suitable for Bayesian applications. These can be found on Seshadri, page 2 and are also discussed by Chhikara and Folks on page 105. Another related parameterisation, the __wald_distrib (where mean μ is unity) is also provided.

Member Functions

```
inverse_gaussian_distribution(RealType df = 1, RealType scale = 1); // optionally scaled.
```

Constructs an inverse_gaussian distribution with μ mean, and scale λ, with both default values 1.

Requires that both the mean μ parameter and scale λ are greater than zero, otherwise calls domain_error.

```
RealType mean()const;
```

Returns the mean μ parameter of this distribution.

```
RealType scale()const;
```

Returns the scale λ parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variate is [0,+∞).

Note

Unlike some definitions, this implementation supports a random variate equal to zero as a special case, returning zero for both pdf and cdf.

Accuracy

The inverse_gaussian distribution is implemented in terms of the exponential function and standard normal distribution *N*0,1 Φ : refer to the accuracy data for those functions for more information. But in general, gamma (and thus inverse gamma) results are often accurate to a few epsilon, >14 decimal digits accuracy for 64-bit double.

Implementation

In the following table μ is the mean parameter and λ is the scale parameter of the inverse_gaussian distribution, *x* is the random variate, *p* is the probability and *q = 1-p* its complement. Parameters μ for shape and λ for scale are used for the inverse gaussian function.

Function	**Implementation Notes**
pdf	$\sqrt{(\lambda/ 2\pi x^3)}\, e^{-\lambda(x - \mu)^2/ 2\mu^2 x}$
cdf	$\Phi\{\sqrt{(\lambda x)}\, (x\mu-1)\} + e^{2\mu/\lambda}\, \Phi\{-\sqrt{(\lambda/\mu)}\, (1+x/\mu)\}$
cdf complement	using complement of Φ above.
quantile	No closed form known. Estimated using a guess refined by Newton-Raphson iteration.
quantile from the complement	No closed form known. Estimated using a guess refined by Newton-Raphson iteration.
mode	$\mu\, \{\sqrt{(1+9\mu^2/4\lambda^2)^2} - 3\mu/2\lambda\}$
median	No closed form analytic equation is known, but is evaluated as quantile(0.5)
mean	μ
variance	μ^3/λ
skewness	$3 \sqrt{(\mu/\lambda)}$
kurtosis_excess	$15\mu/\lambda$
kurtosis	$12\mu/\lambda$

References

1. Wald, A. (1947). Sequential analysis. Wiley, NY.
2. The Inverse Gaussian distribution : theory, methodology, and applications, Raj S. Chhikara, J. Leroy Folks. ISBN 0824779975 (1989).
3. The Inverse Gaussian distribution : statistical theory and applications, Seshadri, V , ISBN - 0387986189 (pbk) (Dewey 519.2) (1998).
4. Numpy and Scipy Documentation.
5. R statmod invgauss functions.
6. R SuppDists invGauss functions. (Note that these R implementations names differ in case).
7. StatSci.org invgauss help.
8. invgauss R source.
9. pwald, qwald.
10. Brighton Webs wald.

Laplace Distribution

```
#include <boost/math/distributions/laplace.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class laplace_distribution;

typedef laplace_distribution<> laplace;

template <class RealType, class Policy>
class laplace_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   laplace_distribution(RealType location = 0, RealType scale = 1);
   // Accessors:
   RealType location()const;
   RealType scale()const;
};

}} // namespaces
```

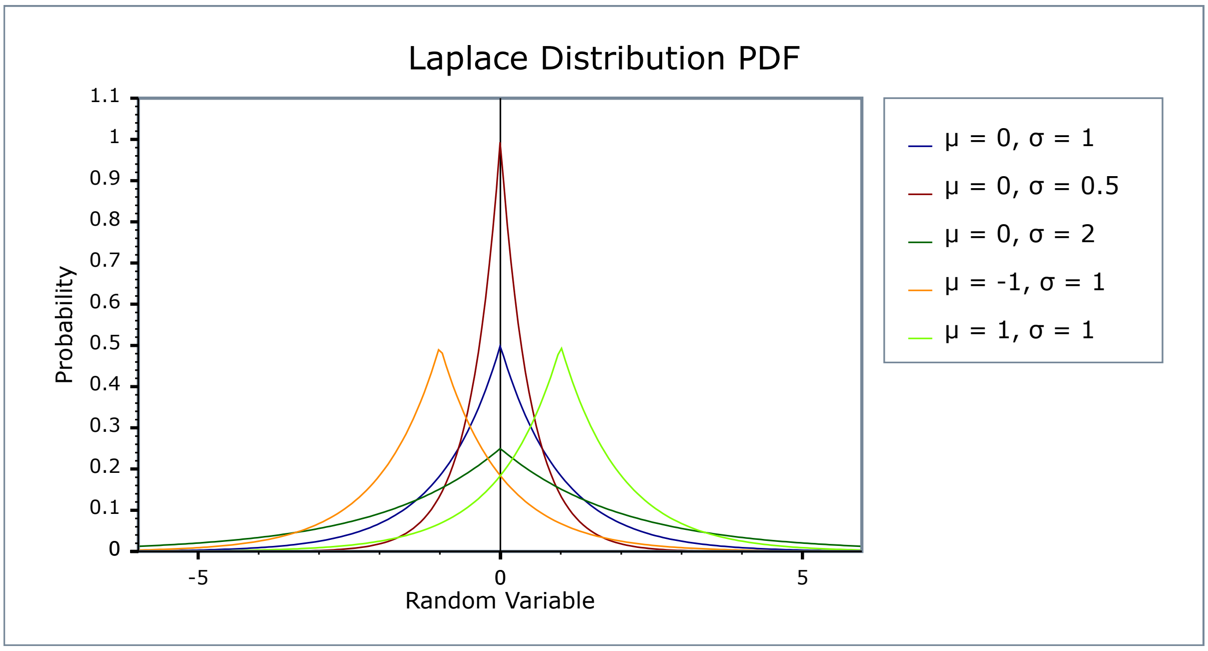
Laplace distribution is the distribution of differences between two independent variates with identical exponential distributions (Abramowitz and Stegun 1972, p. 930). It is also called the double exponential distribution.

For location parameter μ and scale parameter σ , it is defined by the probability density function:

$$\mathrm{f}\,(x;\,\mu,\,\sigma) \quad = \quad \frac{1}{2\sigma} e^{-\frac{|x-\mu|}{\sigma}}$$

The location and scale parameters are equivalent to the mean and standard deviation of the normal or Gaussian distribution.

The following graph illustrates the effect of the parameters μ and σ on the PDF. Note that the domain of the random variable remains [-∞,+∞] irrespective of the value of the location parameter:



Member Functions

```
laplace_distribution(RealType location = 0, RealType scale = 1);
```

Constructs a laplace distribution with location *location* and scale *scale*.

The location parameter is the same as the mean of the random variate.

The scale parameter is proportional to the standard deviation of the random variate.

Requires that the scale parameter is greater than zero, otherwise calls domain_error.

```
RealType location()const;
```

Returns the *location* parameter of this distribution.

```
RealType scale()const;
```

Returns the *scale* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [-∞,+∞].

Accuracy

The laplace distribution is implemented in terms of the standard library log and exp functions and as such should have very small errors.

Implementation

In the following table μ is the location parameter of the distribution, σ is its scale parameter, *x* is the random variate, *p* is the probability and its complement *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = $e^{-abs(x-\mu) / \sigma}$ / (2 * σ)
cdf	Using the relations: x < μ : p = $e^{(x-\mu)/\sigma}$ / σ x >= μ : p = 1 - $e^{(\mu-x)/\sigma}$ / σ
cdf complement	Using the relation: -x < μ : q = $e^{(-x-\mu)/\sigma}$ / σ -x >= μ : q = 1 - $e^{(\mu+x)/\sigma}$ / σ
quantile	Using the relations: p < 0.5 : x = μ + σ * log(2*p) p >= 0.5 : x = μ - σ * log(2-2*p)
quantile from the complement	Using the relation: q > 0.5: x = μ + σ*log(2-2*q) q <=0.5: x = μ - σ*log(2*q)
mean	μ
variance	2 * σ^2
mode	μ
skewness	0
kurtosis	6
kurtosis excess	3

References

- Weisstein, Eric W. "Laplace Distribution." From MathWorld--A Wolfram Web Resource.
- Laplace Distribution
- M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions, 1972, p. 930.

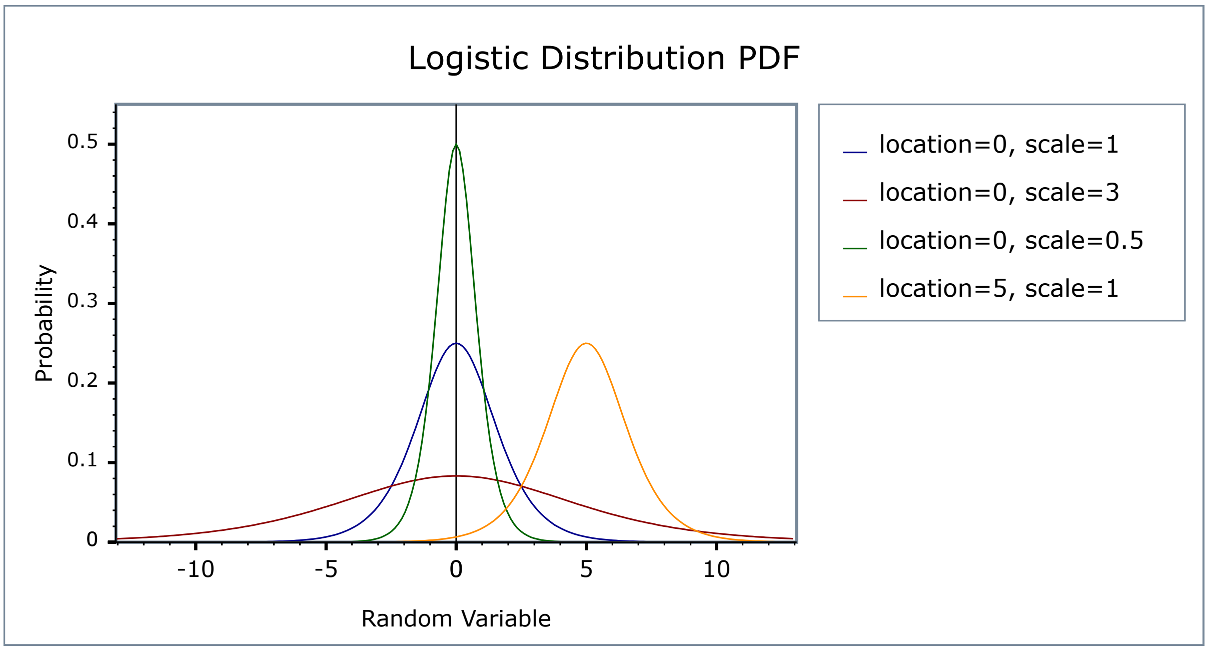
Logistic Distribution

```
#include <boost/math/distributions/logistic.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class logistic_distribution;

template <class RealType, class Policy>
class logistic_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   logistic_distribution(RealType location = 0, RealType scale = 1);
   // Accessors:
   RealType location()const; // location.
   RealType scale()const; // scale.

};

typedef logistic_distribution<> logistic;

}} // namespaces
```

The logistic distribution is a continous probability distribution. It has two parameters - location and scale. The cumulative distribution function of the logistic distribution appears in logistic regression and feedforward neural networks. Among other applications, United State Chess Federation and FIDE use it to calculate chess ratings.

The following graph shows how the distribution changes as the parameters change:



Member Functions

```
logistic_distribution(RealType u = 0, RealType s = 1);
```

Constructs a logistic distribution with location *u* and scale *s*.

Requires `scale > 0`, otherwise a domain_error is raised.

```
RealType location()const;
```

Returns the location of this distribution.

```
RealType scale()const;
```

Returns the scale of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [-[max_value], +[min_value]]. However, the pdf and cdf support inputs of +∞ and -∞ as special cases if RealType permits.

At `p=1` and `p=0`, the quantile function returns the result of +overflow_error and -overflow_error, while the complement quantile function returns the result of -overflow_error and +overflow_error respectively.

Accuracy

The logistic distribution is implemented in terms of the `std::exp` and the `std::log` functions, so its accuracy is related to the accurate implementations of those functions on a given platform. When calculating the quantile with a non-zero *position* parameter catastrophic cancellation errors can occur: in such cases, only a low *absolute error* can be guaranteed.

Implementation

Function	Implementation Notes
pdf	Using the relation: pdf = $e^{-(x-u)/s}$ / $(s*(1+e^{-(x-u)/s})^2)$
cdf	Using the relation: p = $1/(1+e^{-(x-u)/s})$
cdf complement	Using the relation: q = $1/(1+e^{(x-u)/s})$
quantile	Using the relation: x = u - s*log(1/p-1)
quantile from the complement	Using the relation: x = u + s*log(p/1-p)
mean	u
mode	The same as the mean.
skewness	0
kurtosis excess	6/5
variance	$(\pi*s)^2 / 3$

Log Normal Distribution

```
#include <boost/math/distributions/lognormal.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class lognormal_distribution;

typedef lognormal_distribution<> lognormal;

template <class RealType, class Policy>
class lognormal_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   lognormal_distribution(RealType location = 0, RealType scale = 1);
   // Accessors:
   RealType location()const;
   RealType scale()const;
};

}} // namespaces
```

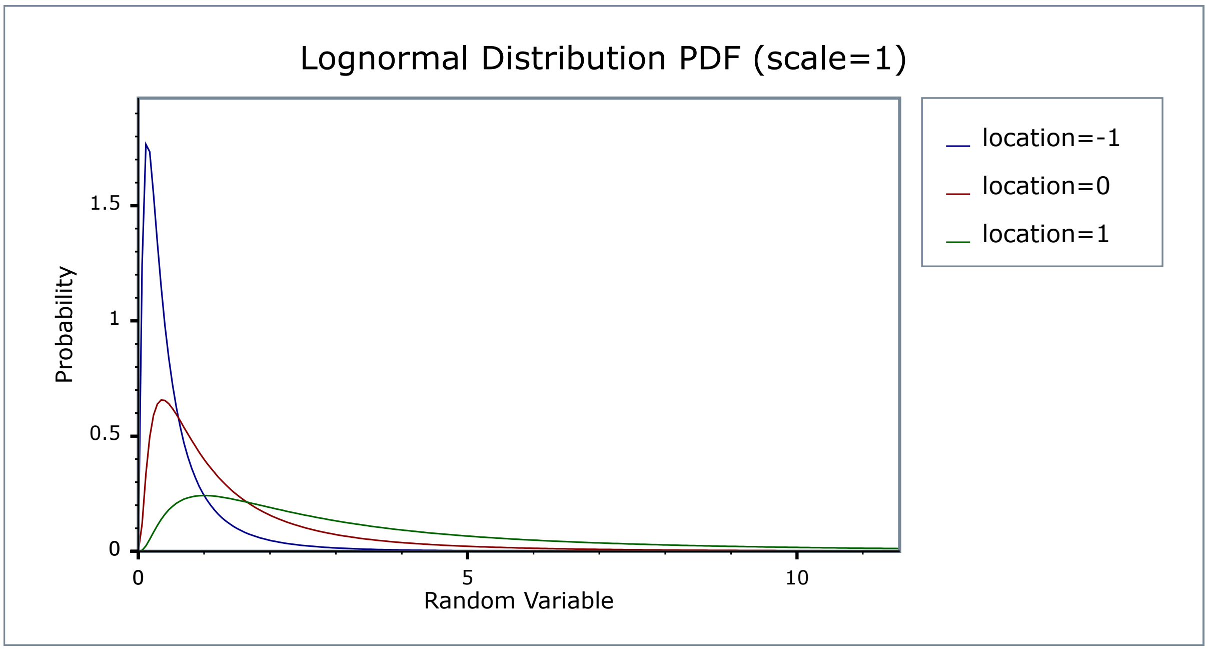
The lognormal distribution is the distribution that arises when the logarithm of the random variable is normally distributed. A lognormal distribution results when the variable is the product of a large number of independent, identically-distributed variables.

For location and scale parameters *m* and *s* it is defined by the probability density function:

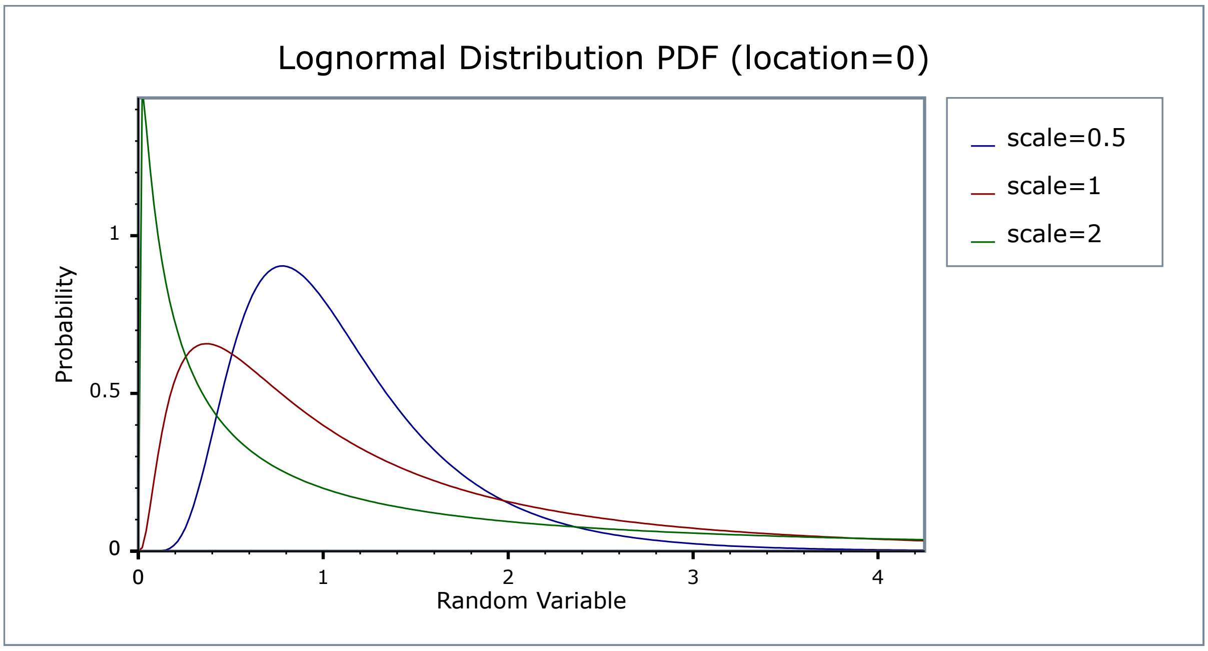
$$f(x) = \frac{1}{xs\sqrt{2\pi}} e^{\frac{-(\ln x - m)^2}{2s^2}}$$

The location and scale parameters are equivalent to the mean and standard deviation of the logarithm of the random variable.

The following graph illustrates the effect of the location parameter on the PDF, note that the range of the random variable remains [0,+∞] irrespective of the value of the location parameter:



The next graph illustrates the effect of the scale parameter on the PDF:



Member Functions

```
lognormal_distribution(RealType location = 0, RealType scale = 1);
```

Constructs a lognormal distribution with location *location* and scale *scale*.

The location parameter is the same as the mean of the logarithm of the random variate.

The scale parameter is the same as the standard deviation of the logarithm of the random variate.

Requires that the scale parameter is greater than zero, otherwise calls domain_error.

```
RealType location()const;
```

Returns the *location* parameter of this distribution.

```
RealType scale()const;
```

Returns the *scale* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0,+∞].

Accuracy

The lognormal distribution is implemented in terms of the standard library log and exp functions, plus the error function, and as such should have very low error rates.

Implementation

In the following table *m* is the location parameter of the distribution, *s* is its scale parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	Implementation Notes
pdf	Using the relation: pdf = $e^{-(\ln(x) - m)^2 / 2s^2}$ / (x * s * sqrt(2pi))
cdf	Using the relation: p = cdf(normal_distribtion<RealType>(m, s), log(x))
cdf complement	Using the relation: q = cdf(complement(normal_distribtion<RealType>(m, s), log(x)))
quantile	Using the relation: x = exp(quantile(normal_distribtion<RealType>(m, s), p))
quantile from the complement	Using the relation: x = exp(quantile(complement(normal_distribtion<RealType>(m, s), q)))
mean	$e^{m + s^2 / 2}$
variance	$(e^{s^2} - 1) * e^{2m + s^2}$
mode	$e^{m - s^2}$
skewness	$\mathrm{sqrt}(e^{s^2} - 1) * (2 + e^{s^2})$
kurtosis	$e^{4s^2} + 2e^{3s^2} + 3e^{2s^2} - 3$
kurtosis excess	$e^{4s^2} + 2e^{3s^2} + 3e^{2s^2} - 6$

Negative Binomial Distribution

```
#include <boost/math/distributions/negative_binomial.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class negative_binomial_distribution;

typedef negative_binomial_distribution<> negative_binomial;

template <class RealType, class Policy>
class negative_binomial_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Constructor from successes and success_fraction:
   negative_binomial_distribution(RealType r, RealType p);

   // Parameter accessors:
   RealType success_fraction() const;
   RealType successes() const;

   // Bounds on success fraction:
   static RealType find_lower_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability); // alpha
   static RealType find_upper_bound_on_p(
      RealType trials,
      RealType successes,
      RealType probability); // alpha

   // Estimate min/max number of trials:
   static RealType find_minimum_number_of_trials(
      RealType k,     // Number of failures.
      RealType p,     // Success fraction.
      RealType probability); // Probability threshold alpha.
   static RealType find_maximum_number_of_trials(
      RealType k,     // Number of failures.
      RealType p,     // Success fraction.
      RealType probability); // Probability threshold alpha.
};

}} // namespaces
```

The class type `negative_binomial_distribution` represents a negative_binomial distribution: it is used when there are exactly two mutually exclusive outcomes of a Bernoulli trial: these outcomes are labelled "success" and "failure".

For k + r Bernoulli trials each with success fraction p, the negative_binomial distribution gives the probability of observing k failures and r successes with success on the last trial. The negative_binomial distribution assumes that success_fraction p is fixed for all (k + r) trials.

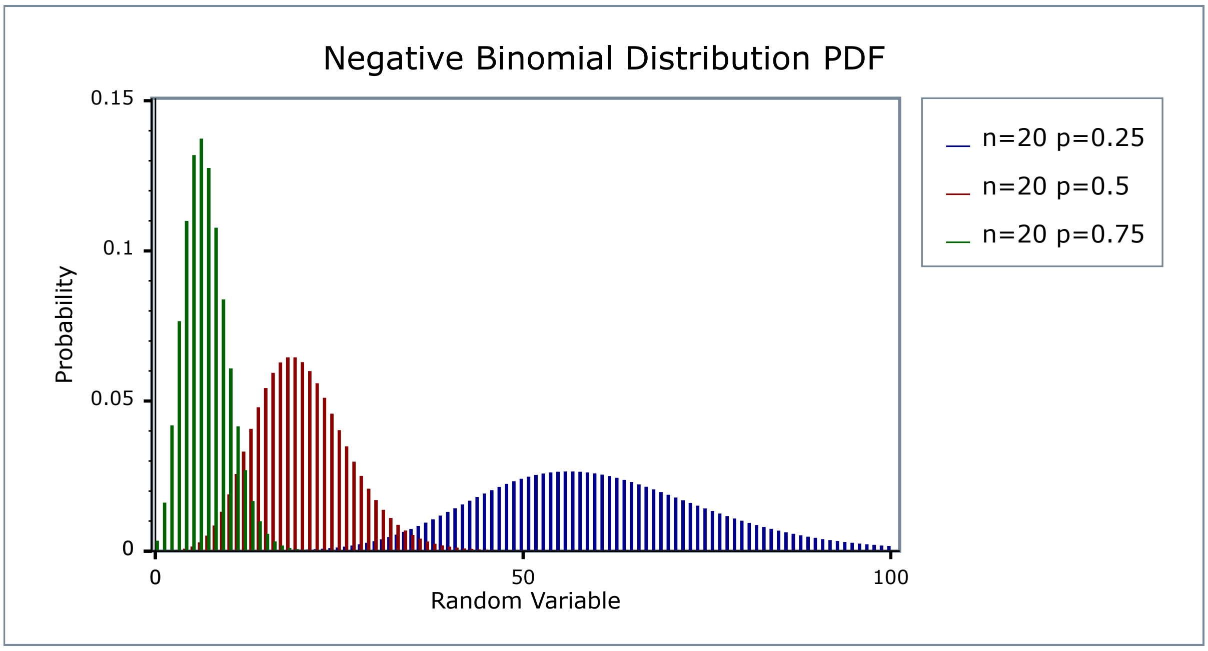
Note

The random variable for the negative binomial distribution is the number of trials, (the number of successes is a fixed property of the distribution) whereas for the binomial, the random variable is the number of successes, for a fixed number of trials.

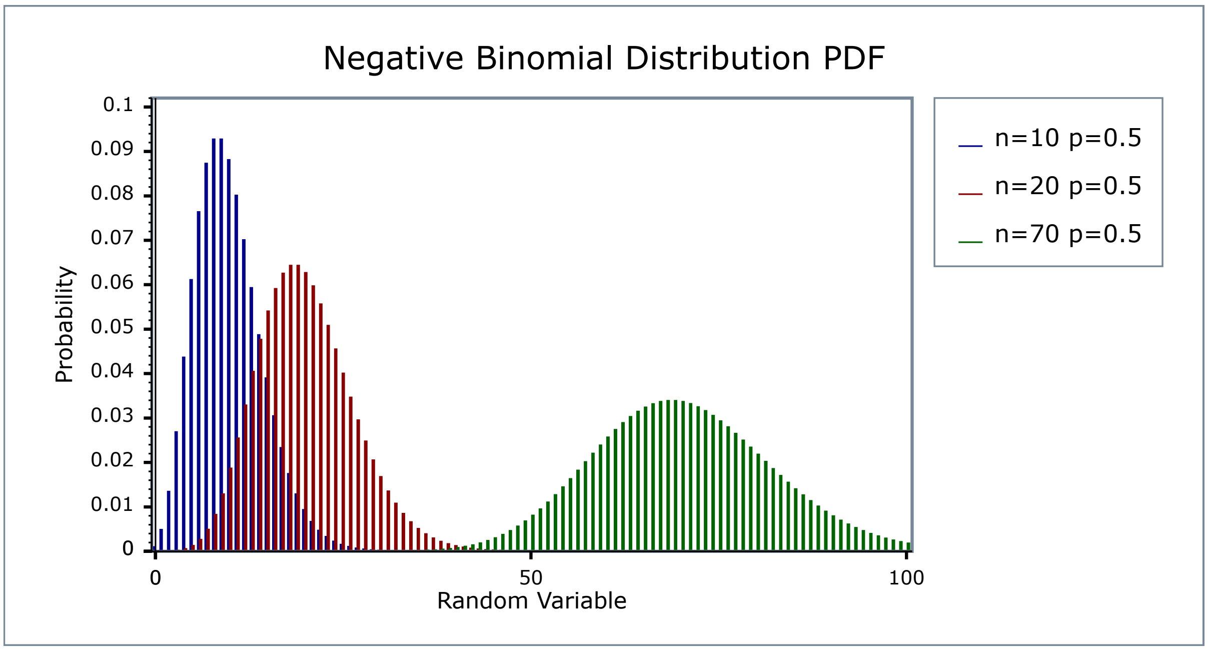
It has the PDF:

$$f(k;\, r,\ p) \quad = \quad \frac{\Gamma(r+k)}{k!\Gamma(r)} p^r (1-p)^k$$

The following graph illustrate how the PDF varies as the success fraction *p* changes:



Alternatively, this graph shows how the shape of the PDF varies as the number of successes changes:



Related Distributions

The name negative binomial distribution is reserved by some to the case where the successes parameter r is an integer. This integer version is also called the Pascal distribution.

This implementation uses real numbers for the computation throughout (because it uses the **real-valued** incomplete beta function family of functions). This real-valued version is also called the Polya Distribution.

The Poisson distribution is a generalization of the Pascal distribution, where the success parameter r is an integer: to obtain the Pascal distribution you must ensure that an integer value is provided for r, and take integer values (floor or ceiling) from functions that return a number of successes.

For large values of r (successes), the negative binomial distribution converges to the Poisson distribution.

The geometric distribution is a special case where the successes parameter r = 1, so only a first and only success is required. geometric(p) = negative_binomial(1, p).

The Poisson distribution is a special case for large successes

$\text{poisson}(\lambda) = \lim_{r \to \infty} \text{negative\_binomial}(r, r / (\lambda + r)))$

Caution

The Negative Binomial distribution is a discrete distribution: internally, functions like the `cdf` and `pdf` are treated "as if" they are continuous functions, but in reality the results returned from these functions only have meaning if an integer value is provided for the random variate argument.

The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on the Negative Binomial distribution. The reference docs describe how to change the rounding policy for these distributions.

Member Functions

Construct

```
negative_binomial_distribution(RealType r, RealType p);
```

Constructor: *r* is the total number of successes, *p* is the probability of success of a single trial.

Requires: `r > 0` and `0 <= p <= 1`.

Accessors

```
RealType success_fraction() const; // successes / trials (0 <= p <= 1)
```

Returns the parameter *p* from which this distribution was constructed.

```
RealType successes() const; // required successes (r > 0)
```

Returns the parameter *r* from which this distribution was constructed.

The best method of calculation for the following functions is disputed: see Binomial Distribution for more discussion.

Lower Bound on Parameter p

```
static RealType find_lower_bound_on_p(
  RealType failures,
  RealType successes,
  RealType probability) // (0 <= alpha <= 1), 0.05 equivalent to 95% confidence.
```

Returns a **lower bound** on the success fraction:

failures	The total number of failures before the *r*th success.
successes	The number of successes required.
alpha	The largest acceptable probability that the true value of the success fraction is **less than** the value returned.

For example, if you observe *k* failures and *r* successes from *n* = k + r trials the best estimate for the success fraction is simply *r/n*, but if you want to be 95% sure that the true value is **greater than** some value, p_{min}, then:

```
pmin = negative_binomial_distribution<RealType>::find_lower_bound_on_p(
                    failures, successes, 0.05);
```

See negative binomial confidence interval example.

This function uses the Clopper-Pearson method of computing the lower bound on the success fraction, whilst many texts refer to this method as giving an "exact" result in practice it produces an interval that guarantees *at least* the coverage required, and may produce pessimistic estimates for some combinations of *failures* and *successes*. See:

Yong Cai and K. Krishnamoorthy, A Simple Improved Inferential Method for Some Discrete Distributions. Computational statistics and data analysis, 2005, vol. 48, no3, 605-621.

Upper Bound on Parameter p

```
static RealType find_upper_bound_on_p(
   RealType trials,
   RealType successes,
   RealType alpha); // (0 <= alpha <= 1), 0.05 equivalent to 95% confidence.
```

Returns an **upper bound** on the success fraction:

trials	The total number of trials conducted.
successes	The number of successes that occurred.
alpha	The largest acceptable probability that the true value of the success fraction is **greater than** the value returned.

For example, if you observe *k* successes from *n* trials the best estimate for the success fraction is simply *k/n*, but if you want to be 95% sure that the true value is **less than** some value, p_{max}, then:

```
pmax = negative_binomial_distribution<RealType>::find_upper_bound_on_p(
                    r, k, 0.05);
```

See negative binomial confidence interval example.

This function uses the Clopper-Pearson method of computing the lower bound on the success fraction, whilst many texts refer to this method as giving an "exact" result in practice it produces an interval that guarantees *at least* the coverage required, and may produce pessimistic estimates for some combinations of *failures* and *successes*. See:

Yong Cai and K. Krishnamoorthy, A Simple Improved Inferential Method for Some Discrete Distributions. Computational statistics and data analysis, 2005, vol. 48, no3, 605-621.

Estimating Number of Trials to Ensure at Least a Certain Number of Failures

```
static RealType find_minimum_number_of_trials(
   RealType k,     // number of failures.
   RealType p,     // success fraction.
   RealType alpha); // probability threshold (0.05 equivalent to 95%).
```

This functions estimates the number of trials required to achieve a certain probability that **more than k failures will be observed**.

k	The target number of failures to be observed.
p	The probability of *success* for each trial.
alpha	The maximum acceptable risk that only k failures or fewer will be observed.

For example:

```
negative_binomial_distribution<RealType>::find_minimum_number_of_trials(10, 0.5, 0.05);
```

Returns the smallest number of trials we must conduct to be 95% sure of seeing 10 failures that occur with frequency one half.

Worked Example.

This function uses numeric inversion of the negative binomial distribution to obtain the result: another interpretation of the result, is that it finds the number of trials (success+failures) that will lead to an *alpha* probability of observing k failures or fewer.

Estimating Number of Trials to Ensure a Maximum Number of Failures or Less

```
static RealType find_maximum_number_of_trials(
   RealType k,     // number of failures.
   RealType p,     // success fraction.
   RealType alpha); // probability threshold (0.05 equivalent to 95%).
```

This functions estimates the maximum number of trials we can conduct and achieve a certain probability that **k failures or fewer will be observed**.

k	The maximum number of failures to be observed.
p	The probability of *success* for each trial.
alpha	The maximum acceptable *risk* that more than k failures will be observed.

For example:

```
negative_binomial_distribution<RealType>::find_maximum_number_of_tri↵
als(0, 1.0-1.0/1000000, 0.05);
```

Returns the largest number of trials we can conduct and still be 95% sure of seeing no failures that occur with frequency one in one million.

This function uses numeric inversion of the negative binomial distribution to obtain the result: another interpretation of the result, is that it finds the number of trials (success+failures) that will lead to an *alpha* probability of observing more than k failures.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

However it's worth taking a moment to define what these actually mean in the context of this distribution:

Table 17. Meaning of the non-member accessors.

Function	Meaning
Probability Density Function	The probability of obtaining **exactly k failures** from k+r trials with success fraction p. For example: `pdf(negative_binomial(r, p), k)`
Cumulative Distribution Function	The probability of obtaining **k failures or fewer** from k+r trials with success fraction p and success on the last trial. For example: `cdf(negative_binomial(r, p), k)`
Complement of the Cumulative Distribution Function	The probability of obtaining **more than k failures** from k+r trials with success fraction p and success on the last trial. For example: `cdf(complement(negative_binomial(r, p), k))`
Quantile	The **greatest** number of failures k expected to be observed from k+r trials with success fraction p, at probability P. Note that the value returned is a real-number, and not an integer. Depending on the use case you may want to take either the floor or ceiling of the real result. For example: `quantile(negative_binomial(r, p), P)`
Quantile from the complement of the probability	The **smallest** number of failures k expected to be observed from k+r trials with success fraction p, at probability P. Note that the value returned is a real-number, and not an integer. Depending on the use case you may want to take either the floor or ceiling of the real result. For example: `quantile(complement(negative_binomi↵ al(r, p), P))`

Accuracy

This distribution is implemented using the incomplete beta functions ibeta and ibetac: please refer to these functions for information on accuracy.

Implementation

In the following table, *p* is the probability that any one trial will be successful (the success fraction), *r* is the number of successes, *k* is the number of failures, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	pdf = exp(lgamma(r + k) - lgamma(r) - lgamma(k+1)) * pow(p, r) * pow((1-p), k) Implementation is in terms of ibeta_derivative: (p/(r + k)) * ibeta_derivative(r, static_cast<RealType>(k+1), p) The function ibeta_derivative is used here, since it has already been optimised for the lowest possible error - indeed this is really just a thin wrapper around part of the internals of the incomplete beta function.
cdf	Using the relation: cdf = I_p(r, k+1) = ibeta(r, k+1, p) = ibeta(r, static_cast<RealType>(k+1), p)
cdf complement	Using the relation: 1 - cdf = I_p(k+1, r) = ibetac(r, static_cast<RealType>(k+1), p)
quantile	ibeta_invb(r, p, P) - 1
quantile from the complement	ibetac_invb(r, p, Q) -1)
mean	`r(1-p)/p`
variance	`r (1-p) / p * p`
mode	`floor((r-1) * (1 - p)/p)`
skewness	`(2 - p) / sqrt(r * (1 - p))`
kurtosis	`6 / r + (p * p) / r * (1 - p )`
kurtosis excess	`6 / r + (p * p) / r * (1 - p ) -3`
parameter estimation member functions	
`find_lower_bound_on_p`	ibeta_inv(successes, failures + 1, alpha)
`find_upper_bound_on_p`	ibetac_inv(successes, failures, alpha) plus see comments in code.
`find_minimum_number_of_trials`	ibeta_inva(k + 1, p, alpha)
`find_maximum_number_of_trials`	ibetac_inva(k + 1, p, alpha)

Implementation notes:

- The real concept type (that deliberately lacks the Lanczos approximation), was found to take several minutes to evaluate some extreme test values, so the test has been disabled for this type.
- Much greater speed, and perhaps greater accuracy, might be achieved for extreme values by using a normal approximation. This is NOT been tested or implemented.

Noncentral Beta Distribution

```
#include <boost/math/distributions/non_central_beta.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class non_central_beta_distribution;

typedef non_central_beta_distribution<> non_central_beta;

template <class RealType, class Policy>
class non_central_beta_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor:
   non_central_beta_distribution(RealType alpha, RealType beta, RealType lambda);

   // Accessor to shape parameters:
   RealType alpha()const;
   RealType beta()const;

   // Accessor to non-centrality parameter lambda:
   RealType non_centrality()const;
};

}} // namespaces
```

The noncentral beta distribution is a generalization of the Beta Distribution.

It is defined as the ratio $X = \chi_m^2(\lambda) / (\chi_m^2(\lambda) + \chi_n^2)$ where $\chi_m^2(\lambda)$ is a noncentral χ^2 random variable with *m* degrees of freedom, and χ_n^2 is a central χ^2 random variable with *n* degrees of freedom.

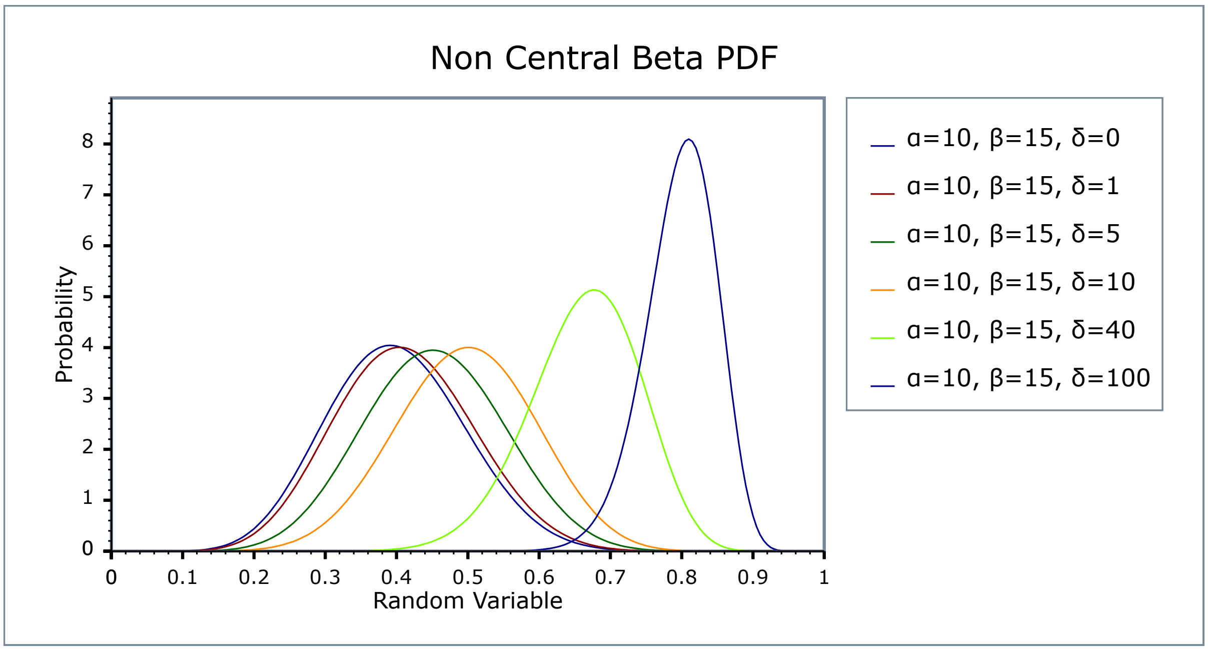
This gives a PDF that can be expressed as a Poisson mixture of beta distribution PDFs:

$$f(x;\,\alpha,\,\beta;\,\lambda) \quad = \quad \sum_{i=0}^{\infty} P\left(i;\,\frac{\lambda}{2}\right) I_x'(\alpha+i,\,\beta)$$

where $P(i;\lambda/2)$ is the discrete Poisson probablity at *i*, with mean $\lambda/2$, and $I_x'(\alpha, \beta)$ is the derivative of the incomplete beta function. This leads to the usual form of the CDF as:

$$F(x;\,\alpha,\,\beta;\,\lambda) \quad = \quad \sum_{i=0}^{\infty} P\left(i;\,\frac{\lambda}{2}\right) I_x(\alpha+i,\,\beta)$$

The following graph illustrates how the distribution changes for different values of λ:



Member Functions

```
non_central_beta_distribution(RealType a, RealType b, RealType lambda);
```

Constructs a noncentral beta distribution with shape parameters *a* and *b* and non-centrality parameter *lambda*.

Requires a > 0, b > 0 and lambda >= 0, otherwise calls domain_error.

```
RealType alpha()const;
```

Returns the parameter *a* from which this object was constructed.

```
RealType beta()const;
```

Returns the parameter *b* from which this object was constructed.

```
RealType non_centrality()const;
```

Returns the parameter *lambda* from which this object was constructed.

Non-member Accessors

Most of the usual non-member accessor functions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, mean, variance, standard deviation, median, mode, Hazard Function, Cumulative Hazard Function, range and support.

Mean and variance are implemented using hypergeometric pfq functions and relations given in Wolfram Noncentral Beta Distribution.

However, the following are not currently implemented: skewness, kurtosis and kurtosis_excess.

The domain of the random variable is [0, 1].

Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. The failures in the comparison to the R Math library, seem to be mostly in the corner cases when the probablity would be very small. Unless otherwise specified any floating-point type that is narrower than the one shown will have effectively zero error.

Table 18. Error rates for non central beta CDF

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Beta, medium parameters	Max = 0.998ε (Mean = 0.0649ε) (*Rmath 3.2.3:* Max = 1.46e+26ε (Mean = 3.5e+24ε) And other failures.)	Max = 824ε (Mean = 27.4ε)	Max = 832ε (Mean = 38.1ε)	Max = 242ε (Mean = 31ε)
Non Central Beta, large parameters	Max = 1.18ε (Mean = 0.175ε) (*Rmath 3.2.3:* Max = 1.01e+36ε (Mean = 1.19e+35ε) And other failures.)	Max = 2.5e+04ε (Mean = 3.78e+03ε)	Max = 2.57e+04ε (Mean = 4.45e+03ε)	Max = 3.66e+03ε (Mean = 500ε)

Table 19. Error rates for non central beta CDF complement

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Beta, medium parameters	Max = 0.998ε (Mean = 0.0936ε) (*Rmath 3.2.3:* Max = 7.5e+97ε (Mean = 1.37e+96ε) And other failures.)	Max = 396ε (Mean = 50.7ε)	Max = 554ε (Mean = 57.2ε)	Max = 624ε (Mean = 62.7ε)
Non Central Beta, large parameters	Max = 0.986ε (Mean = 0.188ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 6.83e+03ε (Mean = 993ε)	Max = 3.56e+03ε (Mean = 707ε)	Max = 1.25e+04ε (Mean = 1.49e+03ε)

Error rates for the PDF, the complement of the CDF and for the quantile functions are broadly similar.

Tests

There are two sets of test data used to verify this implementation: firstly we can compare with a few sample values generated by the R library. Secondly, we have tables of test data, computed with this implementation and using interval arithmetic - this data should be accurate to at least 50 decimal digits - and is the used for our accuracy tests.

Implementation

The CDF and its complement are evaluated as follows:

First we determine which of the two values (the CDF or its complement) is likely to be the smaller, the crossover point is taken to be the mean of the distribution: for this we use the approximation due to: R. Chattamvelli and R. Shanmugam, "Algorithm AS 310: Computing the Non-Central Beta Distribution Function", Applied Statistics, Vol. 46, No. 1. (1997), pp. 146-156.

$$E(X) \quad \simeq \quad 1 - \left(\frac{\beta}{C}\right)\left(1 + \frac{\lambda}{2C^2}\right) \quad ; \quad C = \alpha + \beta + \frac{\lambda}{2}$$

Then either the CDF or its complement is computed using the relations:

$$\begin{aligned} F(x; \alpha, \beta; \lambda) &= \sum_{i=0}^{\infty} P\left(i; \tfrac{\lambda}{2}\right) I_x(\alpha + i, \beta) \\ 1 - F(x; \alpha, \beta; \lambda) &= \sum_{i=0}^{\infty} P\left(i; \tfrac{\lambda}{2}\right)(1 - I_x(\alpha + i, \beta)) \end{aligned}$$

The summation is performed by starting at i = λ/2, and then recursing in both directions, using the usual recurrence relations for the Poisson PDF and incomplete beta functions. This is the "Method 2" described by:

Denise Benton and K. Krishnamoorthy, "Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of the square of the sample multiple correlation coefficient", Computational Statistics & Data Analysis 43 (2003) 249-267.

Specific applications of the above formulae to the noncentral beta distribution can be found in:

Russell V. Lenth, "Algorithm AS 226: Computing Noncentral Beta Probabilities", Applied Statistics, Vol. 36, No. 2. (1987), pp. 241-244.

H. Frick, "Algorithm AS R84: A Remark on Algorithm AS 226: Computing Non-Central Beta Probabilities", Applied Statistics, Vol. 39, No. 2. (1990), pp. 311-312.

Ming Long Lam, "Remark AS R95: A Remark on Algorithm AS 226: Computing Non-Central Beta Probabilities", Applied Statistics, Vol. 44, No. 4. (1995), pp. 551-552.

Harry O. Posten, "An Effective Algorithm for the Noncentral Beta Distribution Function", The American Statistician, Vol. 47, No. 2. (May, 1993), pp. 129-131.

R. Chattamvelli, "A Note on the Noncentral Beta Distribution Function", The American Statistician, Vol. 49, No. 2. (May, 1995), pp. 231-234.

Of these, the Posten reference provides the most complete overview, and includes the modification starting iteration at λ/2.

The main difference between this implementation and the above references is the direct computation of the complement when most efficient to do so, and the accumulation of the sum to -1 rather than subtracting the result from 1 at the end: this can substantially reduce the number of iterations required when the result is near 1.

The PDF is computed using the methodology of Benton and Krishnamoorthy and the relation:

$$f(x; \alpha, \beta; \lambda) \quad = \quad \sum_{i=0}^{\infty} P\left(i; \frac{\lambda}{2}\right) I_x'(\alpha + i, \beta)$$

Quantiles are computed using a specially modified version of bracket and solve, starting the search for the root at the mean of the distribution. (A Cornish-Fisher type expansion was also tried, but while this gets quite close to the root in many cases, when it is wrong it tends to introduce quite pathological behaviour: more investigation in this area is probably warranted).

Noncentral Chi-Squared Distribution

```
#include <boost/math/distributions/non_central_chi_squared.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class non_central_chi_squared_distribution;

typedef non_central_chi_squared_distribution<> non_central_chi_squared;

template <class RealType, class Policy>
class non_central_chi_squared_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor:
   non_central_chi_squared_distribution(RealType v, RealType lambda);

   // Accessor to degrees of freedom parameter v:
   RealType degrees_of_freedom()const;

   // Accessor to non centrality parameter lambda:
   RealType non_centrality()const;

   // Parameter finders:
   static RealType find_degrees_of_freedom(RealType lambda, RealType x, RealType p);
   template <class A, class B, class C>
   static RealType find_degrees_of_freedom(const complemented3_type<A,B,C>& c);

   static RealType find_non_centrality(RealType v, RealType x, RealType p);
   template <class A, class B, class C>
   static RealType find_non_centrality(const complemented3_type<A,B,C>& c);
};

}} // namespaces
```

The noncentral chi-squared distribution is a generalization of the Chi Squared Distribution. If X_i are ν independent, normally distributed random variables with means μ_i and variances σ_i^2, then the random variable

$$\sum_{i=1}^{k}\left(\frac{\chi_i}{\sigma_i}\right)^2$$

is distributed according to the noncentral chi-squared distribution.

The noncentral chi-squared distribution has two parameters: ν which specifies the number of degrees of freedom (i.e. the number of X_i), and λ which is related to the mean of the random variables X_i by:

$$\lambda = \sum_{i=1}^{k}\left(\frac{\mu_i}{\sigma_i}\right)^2$$

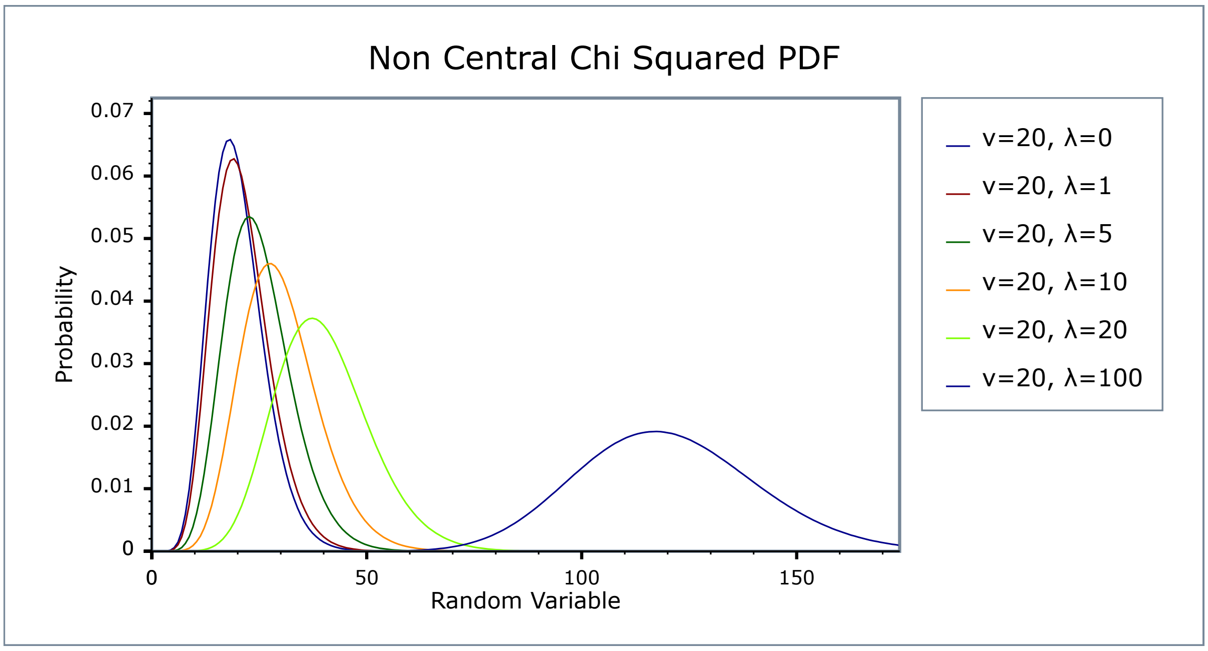
(Note that some references define λ as one half of the above sum).

This leads to a PDF of:

$$f(x;\, \nu,\, \lambda) \quad = \quad \sum_{i=0}^{\infty} \frac{e^{\left(-\frac{\lambda}{2}\right)}\left(\frac{\lambda}{2}\right)^{i}}{i\,!} f(x;\, \nu + 2i) \quad = \quad \frac{1}{2} e^{\left[-\frac{(x+\lambda)}{2}\right]}\left(\frac{x}{\lambda}\right)^{\left(\frac{\nu}{4}-\frac{1}{2}\right)} I_{\left(\frac{k}{2}-1\right)}\left(\sqrt{\lambda x}\right)$$

where *f(x;k)* is the central chi-squared distribution PDF, and $I_v(x)$ is a modified Bessel function of the first kind.

The following graph illustrates how the distribution changes for different values of λ:



Member Functions

```
non_central_chi_squared_distribution(RealType v, RealType lambda);
```

Constructs a Chi-Squared distribution with *v* degrees of freedom and non-centrality parameter *lambda*.

Requires v > 0 and lambda >= 0, otherwise calls domain_error.

```
RealType degrees_of_freedom()const;
```

Returns the parameter *v* from which this object was constructed.

```
RealType non_centrality()const;
```

Returns the parameter *lambda* from which this object was constructed.

```
static RealType find_degrees_of_freedom(RealType lambda, RealType x, RealType p);
```

This function returns the number of degrees of freedom *v* such that: `cdf(non_central_chi_squared<RealType, Policy>(v, lambda), x) == p`

```
template <class A, class B, class C>
static RealType find_degrees_of_freedom(const complemented3_type<A,B,C>& c);
```

When called with argument `boost::math::complement(lambda, x, q)` this function returns the number of degrees of freedom *v* such that:

`cdf(complement(non_central_chi_squared<RealType, Policy>(v, lambda), x)) == q`.

```
static RealType find_non_centrality(RealType v, RealType x, RealType p);
```

This function returns the non centrality parameter *lambda* such that:

`cdf(non_central_chi_squared<RealType, Policy>(v, lambda), x) == p`

```
template <class A, class B, class C>
static RealType find_non_centrality(const complemented3_type<A,B,C>& c);
```

When called with argument `boost::math::complement(v, x, q)` this function returns the non centrality parameter *lambda* such that:

`cdf(complement(non_central_chi_squared<RealType, Policy>(v, lambda), x)) == q`.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, +∞].

Examples

There is a worked example for the noncentral chi-squared distribution.

Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. The failures in the comparison to the R Math library, seem to be mostly in the corner cases when the probablity would be very small. Unless otherwise specified any floating-point type that is narrower than the one shown will have effectively zero error.

Table 20. Error rates for non central chi squared CDF

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Chi Squared, medium parameters	Max = 0.99ε (Mean = 0.0544ε) (*Rmath 3.2.3:* Max = 727ε (Mean = 121ε))	Max = 46.5ε (Mean = 10.3ε)	Max = 115ε (Mean = 13.9ε)	Max = 48.9ε (Mean = 10ε)
Non Central Chi Squared, large parameters	Max = 1.07ε (Mean = 0.102ε) (*Rmath 3.2.3:* Max = 3.27e+08ε (Mean = 2.23e+07ε))	Max = 3.07e+03ε (Mean = 336ε)	Max = 6.17e+03ε (Mean = 677ε)	Max = 9.79e+03ε (Mean = 723ε)

Table 21. Error rates for non central chi squared CDF complement

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Chi Squared, medium parameters	Max = 0.96ε (Mean = 0.0635ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 107ε (Mean = 17.2ε)	Max = 171ε (Mean = 22.8ε)	Max = 98.6ε (Mean = 15.8ε)
Non Central Chi Squared, large parameters	Max = 2.11ε (Mean = 0.278ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 5.02e+03ε (Mean = 630ε)	Max = 5.1e+03ε (Mean = 577ε)	Max = 5.43e+03ε (Mean = 705ε)

Error rates for the quantile functions are broadly similar. Special mention should go to the `mode` function: there is no closed form for this function, so it is evaluated numerically by finding the maxima of the PDF: in principal this can not produce an accuracy greater than the square root of the machine epsilon.

Tests

There are two sets of test data used to verify this implementation: firstly we can compare with published data, for example with Table 6 of "Self-Validating Computations of Probabilities for Selected Central and Noncentral Univariate Probability Functions", Morgan C. Wang and William J. Kennedy, Journal of the American Statistical Association, Vol. 89, No. 427. (Sep., 1994), pp. 878-887. Secondly, we have tables of test data, computed with this implementation and using interval arithmetic - this data should be accurate to at least 50 decimal digits - and is the used for our accuracy tests.

Implementation

The CDF and its complement are evaluated as follows:

First we determine which of the two values (the CDF or its complement) is likely to be the smaller: for this we can use the relation due to Temme (see "Asymptotic and Numerical Aspects of the Noncentral Chi-Square Distribution", N. M. Temme, Computers Math. Applic. Vol 25, No. 5, 55-63, 1993) that:

F(ν,λ;ν+λ) ≈ 0.5

and so compute the CDF when the random variable is less than ν+λ, and its complement when the random variable is greater than ν+λ. If necessary the computed result is then subtracted from 1 to give the desired result (the CDF or its complement).

For small values of the non centrality parameter, the CDF is computed using the method of Ding (see "Algorithm AS 275: Computing the Non-Central #2 Distribution Function", Cherng G. Ding, Applied Statistics, Vol. 41, No. 2. (1992), pp. 478-482). This uses the following series representation:

$$P(x;\, \nu,\, \lambda) \quad = \quad \sum_{i=0}^{\infty} s_i t_i$$

$$s_0 = u_0 = e^{-\frac{\lambda}{2}}, \qquad s_i = s_{i-1} + u_i, \qquad u_i = u_{i-1}\frac{\lambda}{2i}$$

$$t_0 = \frac{1}{\Gamma\left(\frac{\nu}{2}+1\right)}\left(\frac{x}{2}\right)^{\frac{\nu}{2}} e^{-\frac{x}{2}}, \qquad t_i = t_{i-1}\frac{x}{\nu+2i}$$

which requires just one call to gamma_p_derivative with the subsequent terms being computed by recursion as shown above.

For larger values of the non-centrality parameter, Ding's method can take an unreasonable number of terms before convergence is achieved. Furthermore, the largest term is not the first term, so in extreme cases the first term may be zero, leading to a zero result, even though the true value may be non-zero.

Therefore, when the non-centrality parameter is greater than 200, the method due to Krishnamoorthy (see "Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of the square of the sample multiple correlation coefficient", Denise Benton and K. Krishnamoorthy, Computational Statistics & Data Analysis, 43, (2003), 249-267) is used.

This method uses the well known sum:

$$P(x;\, \nu,\, \lambda) \quad = \quad \sum_{i=0}^{\infty} \frac{e^{-\frac{\lambda}{2}}\left(\frac{\lambda}{2}\right)^i}{i\,!} P_{\frac{x}{2}}\left(\frac{\nu}{2}+i\right)$$

Where $P_a(x)$ is the incomplete gamma function.

The method starts at the λth term, which is where the Poisson weighting function achieves its maximum value, although this is not necessarily the largest overall term. Subsequent terms are calculated via the normal recurrence relations for the incomplete gamma function, and iteration proceeds both forwards and backwards until sufficient precision has been achieved. It should be noted that recurrence in the forwards direction of $P_a(x)$ is numerically unstable. However, since we always start *after* the largest term in the series, numeric instability is introduced more slowly than the series converges.

Computation of the complement of the CDF uses an extension of Krishnamoorthy's method, given that:

$$1 - P(x;\, \nu,\, \lambda) \quad = \quad \sum_{i=0}^{\infty} \frac{e^{-\frac{\lambda}{2}}\left(\frac{\lambda}{2}\right)^i}{i\,!} Q_{\frac{x}{2}}\left(\frac{\nu}{2}+i\right)$$

we can again start at the λ'th term and proceed in both directions from there until the required precision is achieved. This time it is backwards recursion on the incomplete gamma function $Q_a(x)$ which is unstable. However, as long as we start well *before* the largest term, this is not an issue in practice.

The PDF is computed directly using the relation:

$$f(x;\, \nu,\, \lambda) \quad = \quad \sum_{i=0}^{\infty} \frac{e^{\left(-\frac{\lambda}{2}\right)}\left(\frac{\lambda}{2}\right)^i}{i\,!} f(x;\, \nu+2i) \quad = \quad \frac{1}{2} e^{\left[-\frac{(x+\lambda)}{2}\right]}\left(\frac{x}{\lambda}\right)^{\left(\frac{\nu}{4}-\frac{1}{2}\right)} I_{\left(\frac{k}{2}-1\right)}\left(\sqrt{\lambda x}\right)$$

Where *f(x; v)* is the PDF of the central Chi Squared Distribution and $I_v(x)$ is a modified Bessel function, see cyl_bessel_i. For small values of the non-centrality parameter the relation in terms of cyl_bessel_i is used. However, this method fails for large values of the non-centrality parameter, so in that case the infinite sum is evaluated using the method of Benton and Krishnamoorthy, and the usual recurrence relations for successive terms.

The quantile functions are computed by numeric inversion of the CDF. An improve starting quess is from Thomas Luu, Fast and accurate parallel computation of quantile functions for random number generation, Doctorial Thesis, 2016.

There is no closed form for the mode of the noncentral chi-squared distribution: it is computed numerically by finding the maximum of the PDF. Likewise, the median is computed numerically via the quantile.

The remaining non-member functions use the following formulas:

$$\begin{aligned}\text{mean} &= v+\lambda \\ \text{variance} &= 2(v+2\lambda) \\ \text{skewness} &= \frac{2^{\frac{3}{2}}(v+3\lambda)}{(v+2\lambda)^{\frac{3}{2}}}\end{aligned}$$

Some analytic properties of noncentral distributions (particularly unimodality, and monotonicity of their modes) are surveyed and summarized by:

Andrea van Aubel & Wolfgang Gawronski, Applied Mathematics and Computation, 141 (2003) 3-12.

Noncentral F Distribution

```
#include <boost/math/distributions/non_central_f.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class non_central_f_distribution;

typedef non_central_f_distribution<> non_central_f;

template <class RealType, class Policy>
class non_central_f_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor:
   non_central_f_distribution(RealType v1, RealType v2, RealType lambda);

   // Accessor to degrees_of_freedom parameters v1 & v2:
   RealType degrees_of_freedom1()const;
   RealType degrees_of_freedom2()const;

   // Accessor to non-centrality parameter lambda:
   RealType non_centrality()const;
};

}} // namespaces
```

The noncentral F distribution is a generalization of the Fisher F Distribution. It is defined as the ratio

```
F = (X/v1) / (Y/v2)
```

where X is a noncentral χ^2 random variable with *v1* degrees of freedom and non-centrality parameter λ, and Y is a central χ^2 random variable with *v2* degrees of freedom.

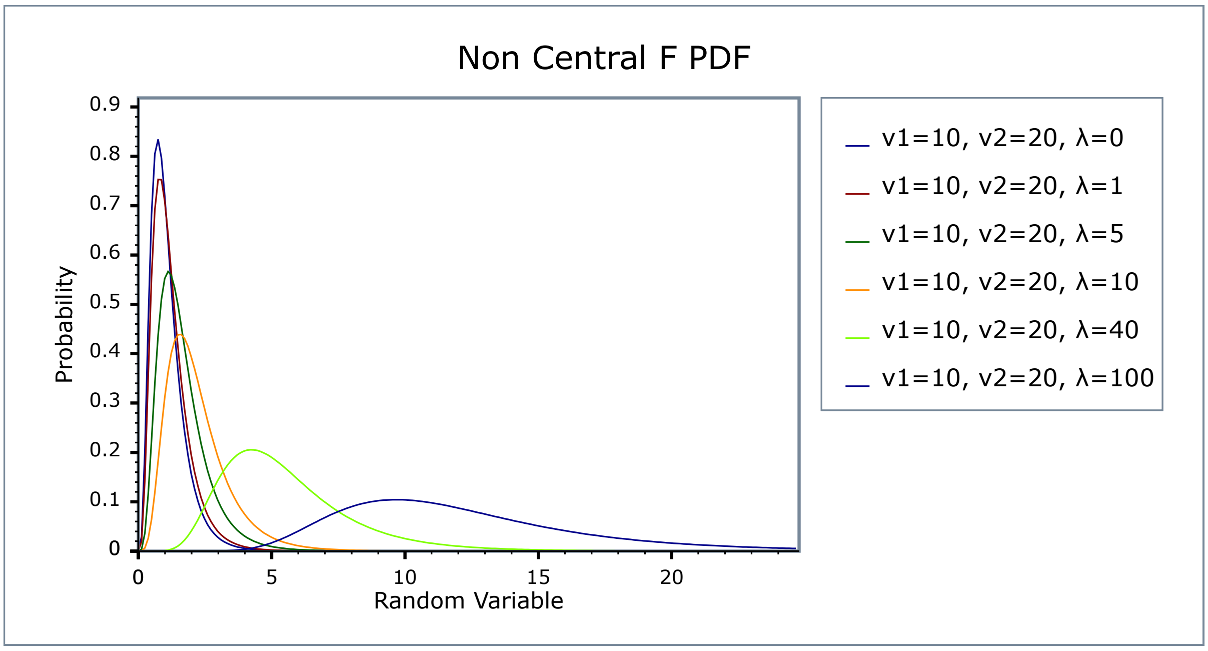
This gives the following PDF:

$$f(x;\, v_1,\, v_2;\, \lambda) \quad = \quad e^{-\frac{\lambda}{2}+\frac{(\lambda v_1 x)}{(2(v_2+v_1 x))}} v_1^{\frac{v_1}{2}} v_2^{\frac{v_2}{2}} x^{\frac{v_1}{2}-1} (v_2+v_1 x)^{-\frac{(v_1+v_2)}{2}} \frac{\Gamma\left(\frac{1}{2}v_1\right)\Gamma\left(1+\frac{1}{2}v_2\right)L_{\frac{v_2}{2}}^{\frac{v_1}{2}-1}\left(-\frac{\lambda v_1 x}{2(v_2+v_1 x)}\right)}{B\left(\frac{1}{2}v_1,\, \frac{1}{2}v_2\right)\Gamma\left(\frac{1}{2}(v_1+v_2)\right)}$$

where $L_a^b(c)$ is a generalised Laguerre polynomial and B(a,b) is the beta function, or

$$f(x;\, v_1,\, v_2;\, \lambda) \quad = \quad \sum_{k=0}^{\infty} \frac{e^{-\frac{\lambda}{2}}\left(\frac{\lambda}{2}\right)^k}{B\left(\frac{v_2}{2},\, \frac{v_1}{2}+k\right)k\,!} \left(\frac{v_1}{v_2}\right)^{\left(\frac{v_1}{2}+k\right)} \left(\frac{v_2}{v_2+v_1 x}\right)^{\left(\frac{v_1+v_2}{2}+k\right)} x^{\left(\frac{v_1}{2}+k-1\right)}$$

The following graph illustrates how the distribution changes for different values of λ:



Member Functions

```
non_central_f_distribution(RealType v1, RealType v2, RealType lambda);
```

Constructs a non-central beta distribution with parameters *v1* and *v2* and non-centrality parameter *lambda*.

Requires v1 > 0, v2 > 0 and lambda >= 0, otherwise calls domain_error.

```
RealType degrees_of_freedom1()const;
```

Returns the parameter *v1* from which this object was constructed.

```
RealType degrees_of_freedom2()const;
```

Returns the parameter *v2* from which this object was constructed.

```
RealType non_centrality()const;
```

Returns the non-centrality parameter *lambda* from which this object was constructed.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, +∞].

Accuracy

This distribution is implemented in terms of the Noncentral Beta Distribution: refer to that distribution for accuracy data.

Tests

Since this distribution is implemented by adapting another distribution, the tests consist of basic sanity checks computed by the R-2.5.1 Math library statistical package and its pbeta and dbeta functions.

Implementation

In the following table *v1* and *v2* are the first and second degrees of freedom parameters of the distribution, λ is the non-centrality parameter, *x* is the random variate, *p* is the probability, and *q = 1-p*.

Function	Implementation Notes
pdf	Implemented in terms of the non-central beta PDF using the relation: $f(x;v1,v2;\lambda) = (v1/v2) / ((1+y)*(1+y)) * g(y/(1+y);v1/2,v2/2;\lambda)$ where $g(x; a, b; \lambda)$ is the non central beta PDF, and: $y = x * v1 / v2$
cdf	Using the relation: $p = B_y(v1/2, v2/2; \lambda)$ where $B_x(a, b; \lambda)$ is the noncentral beta distribution CDF and $y = x * v1 / v2$
cdf complement	Using the relation: $q = 1 - B_y(v1/2, v2/2; \lambda)$ where $1 - B_x(a, b; \lambda)$ is the complement of the noncentral beta distribution CDF and $y = x * v1 / v2$
quantile	Using the relation: $x = (bx / (1-bx)) * (v1 / v2)$ where $bx = Q_p^{-1}(v1/2, v2/2; \lambda)$ and $Q_p^{-1}(v1/2, v2/2; \lambda)$ is the noncentral beta quantile.
quantile from the complement	Using the relation: $x = (bx / (1-bx)) * (v1 / v2)$ where $bx = QC_q^{-1}(v1/2, v2/2; \lambda)$ and $QC_q^{-1}(v1/2, v2/2; \lambda)$ is the noncentral beta quantile from the complement.
mean	$v2 * (v1 + l) / (v1 * (v2 - 2))$
mode	By numeric maximalisation of the PDF.

Function	Implementation Notes
variance	Refer to, Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld--A Wolfram Web Resource.
skewness	Refer to, Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld--A Wolfram Web Resource., and to the Mathematica documentation
kurtosis and kurtosis excess	Refer to, Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld--A Wolfram Web Resource., and to the Mathematica documentation

Some analytic properties of noncentral distributions (particularly unimodality, and monotonicity of their modes) are surveyed and summarized by:

Andrea van Aubel & Wolfgang Gawronski, Applied Mathematics and Computation, 141 (2003) 3-12.

Noncentral T Distribution

```
#include <boost/math/distributions/non_central_t.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class non_central_t_distribution;

typedef non_central_t_distribution<> non_central_t;

template <class RealType, class Policy>
class non_central_t_distribution
{
public:
   typedef RealType  value_type;
   typedef Policy    policy_type;

   // Constructor:
   non_central_t_distribution(RealType v, RealType delta);

   // Accessor to degrees_of_freedom parameter v:
   RealType degrees_of_freedom()const;

   // Accessor to non-centrality parameter delta:
   RealType non_centrality()const;
};

}} // namespaces
```

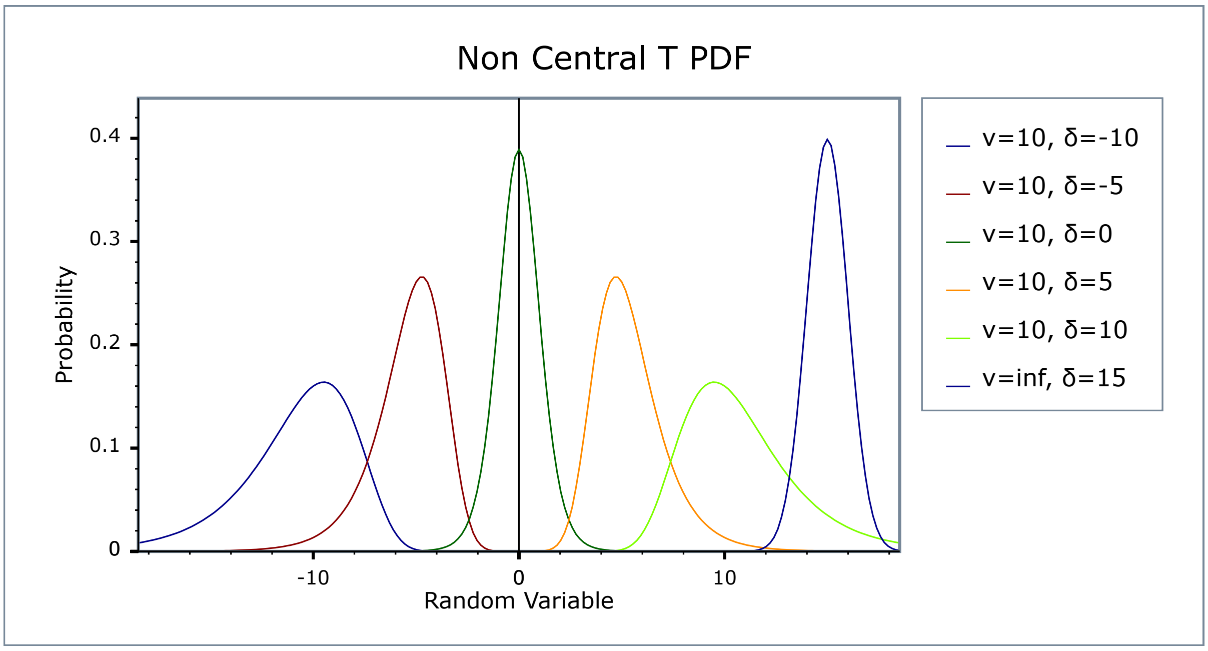
The noncentral T distribution is a generalization of the Students t Distribution. Let X have a normal distribution with mean δ and variance 1, and let ν S^2 have a chi-squared distribution with degrees of freedom ν. Assume that X and S^2 are independent. The distribution of $t_\nu(\delta)=X/S$ is called a noncentral t distribution with degrees of freedom ν and noncentrality parameter δ.

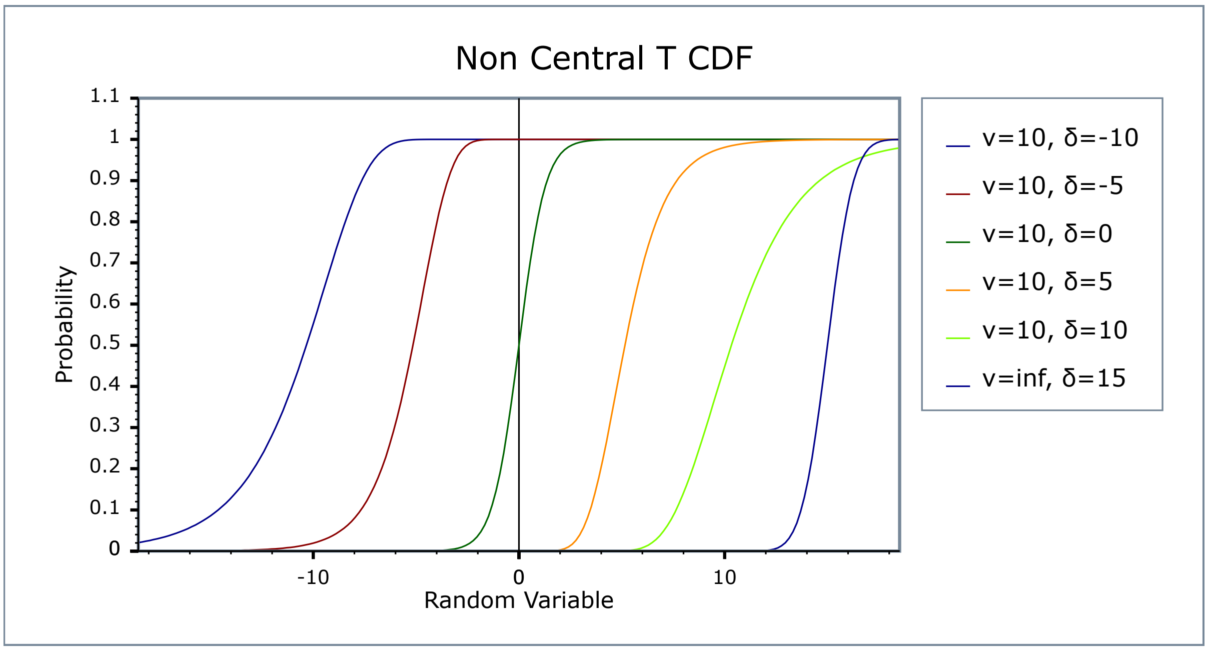
This gives the following PDF:

$$f(x; v; \delta) = \frac{v^{\frac{v}{2}} v!}{2^{v} e^{\frac{\delta^2}{2}} (v + x^2)^{\frac{v}{2}} \Gamma\left(\frac{v}{2}\right)} \left(\frac{\sqrt{2}\, \delta x\, {}_1F_1\left(\frac{n}{2} + 1;\ \frac{3}{2};\ \frac{\delta^2 x^2}{2(n + x^2)}\right)}{(v + x^2)\Gamma\left(\frac{1}{2}(v + 1)\right)} + \frac{{}_1F_1\left(\frac{1}{2}(n + 1);\ \frac{1}{2};\ \frac{\delta^2 x^2}{2(v + x^2)}\right)}{\sqrt{v + x^2}\,\Gamma\left(\frac{v}{2} + 1\right)} \right)$$

where ${}_1F_1$(a;b;x) is a confluent hypergeometric function.

The following graph illustrates how the distribution changes for different values of ν and δ:





Member Functions

```
non_central_t_distribution(RealType v, RealType delta);
```

Constructs a non-central t distribution with degrees of freedom parameter *v* and non-centrality parameter *delta*.

Requires *v* > 0 (including positive infinity) and finite *delta*, otherwise calls domain_error.

```
RealType degrees_of_freedom()const;
```

Returns the parameter *v* from which this object was constructed.

```
RealType non_centrality()const;
```

Returns the non-centrality parameter *delta* from which this object was constructed.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [-∞, +∞].

Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating-point types. Unless otherwise specified, any floating-point type that is narrower than the one shown will have effectively zero error.

Table 22. Error rates for non central t CDF

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central T	Max = 0.796ε (Mean = 0.0691ε) (*Rmath 3.2.3:* Max = 5.28e+15ε (Mean = 8.49e+14ε) And other failures.)	Max = 139ε (Mean = 31ε)	Max = 145ε (Mean = 30.9ε)	Max = 135ε (Mean = 32.1ε)
Non Central T (small non-centrality)	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 2.09e+03ε (Mean = 244ε))	Max = 3.86ε (Mean = 1.4ε)	Max = 9.15ε (Mean = 2.13ε)	Max = 6.17ε (Mean = 1.45ε)
Non Central T (large parameters)	Max = 257ε (Mean = 72.1ε) (*Rmath 3.2.3:* Max = 2.46ε (Mean = 0.657ε))	Max = 5.26e+05ε (Mean = 1.48e+05ε)	Max = 5.24e+05ε (Mean = 1.47e+05ε)	Max = 286ε (Mean = 62.8ε)

Table 23. Error rates for non central t CDF complement

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Non Central T	Max = 0.707ε (Mean = 0.0497ε) (*Rmath 3.2.3:* Max = 6.19e+15ε (Mean = 6.72e+14ε) And other failures.)	Max = 201ε (Mean = 31.7ε)	Max = 340ε (Mean = 43.2ε)	Max = 154ε (Mean = 32.1ε)
Non Central T (small non-centrality)	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 1.87e+03ε (Mean = 263ε))	Max = 10.5ε (Mean = 2.13ε)	Max = 10.5ε (Mean = 2.39ε)	Max = 4.6ε (Mean = 1.63ε)
Non Central T (large parameters)	Max = 478ε (Mean = 96.3ε) (*Rmath 3.2.3:* Max = 2.24ε (Mean = 0.945ε))	Max = 9.79e+05ε (Mean = 1.97e+05ε)	Max = 9.79e+05ε (Mean = 1.97e+05ε)	Max = 227ε (Mean = 50.4ε)

Caution

The complexity of the current algorithm is dependent upon δ^2: consequently the time taken to evaluate the CDF increases rapidly for $\delta > 500$, likewise the accuracy decreases rapidly for very large δ.

Accuracy for the quantile and PDF functions should be broadly similar. The *mode* is determined numerically and cannot in principal be more accurate than the square root of floating-point type FPT epsilon, accessed using `boost::math::tools::epsilon<FPT>()`. For 64-bit `double`, epsilon is about 1e-16, so the fractional accuracy is limited to 1e-8.

Tests

There are two sets of tests of this distribution:

Basic sanity checks compare this implementation to the test values given in "Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of the square of the sample multiple correlation coefficient." Denise Benton, K. Krishnamoorthy, Computational Statistics & Data Analysis 43 (2003) 249-267.

Accuracy checks use test data computed with this implementation and arbitary precision interval arithmetic: this test data is believed to be accurate to at least 50 decimal places.

The cases of large (or infinite) ν and/or large δ has received special treatment to avoid catastrophic loss of accuracy. New tests have been added to confirm the improvement achieved.

From Boost 1.52, degrees of freedom ν can be +∞ when the normal distribution located at δ (equivalent to the central Student's t distribution) is used in place for accuracy and speed.

Implementation

The CDF is computed using a modification of the method described in "Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of the square of the sample multiple correlation coefficient." Denise Benton, K. Krishnamoorthy, Computational Statistics & Data Analysis 43 (2003) 249-267.

This uses the following formula for the CDF:

$$P(t;\, v;\, \delta) \quad = \quad \Phi(-\delta) + \tfrac{1}{2}\sum\nolimits_{i=0}^{\infty}\left(P_i I_x\left(i+\tfrac{1}{2},\, \tfrac{v}{2}\right) + \tfrac{\delta}{\sqrt{2}} Q_i I_x\left(i+1, \tfrac{v}{2}\right)\right)$$

$$P_i = e^{-\frac{\delta^2}{2}} \frac{\left(\frac{\delta^2}{2}\right)^i}{i!} \quad , \quad Q_i = e^{-\frac{\delta^2}{2}} \frac{\left(\frac{\delta^2}{2}\right)^i}{\Gamma\left(i+\frac{3}{2}\right)} \quad , \quad x = \frac{t^2}{\left(v+t^2\right)}$$

Where $I_x(a,b)$ is the incomplete beta function, and Φ(x) is the normal CDF at x.

Iteration starts at the largest of the Poisson weighting terms (at i = $\delta^2 / 2$) and then proceeds in both directions as per Benton and Krishnamoorthy's paper.

Alternatively, by considering what happens when t = ∞, we have x = 1, and therefore $I_x(a,b) = 1$ and:

$$P(\infty;\, v;\, \delta) = 1 = \Phi(-\delta) + \frac{1}{2}\sum_{i=0}^{\infty}\left(P_i + \frac{\delta}{\sqrt{2}} Q_i\right)$$

From this we can easily show that:

$$1 - P(t;\, v;\, \delta) \quad = \quad \frac{1}{2}\sum_{i=0}^{\infty}\left(P_i I_y\left(\frac{v}{2},\, i+\frac{1}{2}\right) + \frac{\delta}{\sqrt{2}} Q_i I_y\left(\frac{v}{2},\, i+1\right)\right) \quad ; \quad y = 1 - x = \frac{v}{v+t^2}$$

and therefore we have a means to compute either the probability or its complement directly without the risk of cancellation error. The crossover criterion for choosing whether to calculate the CDF or its complement is the same as for the Noncentral Beta Distribution.

The PDF can be computed by a very similar method using:

$$f(t;\, v;\, \delta) \quad = \quad \frac{vt}{v^2 + 2vt^2 + t^4}\sum_{i=0}^{\infty}\left(P_i I_x'\left(i+\frac{1}{2},\, \frac{v}{2}\right) + \frac{\delta}{\sqrt{2}} Q_i I_x'\left(i+1, \frac{v}{2}\right)\right)$$

Where $I_x{}'(a,b)$ is the derivative of the incomplete beta function.

For both the PDF and CDF we switch to approximating the distribution by a Student's t distribution centred on δ when ν is very large. The crossover location appears to be when δ/(4ν) < ε, this location was estimated by inspection of equation 2.6 in "A Comparison of Approximations To Percentiles of the Noncentral t-Distribution". H. Sahai and M. M. Ojeda, Revista Investigacion Operacional Vol 21, No 2, 2000, page 123.

Equation 2.6 is a Fisher-Cornish expansion by Eeden and Johnson. The second term includes the ratio δ/(4ν), so when this term become negligible, this and following terms can be ignored, leaving just Student's t distribution centred on δ.

This was also confirmed by experimental testing.

See also

- "Some Approximations to the Percentage Points of the Noncentral t-Distribution". C. van Eeden. International Statistical Review, 29, 4-31.
- "Continuous Univariate Distributions". N.L. Johnson, S. Kotz and N. Balkrishnan. 1995. John Wiley and Sons New York.

The quantile is calculated via the usual root-finding without derivatives method with the initial guess taken as the quantile of a normal approximation to the noncentral T.

There is no closed form for the mode, so this is computed via functional maximisation of the PDF.

The remaining functions (mean, variance etc) are implemented using the formulas given in Weisstein, Eric W. "Noncentral Student's t-Distribution." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralStudentst-Distribution.html and in the Mathematica documentation.

Some analytic properties of noncentral distributions (particularly unimodality, and monotonicity of their modes) are surveyed and summarized by:

Andrea van Aubel & Wolfgang Gawronski, Applied Mathematics and Computation, 141 (2003) 3-12.

Normal (Gaussian) Distribution

```
#include <boost/math/distributions/normal.hpp>
```

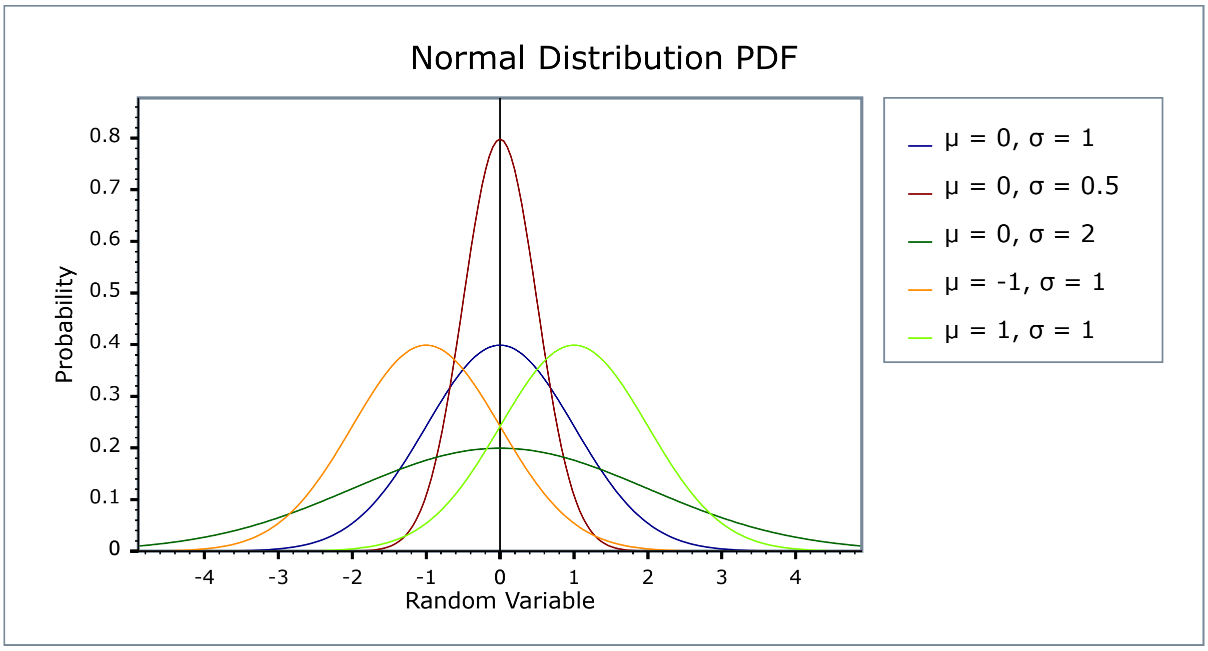
```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class normal_distribution;

typedef normal_distribution<> normal;

template <class RealType, class Policy>
class normal_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   normal_distribution(RealType mean = 0, RealType sd = 1);
   // Accessors:
   RealType mean()const; // location.
   RealType standard_deviation()const; // scale.
   // Synonyms, provided to allow generic use of find_location and find_scale.
   RealType location()const;
   RealType scale()const;
};

}} // namespaces
```

The normal distribution is probably the most well known statistical distribution: it is also known as the Gaussian Distribution. A normal distribution with mean zero and standard deviation one is known as the *Standard Normal Distribution*.

Given mean μ and standard deviation σ it has the PDF:

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

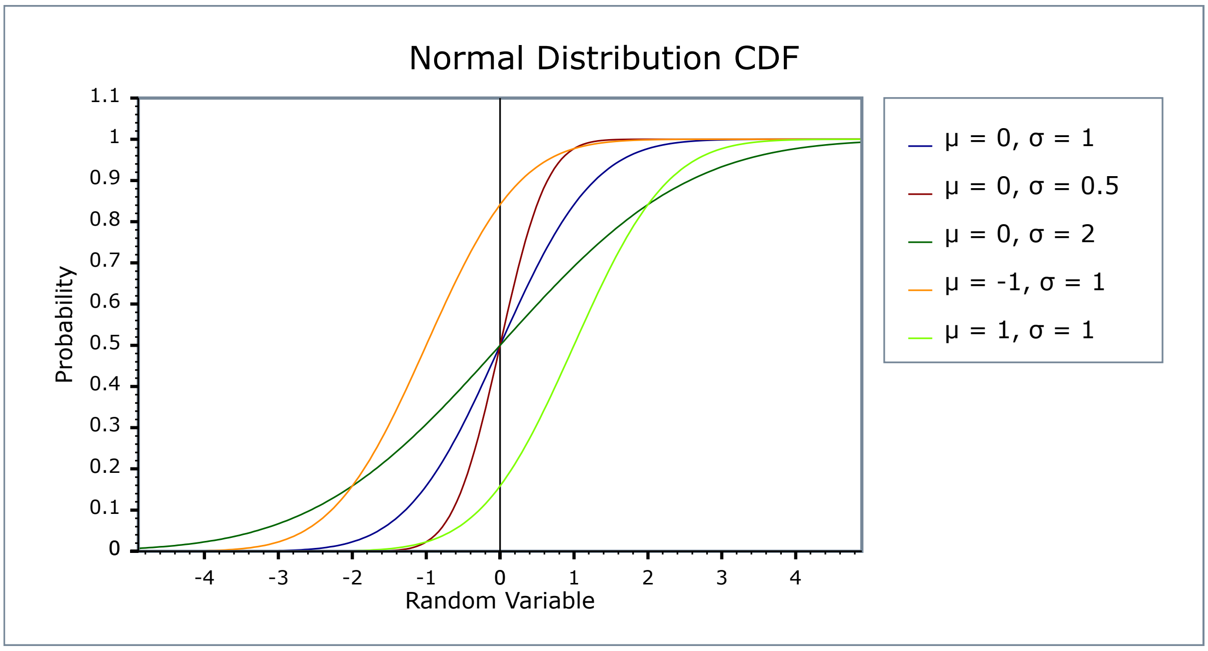
The variation the PDF with its parameters is illustrated in the following graph:



The cumulative distribution function is given by

$$\frac{1}{2}\left\{1 + erf\left(\frac{(x-\mu)}{\sqrt{(2\sigma^2)}}\right)\right]$$

and illustrated by this graph



Member Functions

```
normal_distribution(RealType mean = 0, RealType sd = 1);
```

Constructs a normal distribution with mean *mean* and standard deviation *sd*.

Requires sd > 0, otherwise domain_error is called.

```
RealType mean()const;
RealType location()const;
```

both return the *mean* of this distribution.

```
RealType standard_deviation()const;
RealType scale()const;
```

both return the *standard deviation* of this distribution. (Redundant location and scale function are provided to match other similar distributions, allowing the functions find_location and find_scale to be used generically).

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [-[max_value], +[min_value]]. However, the pdf of +∞ and -∞ = 0 is also supported, and cdf at -∞ = 0, cdf at +∞ = 1, and complement cdf -∞ = 1 and +∞ = 0, if RealType permits.

Accuracy

The normal distribution is implemented in terms of the error function, and as such should have very low error rates.

Implementation

In the following table *m* is the mean of the distribution, and *s* is its standard deviation.

Function	Implementation Notes
pdf	Using the relation: pdf = $e^{-(x-m)^2/(2s^2)}$ / (s * sqrt(2*pi))
cdf	Using the relation: p = 0.5 * erfc(-(x-m)/(s*sqrt(2)))
cdf complement	Using the relation: q = 0.5 * erfc((x-m)/(s*sqrt(2)))
quantile	Using the relation: x = m - s * sqrt(2) * erfc_inv(2*p)
quantile from the complement	Using the relation: x = m + s * sqrt(2) * erfc_inv(2*p)
mean and standard deviation	The same as `dist.mean()` and `dist.standard_deviation()`
mode	The same as the mean.
median	The same as the mean.
skewness	0
kurtosis	3
kurtosis excess	0

Pareto Distribution

```
#include <boost/math/distributions/pareto.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class pareto_distribution;

typedef pareto_distribution<> pareto;

template <class RealType, class Policy>
class pareto_distribution
{
public:
   typedef RealType value_type;
   // Constructor:
   pareto_distribution(RealType scale = 1, RealType shape = 1)
   // Accessors:
   RealType scale()const;
   RealType shape()const;
};

}} // namespaces
```

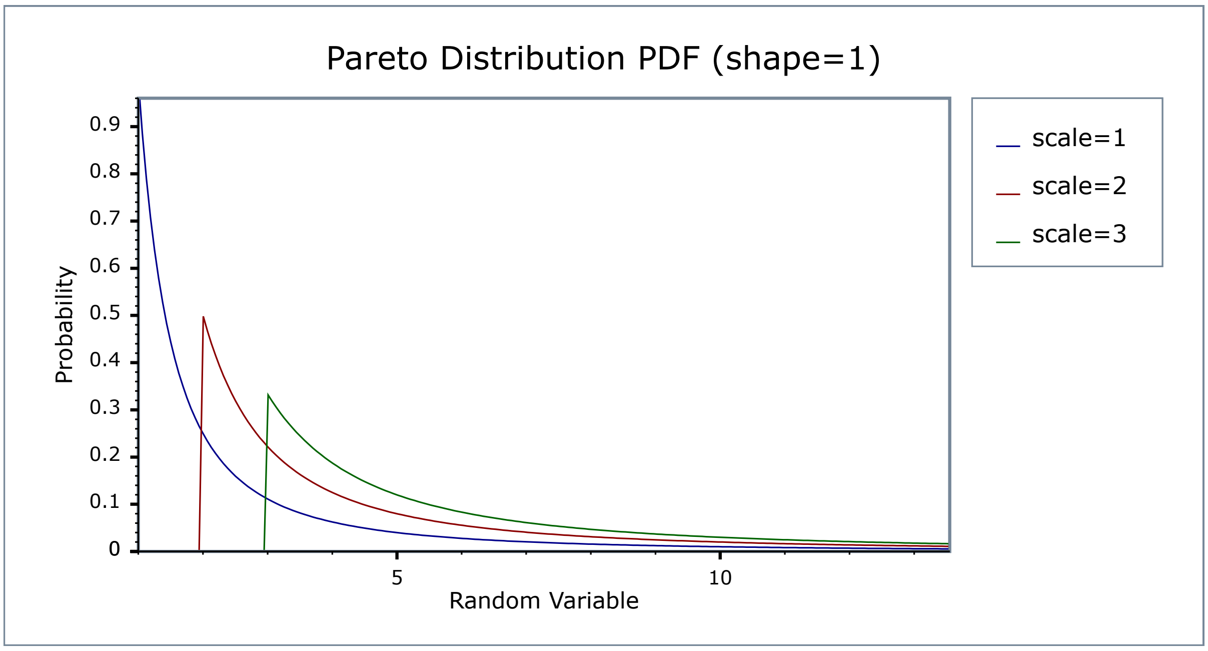
The Pareto distribution is a continuous distribution with the probability density function (pdf):

$f(x; \alpha, \beta) = \alpha\beta^{\alpha} / x^{\alpha+1}$

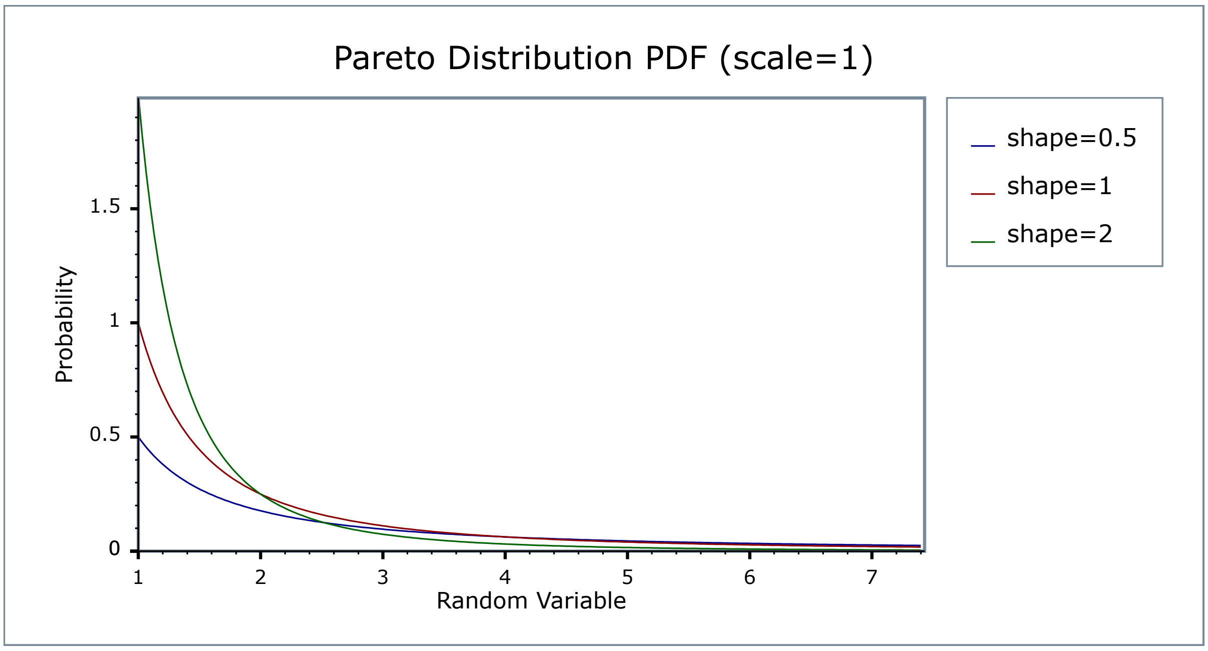
For shape parameter $\alpha > 0$, and scale parameter $\beta > 0$. If $x < \beta$, the pdf is zero.

The Pareto distribution often describes the larger compared to the smaller. A classic example is that 80% of the wealth is owned by 20% of the population.

The following graph illustrates how the PDF varies with the scale parameter β:



And this graph illustrates how the PDF varies with the shape parameter α:



Related distributions

Member Functions

```
pareto_distribution(RealType scale = 1, RealType shape = 1);
```

Constructs a pareto distribution with shape *shape* and scale *scale*.

Requires that the *shape* and *scale* parameters are both greater than zero, otherwise calls domain_error.

```
RealType scale()const;
```

Returns the *scale* parameter of this distribution.

```
RealType shape()const;
```

Returns the *shape* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The supported domain of the random variable is [scale, ∞].

Accuracy

The Pareto distribution is implemented in terms of the standard library `exp` functions plus expm1 and so should have very small errors, usually only a few epsilon.

If probability is near to unity (or the complement of a probability near zero) see also why complements?.

Implementation

In the following table α is the shape parameter of the distribution, and β is its scale parameter, *x* is the random variate, *p* is the probability and its complement *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf p = $\alpha\beta^{\alpha}/x^{\alpha+1}$
cdf	Using the relation: cdf p = 1 - $(\beta / x)^{\alpha}$
cdf complement	Using the relation: q = 1 - p = $-(\beta / x)^{\alpha}$
quantile	Using the relation: x = $\beta / (1 - p)^{1/\alpha}$
quantile from the complement	Using the relation: x = $\beta / (q)^{1/\alpha}$
mean	$\alpha\beta / (\beta - 1)$
variance	$\beta\alpha^2 / (\beta - 1)^2 (\beta - 2)$
mode	α
skewness	Refer to Weisstein, Eric W. "Pareto Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis	Refer to Weisstein, Eric W. "Pareto Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis excess	Refer to Weisstein, Eric W. "pareto Distribution." From MathWorld--A Wolfram Web Resource.

References

- Pareto Distribution
- Weisstein, Eric W. "Pareto Distribution." From MathWorld--A Wolfram Web Resource.
- Handbook of Statistical Distributions with Applications, K Krishnamoorthy, ISBN 1-58488-635-8, Chapter 23, pp 257 - 267. (Note the meaning of a and b is reversed in Wolfram and Krishnamoorthy).

Poisson Distribution

```
#include <boost/math/distributions/poisson.hpp>
```

```
namespace boost { namespace math {

template <class RealType = double,
          class Policy   = policies::policy<> >
class poisson_distribution;

typedef poisson_distribution<> poisson;

template <class RealType, class Policy>
class poisson_distribution
{
public:
  typedef RealType value_type;
  typedef Policy   policy_type;

  poisson_distribution(RealType mean = 1); // Constructor.
  RealType mean()const; // Accessor.
}

}} // namespaces boost::math
```

The Poisson distribution is a well-known statistical discrete distribution. It expresses the probability of a number of events (or failures, arrivals, occurrences ...) occurring in a fixed period of time, provided these events occur with a known mean rate λ (events/time), and are independent of the time since the last event.

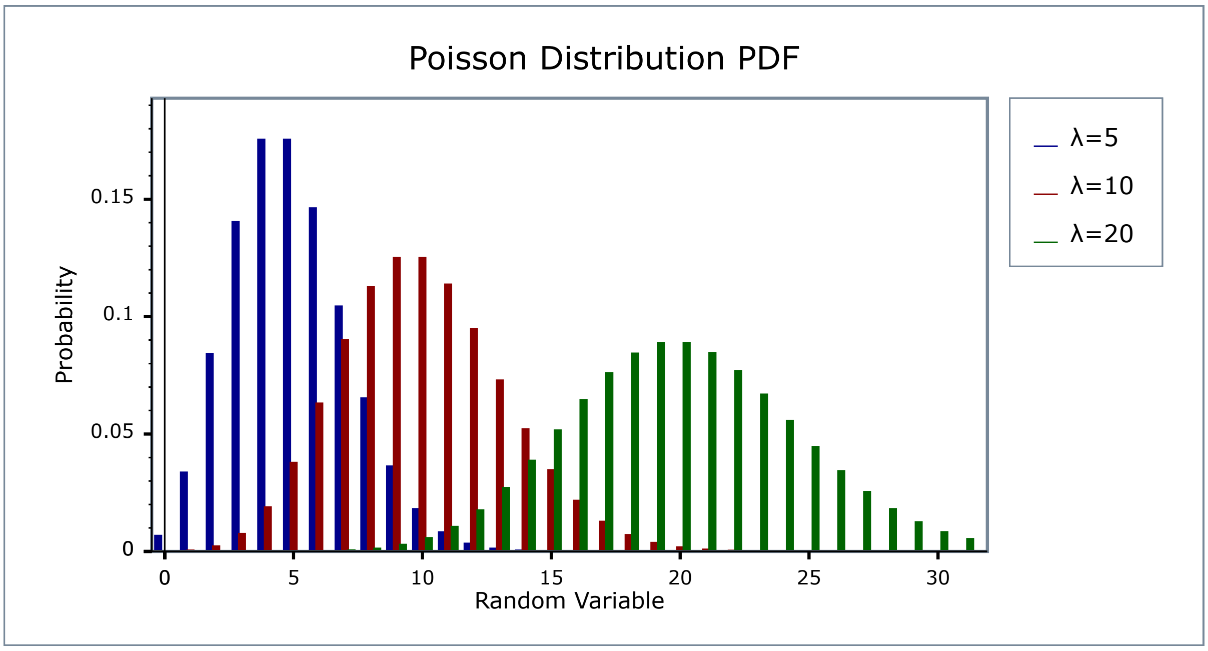
The distribution was discovered by Simé on-Denis Poisson (1781 to 1840).

It has the Probability Mass Function:

$$\mathrm{f}\,(k;\,\lambda) \quad = \quad \frac{e^{-\lambda}\lambda^{k}}{k\,!}$$

for k events, with an expected number of events λ.

The following graph illustrates how the PDF varies with the parameter λ:



Caution

The Poisson distribution is a discrete distribution: internally, functions like the `cdf` and `pdf` are treated "as if" they are continuous functions, but in reality the results returned from these functions only have meaning if an integer value is provided for the random variate argument.

The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails.

This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on the Poisson distribution. The reference docs describe how to change the rounding policy for these distributions.

Member Functions

```
poisson_distribution(RealType mean = 1);
```

Constructs a poisson distribution with mean *mean*.

```
RealType mean()const;
```

Returns the *mean* of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, ∞].

Accuracy

The Poisson distribution is implemented in terms of the incomplete gamma functions gamma_p and gamma_q and as such should have low error rates: but refer to the documentation of those functions for more information. The quantile and its complement use the inverse gamma functions and are therefore probably slightly less accurate: this is because the inverse gamma functions are implemented using an iterative method with a lower tolerance to avoid excessive computation.

Implementation

In the following table λ is the mean of the distribution, *k* is the random variable, *p* is the probability and $q = 1-p$.

Function	**Implementation Notes**
pdf	Using the relation: pdf = $e^{-\lambda} \lambda^k$ / k!
cdf	Using the relation: p = Γ(k+1, λ) / k! = gamma_q(k+1, λ)
cdf complement	Using the relation: q = gamma_p(k+1, λ)
quantile	Using the relation: k = gamma_q_inva(λ, p) - 1
quantile from the complement	Using the relation: k = gamma_p_inva(λ, q) - 1
mean	λ
mode	floor (λ) or λ
skewness	1/√λ
kurtosis	3 + 1/λ
kurtosis excess	1/λ

Rayleigh Distribution

```
#include <boost/math/distributions/rayleigh.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class rayleigh_distribution;

typedef rayleigh_distribution<> rayleigh;

template <class RealType, class Policy>
class rayleigh_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   rayleigh_distribution(RealType sigma = 1)
   // Accessors:
   RealType sigma()const;
};

}} // namespaces
```

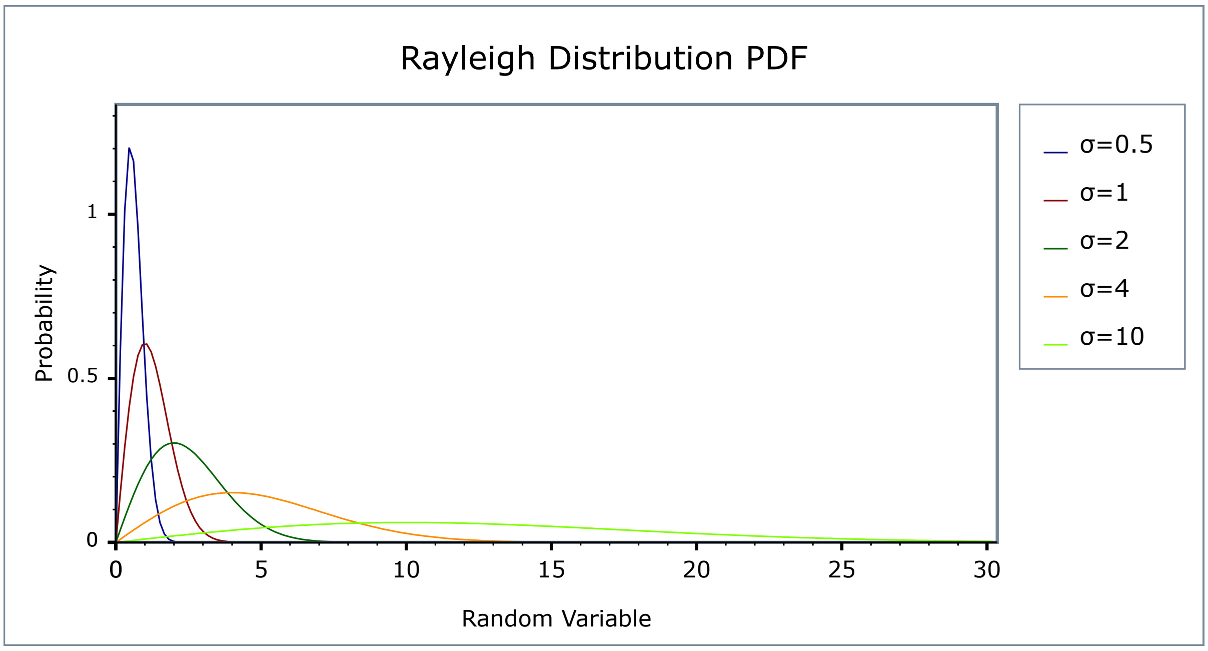
The Rayleigh distribution is a continuous distribution with the probability density function:

f(x; sigma) = x * exp($-x^2$/2 σ^2) / σ^2

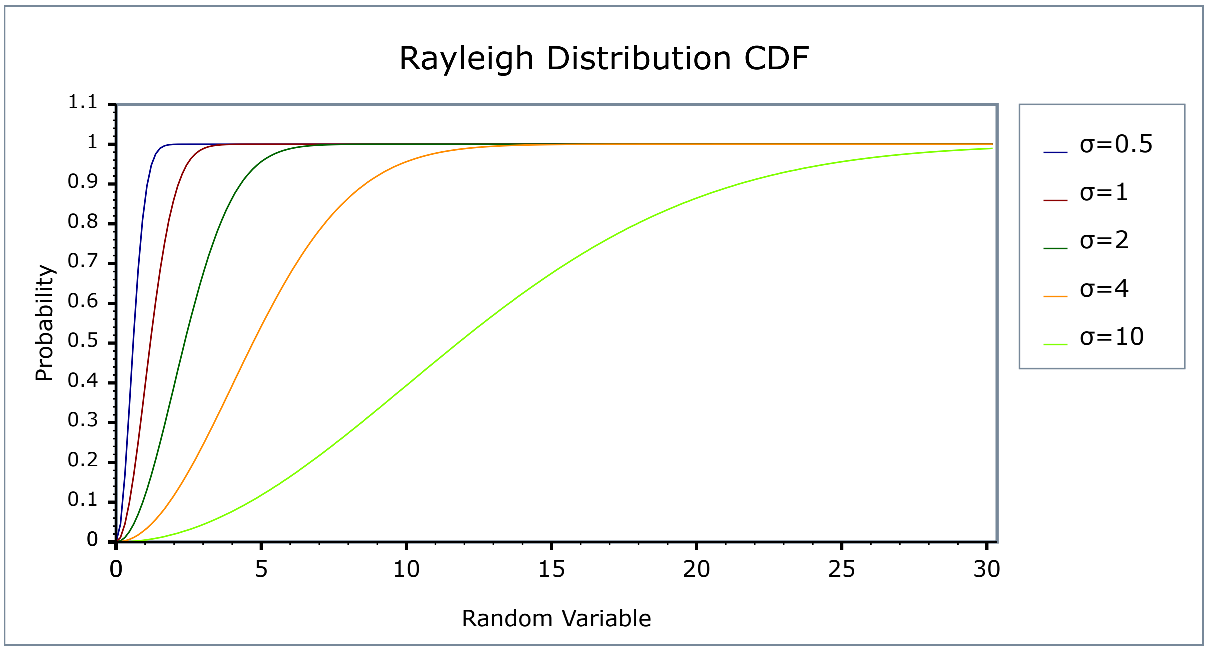
For sigma parameter σ > 0, and x > 0.

The Rayleigh distribution is often used where two orthogonal components have an absolute value, for example, wind velocity and direction may be combined to yield a wind speed, or real and imaginary components may have absolute values that are Rayleigh distributed.

The following graph illustrates how the Probability density Function(pdf) varies with the shape parameter σ:



and the Cumulative Distribution Function (cdf)



Related distributions

The absolute value of two independent normal distributions X and Y, √ (X^2 + Y^2) is a Rayleigh distribution.

The Chi, Rice and Weibull distributions are generalizations of the Rayleigh distribution.

Member Functions

```
rayleigh_distribution(RealType sigma = 1);
```

Constructs a Rayleigh distribution with σ *sigma*.

Requires that the σ parameter is greater than zero, otherwise calls domain_error.

```
RealType sigma()const;
```

Returns the *sigma* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, max_value].

Accuracy

The Rayleigh distribution is implemented in terms of the standard library `sqrt` and `exp` and as such should have very low error rates. Some constants such as skewness and kurtosis were calculated using NTL RR type with 150-bit accuracy, about 50 decimal digits.

Implementation

In the following table σ is the sigma parameter of the distribution, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = $x * \exp(-x^2)/2\ \sigma^2$
cdf	Using the relation: p = $1 - \exp(-x^2/2)\ \sigma^2 = -$expm1$(-x^2/2)\ \sigma^2$
cdf complement	Using the relation: q = $\exp(-x^2/ 2) * \sigma^2$
quantile	Using the relation: x = $\mathrm{sqrt}(-2 * \sigma^2) * \log(1 - p)) = \mathrm{sqrt}(-2 * \sigma^2) *$ log1p(-p))
quantile from the complement	Using the relation: x = $\mathrm{sqrt}(-2 * \sigma^2) * \log(q))$
mean	$\sigma * \mathrm{sqrt}(\pi/2)$
variance	$\sigma^2 * (4 - \pi/2)$
mode	σ
skewness	Constant from Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis	Constant from Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis excess	Constant from Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.

References

- http://en.wikipedia.org/wiki/Rayleigh_distribution
- Weisstein, Eric W. "Rayleigh Distribution." From MathWorld--A Wolfram Web Resource.

Skew Normal Distribution

```
#include <boost/math/distributions/skew_normal.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class skew_normal_distribution;

typedef skew_normal_distribution<> normal;

template <class RealType, class Policy>
class skew_normal_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Constructor:
   skew_normal_distribution(RealType location = 0, RealType scale = 1, RealType shape = 0);
   // Accessors:
   RealType location()const; // mean if normal.
   RealType scale()const; // width, standard deviation if normal.
  RealType shape()const; // The distribution is right skewed if shape > 0 and is left skewed ↵
if shape < 0.
                          // The distribution is normal if shape is zero.
};

}} // namespaces
```

The skew normal distribution is a variant of the most well known Gaussian statistical distribution.

The skew normal distribution with shape zero resembles the Normal Distribution, hence the latter can be regarded as a special case of the more generic skew normal distribution.

If the standard (mean = 0, scale = 1) normal distribution probability density function is

$$\varphi(x) = \frac{1}{\sqrt{(2\pi)}} e^{-\frac{x^2}{2}}$$

and the cumulative distribution function

$$\Phi(x) = \int_{-\infty}^{x} \varphi(t)dt \quad = \quad \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right]$$

then the PDF of the skew normal distribution with shape parameter α, defined by O'Hagan and Leonhard (1976) is

$$f(x) = 2\varphi(x)\Phi(\alpha x)$$

Given location ξ, scale ω, and shape α, it can be transformed, to the form:

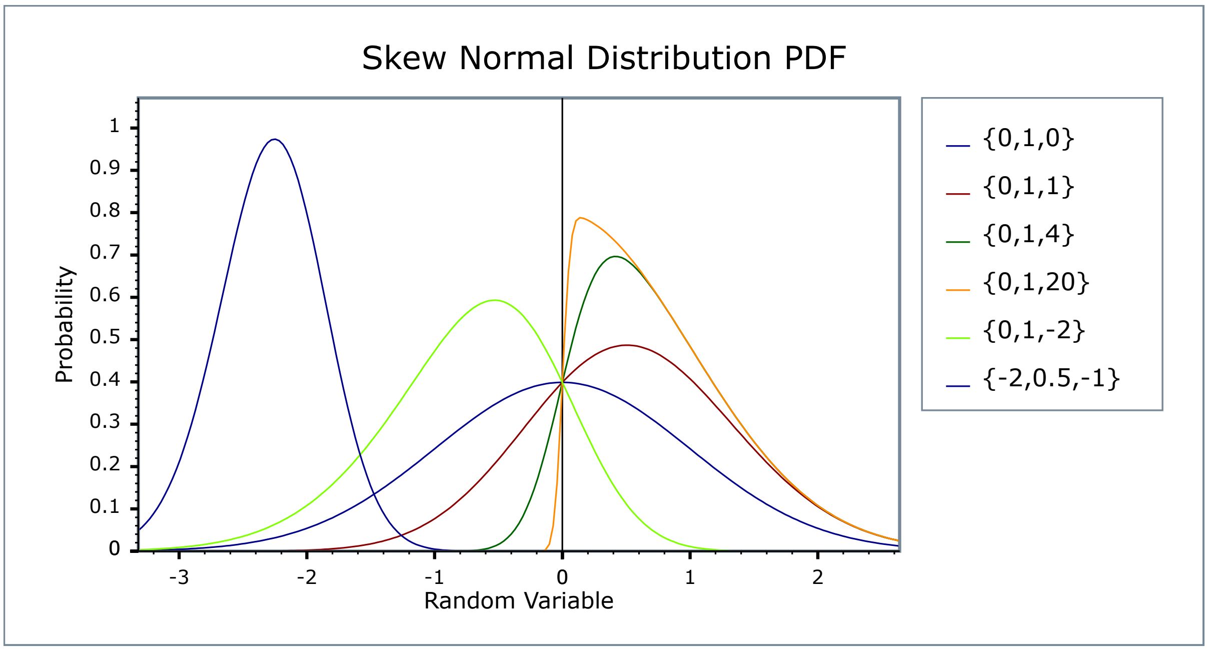
$$\frac{1}{(\omega\pi)}e^{-\frac{(x-\xi)^2}{(2\omega^2)}}\int_{-\infty}^{\left(\alpha\left(\frac{(x-\xi)}{\omega}\right)\right)} e^{-\frac{t^2}{2}} \quad dt$$

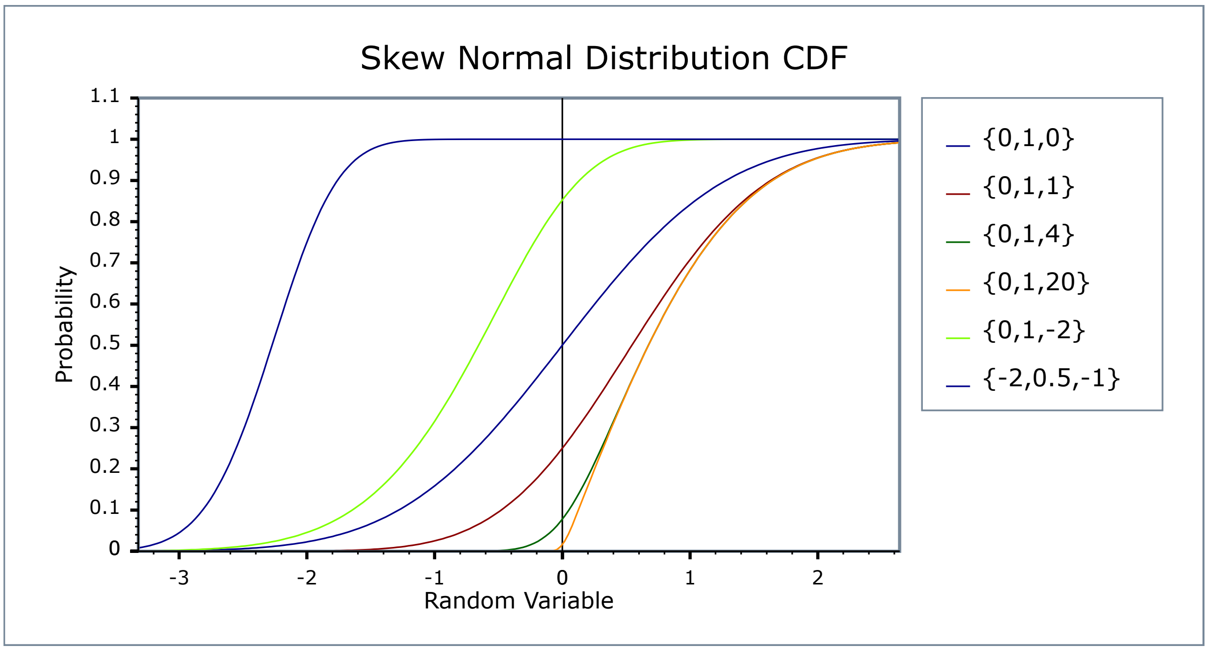
and CDF:

$$\Phi\left(\frac{(x-\xi)}{\omega}\right)-2T\left(\left(\frac{(x-\xi)}{\omega}\right),\ \alpha\right)$$

where *T(h,a)* is Owen's T function, and *Φ(x)* is the normal distribution.

The variation the PDF and CDF with its parameters is illustrated in the following graphs:





Member Functions

```
skew_normal_distribution(RealType location = 0, RealType scale = 1, RealType shape = 0);
```

Constructs a skew_normal distribution with location ξ, scale ω and shape α.

Requires scale > 0, otherwise domain_error is called.

```
RealType location()const;
```

returns the location ξ of this distribution,

```
RealType scale()const;
```

returns the scale ω of this distribution,

```
RealType shape()const;
```

returns the shape α of this distribution.

(Location and scale function match other similar distributions, allowing the functions `find_location` and `find_scale` to be used generically).

Note

While the shape parameter may be chosen arbitrarily (finite), the resulting **skewness** of the distribution is in fact limited to about (-1, 1); strictly, the interval is (-0.9952717, 0.9952717).

A parameter δ is related to the shape α by δ = α / (1 + α²), and used in the expression for skewness

$$\frac{(4-\pi)}{2} \quad \frac{\left(\delta\sqrt{\left(\frac{2}{\pi}\right)}\right)^3}{\left(1-2\frac{\delta^2}{\pi}\right)^{\left(\frac{3}{2}\right)}}$$

References

- Skew-Normal Probability Distribution for many links and bibliography.
- A very brief introduction to the skew-normal distribution by Adelchi Azzalini (2005-11-2).
- See a skew-normal function animation.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is *-[max_value], +[min_value]*. Infinite values are not supported.

There are no closed-form expression known for the mode and median, but these are computed for the

- mode - by finding the maximum of the PDF.

- median - by computing `quantile(1/2)`.

The maximum of the PDF is sought through searching the root of f'(x)=0.

Both involve iterative methods that will have lower accuracy than other estimates.

Testing

The R Project for Statistical Computing using library(sn) described at Skew-Normal Probability Distribution, and at R skew-normal(sn) package.

Package sn provides functions related to the skew-normal (SN) and the skew-t (ST) probability distributions, both for the univariate and for the the multivariate case, including regression models.

Wolfram Mathematica was also used to generate some more accurate spot test data.

Accuracy

The skew_normal distribution with shape = zero is implemented as a special case, equivalent to the normal distribution in terms of the error function, and therefore should have excellent accuracy.

The PDF and mean, variance, skewness and kurtosis are also accurately evaluated using analytical expressions. The CDF requires Owen's T function that is evaluated using a Boost C++ Owens T implementation of the algorithms of M. Patefield and D. Tandy, Journal of Statistical Software, 5(5), 1-25 (2000); the complicated accuracy of this function is discussed in detail at Owens T.

The median and mode are calculated by iterative root finding, and both will be less accurate.

Implementation

In the following table, ξ is the location of the distribution, and ω is its scale, and α is its shape.

Function	**Implementation Notes**
pdf	Using: $$\frac{1}{(\omega\pi)} e^{-\frac{(x-\xi)^2}{(2\omega^2)}} \int_{-\infty}^{\left(\alpha\left(\frac{(x-\xi)}{\omega}\right)\right)} e^{-\frac{t^2}{2}} \quad dt$$
cdf	Using: $$\Phi\left(\frac{(x-\xi)}{\omega}\right) - 2T\left(\left(\frac{(x-\xi)}{\omega}\right),\ \alpha\right)$$ where *T(h,a)* is Owen's T function, and *Φ(x)* is the normal distribution.
cdf complement	Using: complement of normal distribution + 2 * Owens_t
quantile	Maximum of the pdf is sought through searching the root of f'(x)=0
quantile from the complement	-quantile(SN(-location ξ, scale ω, -shapeα), p)
location	location ξ
scale	scale ω
shape	shape α
median	quantile(1/2)
mean	$$\xi + \omega\delta\sqrt{\left(\frac{2}{\pi}\right)} \quad where \quad \delta = \frac{\alpha}{\sqrt{(1+\alpha^2)}}$$
mode	Maximum of the pdf is sought through searching the root of f'(x)=0
variance	$$\omega^2\left(1 - 2\frac{\delta^2}{\pi}\right)$$
skewness	$$\frac{(4-\pi)}{2} \quad \frac{\left(\delta\sqrt{\left(\frac{2}{\pi}\right)}\right)^3}{\left(1-2\frac{\delta^2}{\pi}\right)^{\left(\frac{3}{2}\right)}}$$
kurtosis	kurtosis excess-3
kurtosis excess	$$2(\pi-3) \quad \frac{\left(\delta\sqrt{\left(\frac{2}{\pi}\right)}\right)^4}{\left(1-2\frac{\delta^2}{\pi}\right)^2}$$

Students t Distribution

```
#include <boost/math/distributions/students_t.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class students_t_distribution;

typedef students_t_distribution<> students_t;

template <class RealType, class Policy>
class students_t_distribution
{
   typedef RealType value_type;
   typedef Policy   policy_type;

   // Constructor:
   students_t_distribution(const RealType& v);

   // Accessor:
   RealType degrees_of_freedom()const;

   // degrees of freedom estimation:
   static RealType find_degrees_of_freedom(
      RealType difference_from_mean,
      RealType alpha,
      RealType beta,
      RealType sd,
      RealType hint = 100);
};

}} // namespaces
```

Student's t-distribution is a statistical distribution published by William Gosset in 1908. His employer, Guinness Breweries, required him to publish under a pseudonym (possibly to hide that they were using statistics to improve beer quality), so he chose "Student".

Given N independent measurements, let

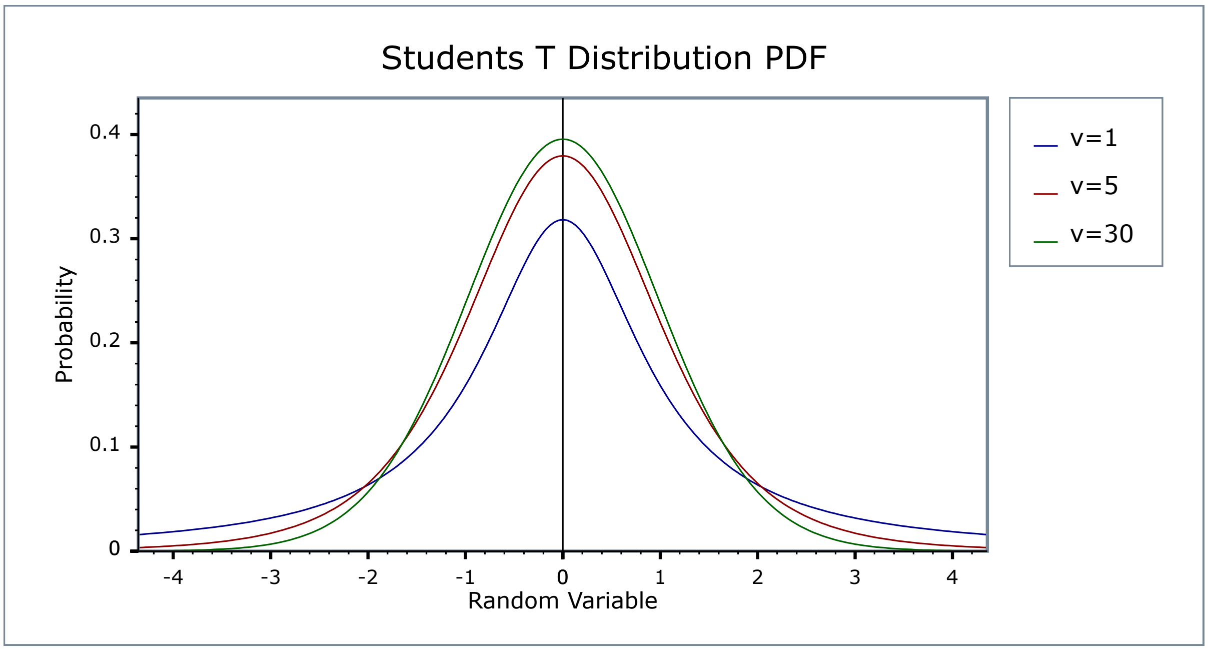
$$t = \frac{\mu - M}{\frac{s}{\sqrt{N}}}$$

where *M* is the population mean, *μ* is the sample mean, and *s* is the sample variance.

Student's t-distribution is defined as the distribution of the random variable t which is - very loosely - the "best" that we can do not knowing the true standard deviation of the sample. It has the PDF:

$$\mathrm{f}(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)\left(1+\frac{x^2}{\nu}\right)^{\frac{\nu+1}{2}}}$$

The Student's t-distribution takes a single parameter: the number of degrees of freedom of the sample. When the degrees of freedom is *one* then this distribution is the same as the Cauchy-distribution. As the number of degrees of freedom tends towards infinity, then this distribution approaches the normal-distribution. The following graph illustrates how the PDF varies with the degrees of freedom ν:



Member Functions

```
students_t_distribution(const RealType& v);
```

Constructs a Student's t-distribution with *v* degrees of freedom.

Requires *v* > 0, including infinity (if RealType permits), otherwise calls domain_error. Note that non-integral degrees of freedom are supported, and are meaningful under certain circumstances.

```
RealType degrees_of_freedom()const;
```

returns the number of degrees of freedom of this distribution.

```
static RealType find_degrees_of_freedom(
   RealType difference_from_mean,
   RealType alpha,
   RealType beta,
   RealType sd,
   RealType hint = 100);
```

returns the number of degrees of freedom required to observe a significant result in the Student's t test when the mean differs from the "true" mean by *difference_from_mean*.

difference_from_mean	The difference between the true mean and the sample mean that we wish to show is significant.
alpha	The maximum acceptable probability of rejecting the null hypothesis when it is in fact true.
beta	The maximum acceptable probability of failing to reject the null hypothesis when it is in fact false.
sd	The sample standard deviation.
hint	A hint for the location to start looking for the result, a good choice for this would be the sample size of a previous borderline Student's t test.

Note

Remember that for a two-sided test, you must divide alpha by two before calling this function.

For more information on this function see the NIST Engineering Statistics Handbook.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [-∞, +∞].

Examples

Various worked examples are available illustrating the use of the Student's t distribution.

Accuracy

The normal distribution is implemented in terms of the incomplete beta function and its inverses, refer to accuracy data on those functions for more information.

Implementation

In the following table *v* is the degrees of freedom of the distribution, *t* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = $(v / (v + t^2))^{(1+v)/2}$ / (sqrt(v) * beta(v/2, 0.5))
cdf	Using the relations: p = 1 - z *iff t > 0* p = z *otherwise* where z is given by: ibeta(v / 2, 0.5, v / (v + t^2)) / 2 *iff v < 2t²* ibetac(0.5, v / 2, t^2 / (v + t^2) / 2 *otherwise*
cdf complement	Using the relation: q = cdf(-t)
quantile	Using the relation: t = sign(p - 0.5) * sqrt(v * y / x) where: x = ibeta_inv(v / 2, 0.5, 2 * min(p, q)) y = 1 - x The quantities *x* and *y* are both returned by ibeta_inv without the subtraction implied above.
quantile from the complement	Using the relation: t = -quantile(q)
mode	0
mean	0
variance	if (v > 2) v / (v - 2) else NaN
skewness	if (v > 3) 0 else NaN
kurtosis	if (v > 4) 3 * (v - 2) / (v - 4) else NaN
kurtosis excess	if (v > 4) 6 / (df - 4) else NaN

If the moment index *k* is less than *v*, then the moment is undefined. Evaluating the moment will throw a domain_error unless ignored by a policy, when it will return `std::numeric_limits<>::quiet_NaN();`

Implementation

(By popular demand, we now support infinite argument and random deviate. But we have not implemented the return of infinity as suggested by Wikipedia Student's t, instead throwing a domain error or return NaN. See also https://svn.boost.org/trac/boost/ticket/7177.)

Triangular Distribution

```
#include <boost/math/distributions/triangular.hpp>
```

```
namespace boost{ namespace math{
 template <class RealType = double,
           class Policy   = policies::policy<> >
 class triangular_distribution;

 typedef triangular_distribution<> triangular;

 template <class RealType, class Policy>
 class triangular_distribution
 {
 public:
    typedef RealType value_type;
    typedef Policy   policy_type;

    triangular_distribution(RealType lower = -1, RealType mode = 0) RealType upper = 1); // ↵
Constructor.
       : m_lower(lower), m_mode(mode), m_upper(upper) // Default is -1, 0, +1 symmetric trian↵
gular distribution.
    // Accessor functions.
    RealType lower()const;
    RealType mode()const;
    RealType upper()const;
 }; // class triangular_distribution

}} // namespaces
```

The triangular distribution is a continuous probability distribution with a lower limit a, mode c, and upper limit b.

The triangular distribution is often used where the distribution is only vaguely known, but, like the uniform distribution, upper and limits are 'known', but a 'best guess', the mode or center point, is also added. It has been recommended as a proxy for the beta distribution. The distribution is used in business decision making and project planning.

The triangular distribution is a distribution with the probability density function:

f(x) =

- 2(x-a)/(b-a) (c-a) for a <= x <= c
- 2(b-x)/(b-a)(b-c) for c < x <= b

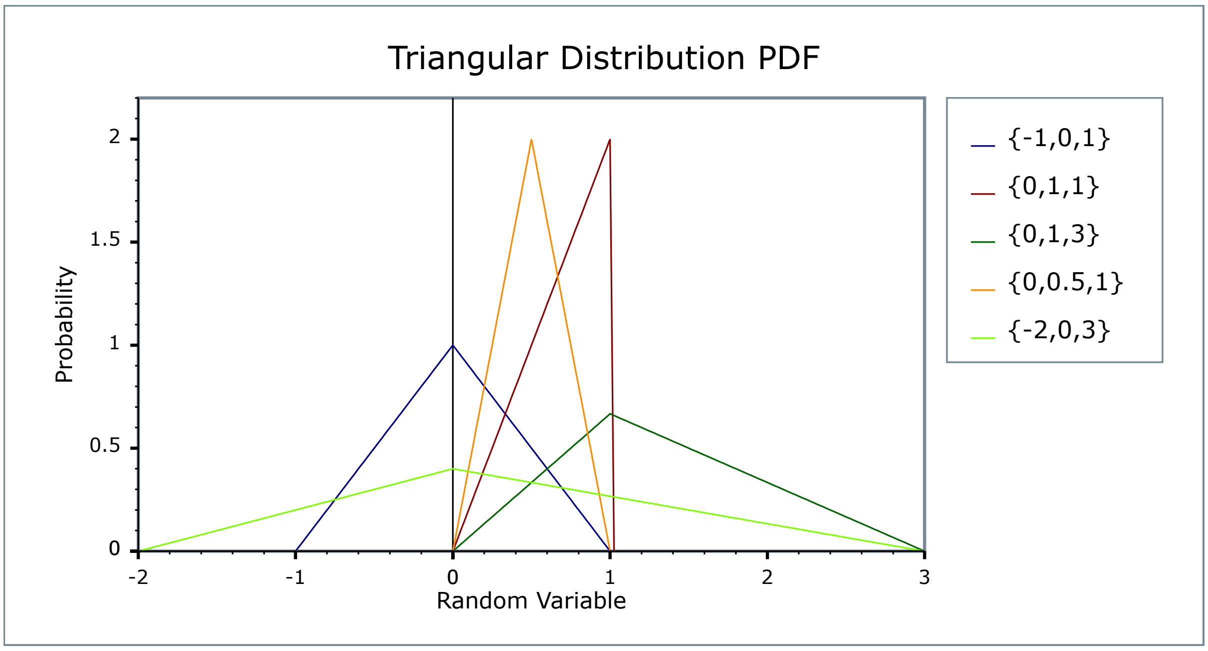
Parameter *a* (lower) can be any finite value. Parameter *b* (upper) can be any finite value > a (lower). Parameter *c* (mode) a <= c <= b. This is the most probable value.

The random variate x must also be finite, and is supported lower <= x <= upper.

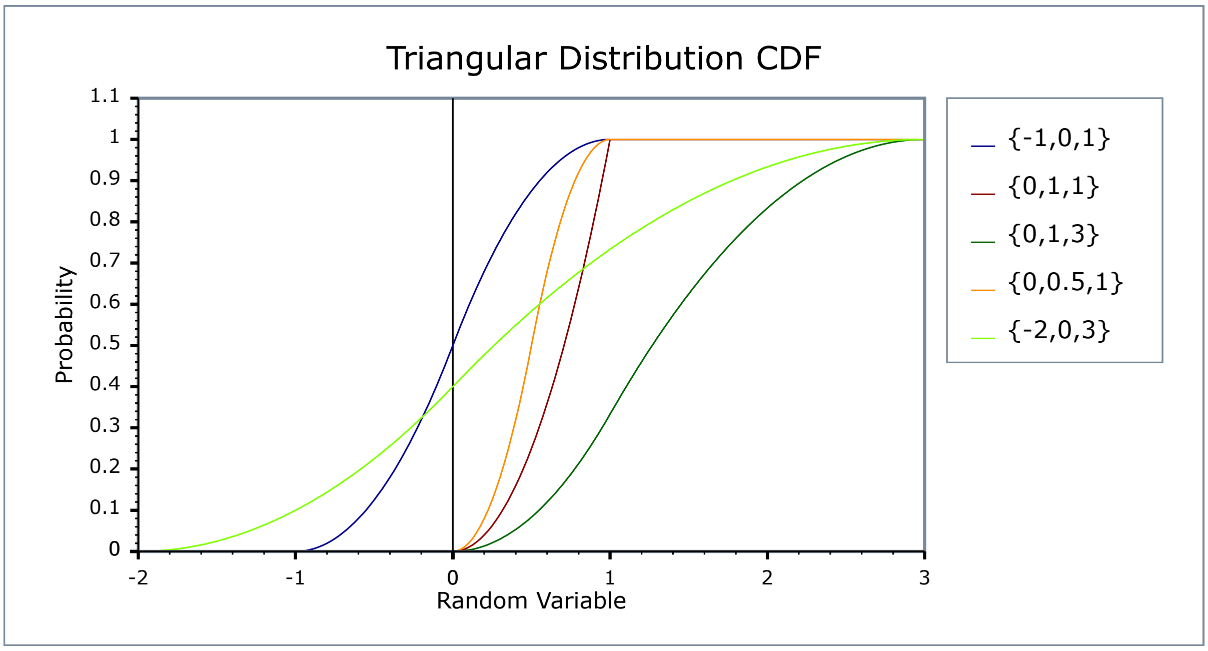
The triangular distribution may be appropriate when an assumption of a normal distribution is unjustified because uncertainty is caused by rounding and quantization from analog to digital conversion. Upper and lower limits are known, and the most probable value lies midway.

The distribution simplifies when the 'best guess' is either the lower or upper limit - a 90 degree angle triangle. The 001 triangular distribution which expresses an estimate that the lowest value is the most likely; for example, you believe that the next-day quoted delivery date is most likely (knowing that a quicker delivery is impossible - the postman only comes once a day), and that longer delays are decreasingly likely, and delivery is assumed to never take more than your upper limit.

The following graph illustrates how the probability density function PDF varies with the various parameters:



and cumulative distribution function



Member Functions

```
triangular_distribution(RealType lower = 0, RealType mode = 0 RealType upper = 1);
```

Constructs a triangular distribution with lower *lower* (a) and upper *upper* (b).

Requires that the *lower*, *mode* and *upper* parameters are all finite, otherwise calls domain_error.

Warning

These constructors are slightly different from the analogs provided by Wolfram MathWorld Triangular distribution, where

`TriangularDistribution[{min, max}]` represents a **symmetric** triangular statistical distribution giving values between min and max.
`TriangularDistribution[]` represents a **symmetric** triangular statistical distribution giving values between 0 and 1.
`TriangularDistribution[{min, max}, c]` represents a triangular distribution with mode at c (usually **asymmetric**).

So, for example, to compute a variance using Wolfram Alpha, use `N[variance[TriangularDistribution{1, +2}], 50]`

The parameters of a distribution can be obtained using these member functions:

```
RealType lower()const;
```

Returns the *lower* parameter of this distribution (default -1).

```
RealType mode()const;
```

Returns the *mode* parameter of this distribution (default 0).

```
RealType upper()const;
```

Returns the *upper* parameter of this distribution (default+1).

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is \lowerto \upper, and the supported range is lower <= x <= upper.

Accuracy

The triangular distribution is implemented with simple arithmetic operators and so should have errors within an epsilon or two, except quantiles with arguments nearing the extremes of zero and unity.

Implementation

In the following table, a is the *lower* parameter of the distribution, c is the *mode* parameter, b is the *upper* parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = 0 for x < mode, 2(x-a)/(b-a)(c-a) else 2*(b-x)/((b-a)(b-c))
cdf	Using the relation: cdf = 0 for x < mode $(x-a)^2$/((b-a)(c-a)) else 1 - $(b-x)^2$/((b-a)(b-c))
cdf complement	Using the relation: q = 1 - p

quantile	let p0 = (c-a)/(b-a) the point of inflection on the cdf, then given probability p and q = 1-p:
	
	x = sqrt((b-a)(c-a)p) + a ; for p < p0
	
	x = c ; for p == p0
	
	x = b - sqrt((b-a)(b-c)q) ; for p > p0
	
	(See /boost/math/distributions/triangular.hpp for details.)

quantile from the complement	As quantile (See /boost/math/distributions/triangular.hpp for details.)
mean	(a + b + 3) / 3
variance	$(a^2+b^2+c^2$ - ab - ac - bc)/18
mode	c
skewness	(See /boost/math/distributions/triangular.hpp for details).
kurtosis	12/5
kurtosis excess	-3/5

Some 'known good' test values were obtained using Wolfram Alpha.

References

- Wikpedia triangular distribution
- Weisstein, Eric W. "Triangular Distribution." From MathWorld--A Wolfram Web Resource.
- Evans, M.; Hastings, N.; and Peacock, B. "Triangular Distribution." Ch. 40 in Statistical Distributions, 3rd ed. New York: Wiley, pp. 187-188, 2000, ISBN - 0471371246.
- Gejza Wimmer, Viktor Witkovsky and Tomas Duby, Measurement Science Review, Volume 2, Section 1, 2002, Proper Rounding Of The Measurement Results Under The Assumption Of Triangular Distribution.

Uniform Distribution

```
#include <boost/math/distributions/uniform.hpp>
```

```
namespace boost{ namespace math{
 template <class RealType = double,
           class Policy   = policies::policy<> >
 class uniform_distribution;

 typedef uniform_distribution<> uniform;

 template <class RealType, class Policy>
 class uniform_distribution
 {
 public:
    typedef RealType value_type;

    uniform_distribution(RealType lower = 0, RealType upper = 1); // Constructor.
       : m_lower(lower), m_upper(upper) // Default is standard uniform distribution.
    // Accessor functions.
    RealType lower()const;
    RealType upper()const;
 }; // class uniform_distribution

}} // namespaces
```

The uniform distribution, also known as a rectangular distribution, is a probability distribution that has constant probability.

The continuous uniform distribution is a distribution with the probability density function:

f(x) =

- 1 / (upper - lower) for lower < x < upper
- zero for x < lower or x > upper

and in this implementation:

- 1 / (upper - lower) for x = lower or x = upper

The choice of x = lower or x = upper is made because statistical use of this distribution judged is most likely: the method of maximum likelihood uses this definition.

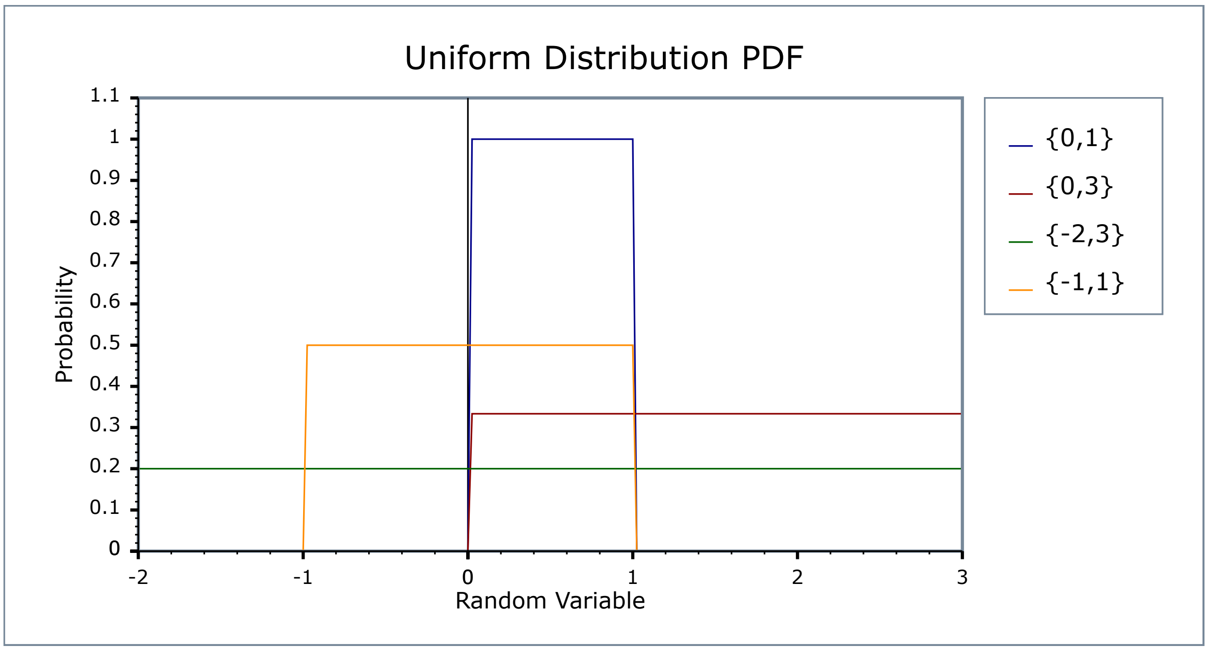
There is also a **discrete** uniform distribution.

Parameters lower and upper can be any finite value.

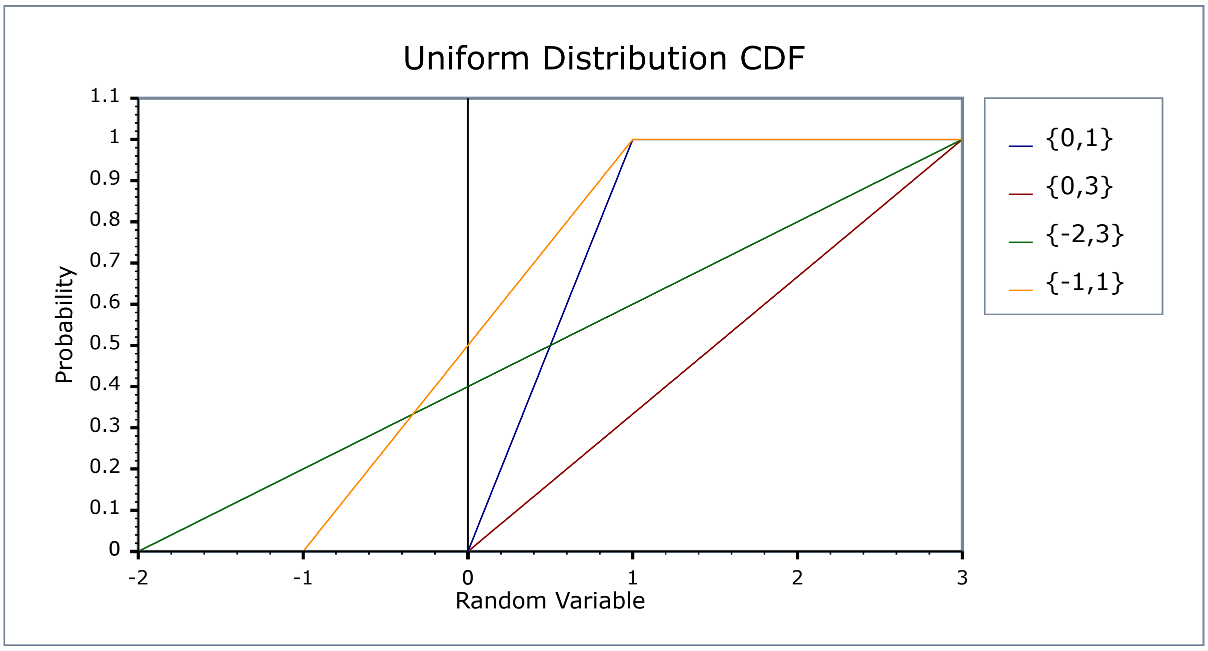
The random variate x must also be finite, and is supported lower <= x <= upper.

The lower parameter is also called the location parameter, that is where the origin of a plot will lie, and (upper - lower) is also called the scale parameter.

The following graph illustrates how the probability density function PDF varies with the shape parameter:



Likewise for the CDF:



Member Functions

```
uniform_distribution(RealType lower = 0, RealType upper = 1);
```

Constructs a uniform distribution with lower *lower* (a) and upper *upper* (b).

Requires that the *lower* and *upper* parameters are both finite; otherwise if infinity or NaN then calls domain_error.

```
RealType lower()const;
```

Returns the *lower* parameter of this distribution.

```
RealType upper()const;
```

Returns the *upper* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is any finite value, but the supported range is only *lower* <= x <= *upper*.

Accuracy

The uniform distribution is implemented with simple arithmetic operators and so should have errors within an epsilon or two.

Implementation

In the following table a is the *lower* parameter of the distribution, b is the *upper* parameter, *x* is the random variate, *p* is the probability and *q = 1-p*.

Function	**Implementation Notes**
pdf	Using the relation: pdf = 0 for x < a, 1 / (b - a) for a <= x <= b, 0 for x > b
cdf	Using the relation: cdf = 0 for x < a, (x - a) / (b - a) for a <= x <= b, 1 for x > b
cdf complement	Using the relation: q = 1 - p, (b - x) / (b - a)
quantile	Using the relation: x = p * (b - a) + a;
quantile from the complement	x = -q * (b - a) + b
mean	(a + b) / 2
variance	$(b - a)^2 / 12$
mode	any value in [a, b] but a is chosen. (Would NaN be better?)
skewness	0
kurtosis excess	-6/5 = -1.2 exactly. (kurtosis - 3)
kurtosis	9/5

References

- Wikpedia continuous uniform distribution
- Weisstein, Weisstein, Eric W. "Uniform Distribution." From MathWorld--A Wolfram Web Resource.
- http://www.itl.nist.gov/div898/handbook/eda/section3/eda3662.htm

Weibull Distribution

```
#include <boost/math/distributions/weibull.hpp>
```

```
namespace boost{ namespace math{

template <class RealType = double,
          class Policy   = policies::policy<> >
class weibull_distribution;

typedef weibull_distribution<> weibull;

template <class RealType, class Policy>
class weibull_distribution
{
public:
   typedef RealType value_type;
   typedef Policy   policy_type;
   // Construct:
   weibull_distribution(RealType shape, RealType scale = 1)
   // Accessors:
   RealType shape()const;
   RealType scale()const;
};

}} // namespaces
```

The Weibull distribution is a continuous distribution with the probability density function:

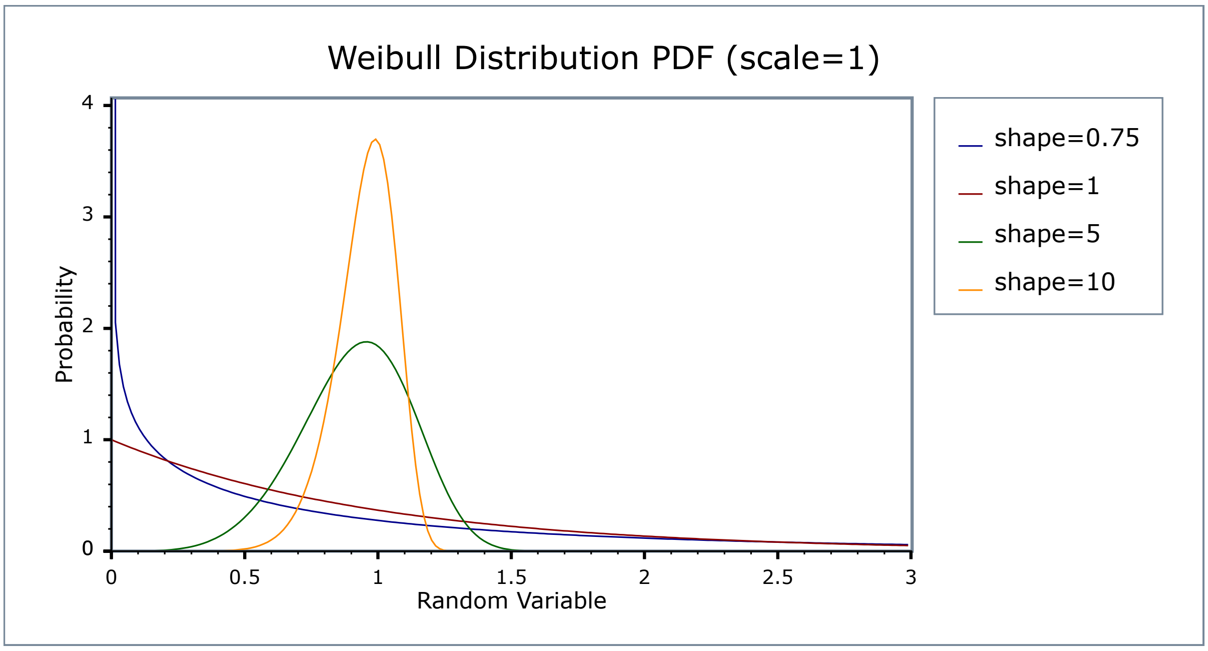
$f(x; \alpha, \beta) = (\alpha/\beta) * (x / \beta)^{\alpha - 1} * e^{-(x/\beta)^{\alpha}}$

For shape parameter $\alpha > 0$, and scale parameter $\beta > 0$, and $x > 0$.

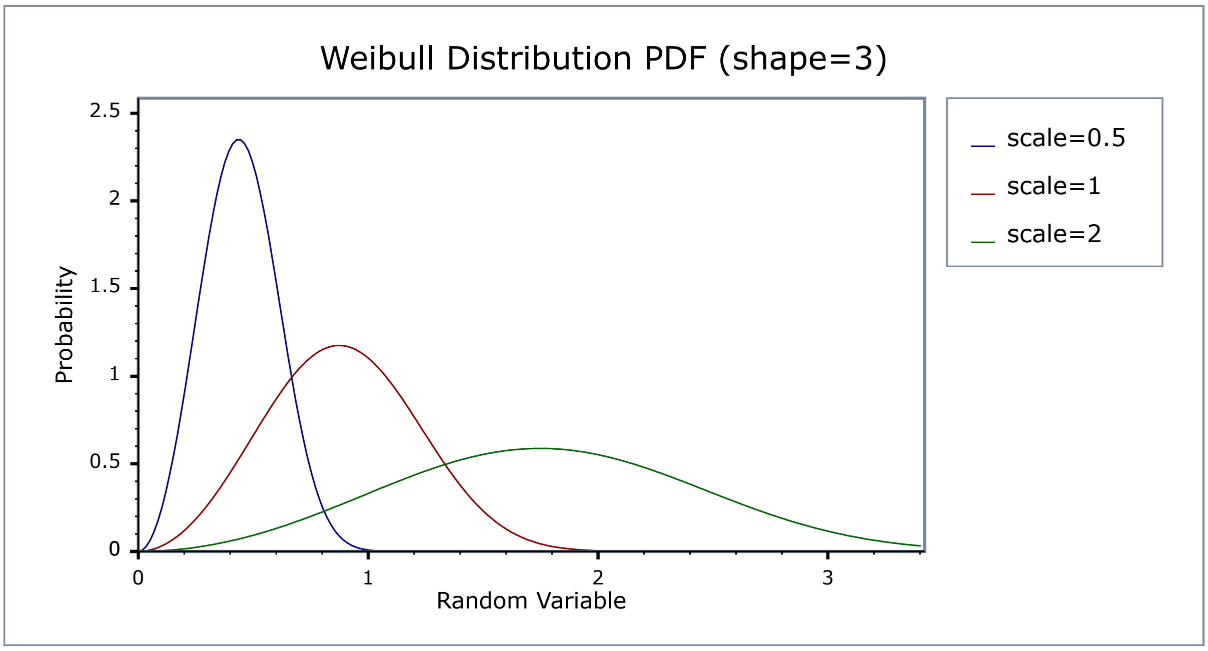
The Weibull distribution is often used in the field of failure analysis; in particular it can mimic distributions where the failure rate varies over time. If the failure rate is:

- constant over time, then $\alpha = 1$, suggests that items are failing from random events.
- decreases over time, then $\alpha < 1$, suggesting "infant mortality".
- increases over time, then $\alpha > 1$, suggesting "wear out" - more likely to fail as time goes by.

The following graph illustrates how the PDF varies with the shape parameter α:



While this graph illustrates how the PDF varies with the scale parameter β:



Related distributions

When α = 3, the Weibull distribution appears similar to the normal distribution. When α = 1, the Weibull distribution reduces to the exponential distribution. The relationship of the types of extreme value distributions, of which the Weibull is but one, is discussed by Extreme Value Distributions, Theory and Applications Samuel Kotz & Saralees Nadarajah.

Member Functions

```
weibull_distribution(RealType shape, RealType scale = 1);
```

Constructs a Weibull distribution with shape *shape* and scale *scale*.

Requires that the *shape* and *scale* parameters are both greater than zero, otherwise calls domain_error.

```
RealType shape()const;
```

Returns the *shape* parameter of this distribution.

```
RealType scale()const;
```

Returns the *scale* parameter of this distribution.

Non-member Accessors

All the usual non-member accessor functions that are generic to all distributions are supported: Cumulative Distribution Function, Probability Density Function, Quantile, Hazard Function, Cumulative Hazard Function, mean, median, mode, variance, standard deviation, skewness, kurtosis, kurtosis_excess, range and support.

The domain of the random variable is [0, ∞].

Accuracy

The Weibull distribution is implemented in terms of the standard library `log` and `exp` functions plus expm1 and log1p and as such should have very low error rates.

Implementation

In the following table α is the shape parameter of the distribution, β is its scale parameter, *x* is the random variate, *p* is the probability and $q = 1-p$.

Function	Implementation Notes
pdf	Using the relation: pdf = $\alpha\beta^{-\alpha} x^{\alpha - 1} e^{-(x/beta)^{alpha}}$
cdf	Using the relation: p = -expm1(-(x/β)$^{\alpha}$)
cdf complement	Using the relation: q = $e^{-(x/\beta)^{\alpha}}$
quantile	Using the relation: x = β * (-log1p(-p))$^{1/\alpha}$
quantile from the complement	Using the relation: x = β * (-log(q))$^{1/\alpha}$
mean	$\beta * \Gamma(1 + 1/\alpha)$
variance	$\beta^2(\Gamma(1 + 2/\alpha) - \Gamma^2(1 + 1/\alpha))$
mode	$\beta((\alpha - 1) / \alpha)^{1/\alpha}$
skewness	Refer to Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis	Refer to Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.
kurtosis excess	Refer to Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.

References

- http://en.wikipedia.org/wiki/Weibull_distribution

- Weisstein, Eric W. "Weibull Distribution." From MathWorld--A Wolfram Web Resource.
- Weibull in NIST Exploratory Data Analysis

Distribution Algorithms

Finding the Location and Scale for Normal and similar distributions

Two functions aid finding location and scale of random variable z to give probability p (given a scale or location). Only applies to distributions like normal, lognormal, extreme value, Cauchy, (and symmetrical triangular), that have scale and location properties.

These functions are useful to predict the mean and/or standard deviation that will be needed to meet a specified minimum weight or maximum dose.

Complement versions are also provided, both with explicit and implicit (default) policy.

```
using boost::math::policies::policy; // May be needed by users defining their own policies.
using boost::math::complement; // Will be needed by users who want to use complements.
```

find_location function

```
#include <boost/math/distributions/find_location.hpp>
```

```
namespace boost{ namespace math{

template <class Dist, class Policy> // explicit error handling policy
  typename Dist::value_type find_location( // For example, normal mean.
  typename Dist::value_type z, // location of random variable z to give probability, P(X > z) ↵
== p.
  // For example, a nominal minimum acceptable z, so that p * 100 % are > z
  typename Dist::value_type p, // probability value desired at x, say 0.95 for 95% > z.
  typename Dist::value_type scale, // scale parameter, for example, normal standard deviation.
  const Policy& pol);

template <class Dist>  // with default policy.
  typename Dist::value_type find_location( // For example, normal mean.
  typename Dist::value_type z, // location of random variable z to give probability, P(X > z) ↵
== p.
  // For example, a nominal minimum acceptable z, so that p * 100 % are > z
  typename Dist::value_type p, // probability value desired at x, say 0.95 for 95% > z.
  typename Dist::value_type scale); // scale parameter, for example, normal standard deviation.

  }} // namespaces
```

find_scale function

```
#include <boost/math/distributions/find_scale.hpp>
```

```
namespace boost{ namespace math{

template <class Dist, class Policy>
  typename Dist::value_type find_scale( // For example, normal mean.
  typename Dist::value_type z, // location of random variable z to give probability, P(X > z) ↵
== p.
  // For example, a nominal minimum acceptable weight z, so that p * 100 % are > z
  typename Dist::value_type p, // probability value desired at x, say 0.95 for 95% > z.
  typename Dist::value_type location, // location parameter, for example, normal distribution ↵
mean.
  const Policy& pol);

 template <class Dist> // with default policy.
   typename Dist::value_type find_scale( // For example, normal mean.
   typename Dist::value_type z, // location of random variable z to give probability, P(X > ↵
z) == p.
   // For example, a nominal minimum acceptable z, so that p * 100 % are > z
   typename Dist::value_type p, // probability value desired at x, say 0.95 for 95% > z.
   typename Dist::value_type location) // location parameter, for example, normal distribution ↵
mean.
}} // namespaces
```

All argument must be finite, otherwise domain_error is called.

Probability arguments must be [0, 1], otherwise domain_error is called.

If the choice of arguments would give a negative scale, domain_error is called, unless the policy is to ignore, when the negative (impossible) value of scale is returned.

Find Mean and standard deviation examples gives simple examples of use of both find_scale and find_location, and a longer example finding means and standard deviations of normally distributed weights to meet a specification.

Extras/Future Directions

Adding Additional Location and Scale Parameters

In some modelling applications we require a distribution with a specific location and scale: often this equates to a specific mean and standard deviation, although for many distributions the relationship between these properties and the location and scale parameters are non-trivial. See http://www.itl.nist.gov/div898/handbook/eda/section3/eda364.htm for more information.

The obvious way to handle this is via an adapter template:

```
template <class Dist>
class scaled_distribution
{
   scaled_distribution(
     const Dist dist,
     typename Dist::value_type location,
     typename Dist::value_type scale = 0);
};
```

Which would then have its own set of overloads for the non-member accessor functions.

An "any_distribution" class

It is easy to add a distribution object that virtualises the actual type of the distribution, and can therefore hold "any" object that conforms to the conceptual requirements of a distribution:

```
template <class RealType>
class any_distribution
{
public:
   template <class Distribution>
   any_distribution(const Distribution& d);
};

// Get the cdf of the underlying distribution:
template <class RealType>
RealType cdf(const any_distribution<RealType>& d, RealType x);
// etc....
```

Such a class would facilitate the writing of non-template code that can function with any distribution type.

The Statistical Distribution Explorer utility for Windows is a usage example.

It's not clear yet whether there is a compelling use case though. Possibly tests for goodness of fit might provide such a use case: this needs more investigation.

Higher Level Hypothesis Tests

Higher-level tests roughly corresponding to the Mathematica Hypothesis Tests package could be added reasonably easily, for example:

```
template <class InputIterator>
typename std::iterator_traits<InputIterator>::value_type
   test_equal_mean(
     InputIterator a,
     InputIterator b,
     typename std::iterator_traits<InputIterator>::value_type expected_mean);
```

Returns the probability that the data in the sequence [a,b) has the mean *expected_mean*.

Integration With Statistical Accumulators

Eric Niebler's accumulator framework - also work in progress - provides the means to calculate various statistical properties from experimental data. There is an opportunity to integrate the statistical tests with this framework at some later date:

```
// Define an accumulator, all required statistics to calculate the test
// are calculated automatically:
accumulator_set<double, features<tag::test_expected_mean> > acc(expected_mean=4);
// Pass our data to the accumulator:
acc = std::for_each(mydata.begin(), mydata.end(), acc);
// Extract the result:
double p = probability(acc);
```

Vector Functionals

Univariate Statistics

Synopsis

```
#include <boost/math/tools/univariate_statistics.hpp>

namespace boost{ namespace math{ namespace tools {

    template<class Container>
    auto mean(Container const & c);

    template<class ForwardIterator>
    auto mean(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto variance(Container const & c);

    template<class ForwardIterator>
    auto variance(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto sample_variance(Container const & c);

    template<class ForwardIterator>
    auto sample_variance(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto skewness(Container const & c);

    template<class ForwardIterator>
    auto skewness(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto kurtosis(Container const & c);

    template<class ForwardIterator>
    auto kurtosis(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto excess_kurtosis(Container const & c);

    template<class ForwardIterator>
    auto excess_kurtosis(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto first_four_moments(Container const & c);

    template<class ForwardIterator>
    auto first_four_moments(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto median(Container & c);

    template<class ForwardIterator>
    auto median(ForwardIterator first, ForwardIterator last);

    template<class RandomAccessIterator>
    auto median_absolute_deviation(ForwardIterator first, ForwardIterator last, type↵
name std::iterator_traits<RandomAccessIterator>::value_type center=std::numeric_lim↵
its<Real>::quiet_NaN());
```

```
    template<class RandomAccessContainer>
    auto median_absolute_deviation(RandomAccessContainer v, typename RandomAccessContain↵
er::value_type center=std::numeric_limits<Real>::quiet_NaN());

    template<class Container>
    auto gini_coefficient(Container & c);

    template<class ForwardIterator>
    auto gini_coefficient(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto sample_gini_coefficient(Container & c);

    template<class ForwardIterator>
    auto sample_gini_coefficient(ForwardIterator first, ForwardIterator last);

}}}
```

Description

The file `boost/math/tools/univariate_statistics.hpp` is a set of facilities for computing scalar values from vectors.

Many of these functionals have trivial naive implementations, but experienced programmers will recognize that even trivial algorithms are easy to screw up, and that numerical instabilities often lurk in corner cases. We have attempted to do our "due diligence" to root out these problems-scouring the literature for numerically stable algorithms for even the simplest of functionals.

Nota bene: Some similar functionality is provided in Boost Accumulators Framework. These accumulators should be used in real-time applications; `univariate_statistics.hpp` should be used when CPU vectorization is needed. As a reminder, remember that to actually *get* vectorization, compile with `-march=native -O3` flags.

We now describe each functional in detail. Our examples use `std::vector<double>` to hold the data, but this not required. In general, you can store your data in an Eigen array, and Armadillo vector, `std::array`, and for many of the routines, a `std::forward_list`. These routines are usable in float, double, long double, and Boost.Multiprecision precision, as well as their complex extensions whenever the computation is well-defined. For certain operations (total variation, for example) integer inputs are supported.

Mean

```
std::vector<double> v{1,2,3,4,5};
double mu = boost::math::tools::mean(v.cbegin(), v.cend());
// Alternative syntax if you want to use entire container:
mu = boost::math::tools::mean(v);
```

The implementation follows Higham 1.6a. The data is not modified and must be forward iterable. Works with real and integer data. If the input is an integer type, the output is a double precision float.

Variance

```
std::vector<double> v{1,2,3,4,5};
Real sigma_sq = boost::math::tools::variance(v.cbegin(), v.cend());
```

If you don't need to calculate on a subset of the input, then the range call is more terse:

```
std::vector<double> v{1,2,3,4,5};
Real sigma_sq = boost::math::tools::variance(v);
```

The implementation follows Higham 1.6b. The input data must be forward iterable and the range `[first, last)` must contain at least two elements. It is *not* in general sensible to pass complex numbers to this routine. If integers are passed as input, then the output is a double precision float.

`boost::math::tools::variance` returns the population variance. If you want a sample variance, use

```
std::vector<double> v{1,2,3,4,5};
Real sn_sq = boost::math::tools::sample_variance(v);
```

Skewness

Computes the skewness of a dataset:

```
std::vector<double> v{1,2,3,4,5};
double skewness = boost::math::tools::skewness(v);
// skewness = 0.
```

The input vector is not modified, works with integral and real data. If the input data is integral, the output is a double precision float.

For a dataset consisting of a single constant value, we take the skewness to be zero by definition.

The implementation follows Pebay.

Kurtosis

Computes the kurtosis of a dataset:

```
std::vector<double> v{1,2,3,4,5};
double kurtosis = boost::math::tools::kurtosis(v);
// kurtosis = 17/10
```

The implementation follows Pebay. The input data must be forward iterable and must consist of real or integral values. If the input data is integral, the output is a double precision float. Note that this is *not* the excess kurtosis. If you require the excess kurtosis, use `boost::math::tools::excess_kurtosis`. This function simply subtracts 3 from the kurtosis, but it makes eminently clear our definition of kurtosis.

First four moments

Simultaneously computes the first four central moments in a single pass through the data:

```
std::vector<double> v{1,2,3,4,5};
auto [M1, M2, M3, M4] = boost::math::tools::first_four_moments(v);
```

Median

Computes the median of a dataset:

```
std::vector<double> v{1,2,3,4,5};
double m = boost::math::tools::median(v.begin(), v.end());
```

Nota bene: The input vector is modified. The calculation of the median is a thin wrapper around the C++11 `nth_element`. Therefore, all requirements of `std::nth_element` are inherited by the median calculation. In particular, the container must allow random access.

Median Absolute Deviation

Computes the median absolute deviation of a dataset:

```
std::vector<double> v{1,2,3,4,5};
double mad = boost::math::tools::median_absolute_deviation(v);
```

By default, the deviation from the median is used. If you have some prior that the median is zero, or wish to compute the median absolute deviation from the mean, use the following:

```
// prior is that center is zero:
double center = 0;
double mad = boost::math::tools::median_absolute_deviation(v, center);

// compute median absolute deviation from the mean:
double mu = boost::math::tools::mean(v);
double mad = boost::math::tools::median_absolute_deviation(v, mu);
```

Nota bene: The input vector is modified. Again the vector is passed into a call to `nth_element`.

Gini Coefficient

Compute the Gini coefficient of a dataset:

```
std::vector<double> v{1,0,0,0};
double gini = boost::math::tools::gini_coefficient(v);
// gini = 3/4
double s_gini = boost::math::tools::sample_gini_coefficient(v);
// s_gini = 1.
std::vector<double> w{1,1,1,1};
gini = boost::math::tools::gini_coefficient(w.begin(), w.end());
// gini = 0, as all elements are now equal.
```

Nota bene: The input data is altered: in particular, it is sorted. Makes a call to `std::sort`, and as such requires random access iterators.

The sample Gini coefficient lies in the range [0,1], whereas the population Gini coefficient is in the range [0, 1 - 1/ *n*].

Nota bene: There is essentially no reason to pass negative values to the Gini coefficient function. However, a use case (measuring wealth inequality when some people have negative wealth) exists, so we do not throw an exception when negative values are encountered. You should have *very* good cause to pass negative values to the Gini coefficient calculator. Another use case is found in signal processing, but the sorting is by magnitude and hence has a different implementation. See `absolute_gini_coefficient` for details.

References

- Higham, Nicholas J. *Accuracy and stability of numerical algorithms.* Vol. 80. Siam, 2002.
- Philippe P. Pébay: “Formulas for Robust, One-Pass Parallel Computation of Covariances and Arbitrary-Order Statistical Moments.” Technical Report SAND2008-6212, Sandia National Laboratories, September 2008.

Bivariate Statistics

Synopsis

```
#include <boost/math/tools/bivariate_statistics.hpp>

namespace boost{ namespace math{ namespace tools {

    template<class Container>
    auto covariance(Container const & u, Container const & v);

    template<class Container>
    auto means_and_covariance(Container const & u, Container const & v);

    template<class Container>
    auto correlation_coefficient(Container const & u, Container const & v);

}}}
```

Description

This file provides functions for computing bivariate statistics.

Covariance

Computes the population covariance of two datasets:

```
std::vector<double> u{1,2,3,4,5};
std::vector<double> v{1,2,3,4,5};
double cov_uv = boost::math::tools::covariance(u, v);
```

The implementation follows Bennet et al. The data is not modified. Requires a random-access container. Works with real-valued inputs and does not work with complex-valued inputs.

The algorithm used herein simultaneously generates the mean values of the input data *u* and *v*. For certain applications, it might be useful to get them in a single pass through the data. As such, we provide `means_and_covariance`:

```
std::vector<double> u{1,2,3,4,5};
std::vector<double> v{1,2,3,4,5};
auto [mu_u, mu_v, cov_uv] = boost::math::tools::means_and_covariance(u, v);
```

Correlation Coefficient

Computes the Pearson correlation coefficient of two datasets *u* and *v*:

```
std::vector<double> u{1,2,3,4,5};
std::vector<double> v{1,2,3,4,5};
double rho_uv = boost::math::tools::correlation_coefficient(u, v);
// rho_uv = 1.
```

The data must be random access and cannot be complex.

If one or both of the datasets is constant, the correlation coefficient is an indeterminant form (0/0) and definitions must be introduced to assign it a value. We use the following: If both datasets are constant, then the correlation coefficient is 1. If one dataset is constant, and the other is not, then the correlation coefficient is zero.

References

- Bennett, Janine, et al. *Numerically stable, single-pass, parallel statistics algorithms.* Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on. IEEE, 2009.

Signal Statistics

Synopsis

```
#include <boost/math/tools/signal_statistics.hpp>

namespace boost::math::tools {

    template<class Container>
    auto absolute_gini_coefficient(Container & c);

    template<class ForwardIterator>
    auto absolute_gini_coefficient(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto sample_absolute_gini_coefficient(Container & c);

    template<class ForwardIterator>
    auto sample_absolute_gini_coefficient(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto hoyer_sparsity(Container const & c);

    template<class ForwardIterator>
    auto hoyer_sparsity(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto oracle_snr(Container const & signal, Container const & noisy_signal);

    template<class Container>
    auto oracle_snr_db(Container const & signal, Container const & noisy_signal);

    template<class ForwardIterator>
    auto m2m4_snr_estimator(ForwardIterator first, ForwardIterator last, decltype(*first) es↵
timated_signal_kurtosis=1, decltype(*first) estimated_noise_kurtosis=3);

    template<class Container>
   auto m2m4_snr_estimator(Container const & noisy_signal, typename Container::value_type es↵
timated_signal_kurtosis=1, typename Container::value_type estimate_noise_kurtosis=3);

    template<class ForwardIterator>
   auto m2m4_snr_estimator_db(ForwardIterator first, ForwardIterator last, decltype(*first) es↵
timated_signal_kurtosis=1, decltype(*first) estimated_noise_kurtosis=3);

    template<class Container>
   auto m2m4_snr_estimator_db(Container const & noisy_signal,typename Container::value_type es↵
timated_signal_kurtosis=1, typename Container::value_type estimate_noise_kurtosis=3);

}
```

Description

The file `boost/math/tools/signal_statistics.hpp` is a set of facilities for computing quantities commonly used in signal analysis.

Our examples use `std::vector<double>` to hold the data, but this not required. In general, you can store your data in an Eigen array, and Armadillo vector, `std::array`, and for many of the routines, a `std::forward_list`. These routines are usable in float, double, long double, and Boost.Multiprecision precision, as well as their complex extensions whenever the computation is well-defined.

Absolute Gini Coefficient

The Gini coefficient, first used to measure wealth inequality, is also one of the best measures of the sparsity of an expansion in a basis. A sparse expansion has most of its norm concentrated in just a few coefficients, making the connection with wealth inequality obvious. See Hurley and Rickard for details. However, for measuring sparsity, the phase of the numbers is irrelevant, so we provide the `absolute_gini_coefficient`:

```
using boost::math::tools::sample_absolute_gini_coefficient;
using boost::math::tools::absolute_gini_coefficient;
std::vector<std::complex<double>> v{{0,1}, {0,0}, {0,0}, {0,0}};
double abs_gini = sample_absolute_gini_coefficient(v);
// now abs_gini = 1; maximally unequal

std::vector<std::complex<double>> w{{0,1}, {1,0}, {0,-1}, {-1,0}};
abs_gini = absolute_gini_coefficient(w);
// now abs_gini = 0; every element of the vector has equal magnitude

std::vector<double> u{-1, 1, -1};
abs_gini = absolute_gini_coefficient(u);
// now abs_gini = 0
// Alternative call useful for computing over subset of the input:
abs_gini = absolute_gini_coefficient(u.begin(), u.begin() + 1);
```

The sample Gini coefficient returns unity for a vector which has only one nonzero coefficient. The population Gini coefficient of a vector with one non-zero element is dependent on the length of the input.

The sample Gini coefficient lacks one desirable property of the population Gini coefficient, namely that "cloning" a vector has the same Gini coefficient; though cloning holds to very high accuracy with the sample Gini coefficient and can easily be recovered by a rescaling.

If sorting the input data is too much expense for a sparsity measure (is it going to be perfect anyway?), consider calculating the Hoyer sparsity instead.

Hoyer Sparsity

The Hoyer sparsity measures a normalized ratio of the ^{1} and ^{2} norms. As the name suggests, it is used to measure the sparsity of an expansion in some basis.

The Hoyer sparsity computes ($\sqrt{N}$ - ^{1}(v)/ ^{2}(v))/(√N -1). For details, see Hoyer as well as Hurley and Rickard.

A few special cases will serve to clarify the intended use: If *v* has only one nonzero coefficient, the Hoyer sparsity attains its maxima of 1. If the coefficients of *v* all have the same magnitude, then the Hoyer sparsity attains its minima of zero. If the elements of *v* are uniformly distributed on an interval [0, *b*], then the Hoyer sparsity is approximately 0.133.

Usage:

```
std::vector<Real> v{1,0,0};
Real hs = boost::math::tools::hoyer_sparsity(v);
// hs = 1
std::vector<Real> v{1,-1,1};
Real hs = boost::math::tools::hoyer_sparsity(v.begin(), v.end());
// hs = 0
```

The container must be forward iterable and the contents are not modified. Accepts real, complex, and integer inputs. If the input is an integral type, the output is a double precision float.

Oracle Signal-to-noise ratio

The function `oracle_snr` computes the ratio $s_2^2 / s - x_2^2$, where s is signal and x is a noisy signal. The function `oracle_snr_db` computes $10\log_{10}(s^2 / s - x^2)$. The functions are so named because in general, one does not know how to decompose a real signal x into $s + w$ and as such s is regarded as oracle information. Hence this function is mainly useful for unit testing other SNR estimators.

Usage:

```
std::vector<double> signal(500, 3.2);
std::vector<double> noisy_signal(500);
// fill 'noisy_signal' signal + noise
double snr_db = boost::math::tools::oracle_snr_db(signal, noisy_signal);
double snr = boost::math::tools::oracle_snr(signal, noisy_signal);
```

The input can be real, complex, or integral. Integral inputs produce double precision floating point outputs. The input data is not modified and must satisfy the requirements of a `RandomAccessContainer`.

M_2M_4 SNR Estimation

Estimates the SNR of a noisy signal via the M_2M_4 method. See Pauluzzi and N.C. Beaulieu and Matzner and Englberger for details.

```
std::vector<double> noisy_signal(512);
// fill noisy_signal with data contaminated by Gaussian white noise:
double est_snr_db = boost::math::tools::m2m4_snr_estimator_db(noisy_signal);
```

The M_2M_4 SNR estimator is an "in-service" estimator, meaning that the estimate is made using the noisy, data-bearing signal, and does not require a background estimate. This estimator has been found to be work best between roughly -3 and 15db, tending to overestimate the noise below -3db, and underestimate the noise above 15db. See Xue et al for details.

The M_2M_4 SNR estimator, by default, assumes that the kurtosis of the signal is 1 and the kurtosis of the noise is 3, the latter corresponding to Gaussian noise. These parameters, however, can be overridden:

```
std::vector<double> noisy_signal(512);
// fill noisy_signal with the data:
double signal_kurtosis = 1.5;
// Noise is assumed to follow Laplace distribution, which has kurtosis of 6:
double noise_kurtosis = 6;
double est_snr = boost::math::tools::m2m4_snr_estimator_db(noisy_signal, signal_kurtos↲
is, noise_kurtosis);
```

Now, technically the method is a "blind SNR estimator", meaning that the no *a-priori* information about the signal is required to use the method. However, the performance of the method is *vastly* better if you can come up with a better estimate of the signal and noise kurtosis. How can we do this? Suppose we know that the SNR is much greater than 1. Then we can estimate the signal kurtosis simply by using the noisy signal kurtosis. If the SNR is much less than one, this method breaks down as the noisy signal kurtosis will tend to the noise kurtosis-though in this limit we have an excellent estimator of the noise kurtosis! In addition, if you have a model of what your signal should look like, you can precompute the signal kurtosis. For example, sinusoids have a kurtosis of 1.5. See here for a study which uses estimates of this sort to improve the performance of the M_2M_4 estimator.

Nota bene: The traditional definition of SNR is *not* mean invariant. By this we mean that if a constant is added to every sample of a signal, the SNR is changed. For example, adding DC bias to a signal changes its SNR. For most use cases, this is really not what you intend; for example a signal consisting of zeros plus Gaussian noise has an SNR of zero, whereas a signal with a constant DC bias and random Gaussian noise might have a very large SNR.

The M_2M_4 SNR estimator is computed from mean-invariant quantities, and hence it should really be compared to the mean-invariant SNR.

Nota bene: This computation requires the solution of a system of quadratic equations involving the noise kurtosis, the signal kurtosis, and the second and fourth moments of the data. There is no guarantee that a solution of this system exists for all value of these parameters, in fact nonexistence can easily be demonstrated for certain data. If there is no solution to the system, then failure is communicated by returning NaNs. This happens distressingly often; if a user is aware of any blind SNR estimators which do not suffer from this drawback, please open a github ticket and let us know.

The author has not managed to fully characterize the conditions under which a real solution with $S > 0$ and $N > 0$ exists. However, a very intuitive example demonstrates why nonexistence can occur. Suppose the signal and noise kurtosis are equal. Then the method has no way to distinguish between the signal and the noise, and the solution is non-unique.

References

- Mallat, Stephane. *A wavelet tour of signal processing: the sparse way.* Academic press, 2008.
- Hurley, Niall, and Scott Rickard. *Comparing measures of sparsity.* IEEE Transactions on Information Theory 55.10 (2009): 4723-4741.
- Jensen, Arne, and Anders la Cour-Harbo. *Ripples in mathematics: the discrete wavelet transform.* Springer Science & Business Media, 2001.
- D. R. Pauluzzi and N. C. Beaulieu, *A comparison of SNR estimation techniques for the AWGN channel,* IEEE Trans. Communications, Vol. 48, No. 10, pp. 1681-1691, 2000.
- Hoyer, Patrik O. *Non-negative matrix factorization with sparseness constraints.*, Journal of machine learning research 5.Nov (2004): 1457-1469.

Norms

Synopsis

```
#include <boost/math/tools/norms.hpp>

namespace boost{ namespace math{ namespace tools {

    template<class Container>
    auto l0_pseudo_norm(Container const & c);

    template<class ForwardIterator>
    auto l0_pseudo_norm(ForwardIterator first, ForwardIterator last);

    template<class ForwardIterator>
    size_t hamming_distance(ForwardIterator first1, ForwardIterator last1, ForwardIterat↵
or first2);

    template<class Container>
    size_t hamming_distance(Container const & u, Container const & v);

    template<class Container>
    auto l1_norm(Container const & c);

    template<class ForwardIterator>
    auto l1_norm(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto l1_distance(Container const & v1, Container const & v2);

    template<class ForwardIterator>
    auto l1_distance(ForwardIterator first1, ForwardIterator last1, ForwardIterator first2);

    template<class Container>
    auto l2_norm(Container const & c);

    template<class ForwardIterator>
    auto l2_norm(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto l2_distance(Container const & v1, Container const & v2);

    template<class ForwardIterator>
    auto l2_distance(ForwardIterator first1, ForwardIterator last1, ForwardIterator first2);

    template<class Container>
    auto sup_norm(Container const & c);

    template<class ForwardIterator>
    auto sup_norm(ForwardIterator first, ForwardIterator last);

    template<class Container>
    auto sup_distance(Container const & v1, Container const & v2);

    template<class ForwardIterator>
    auto sup_distance(ForwardIterator first1, ForwardIterator last1, ForwardIterator first2);

    template<class Container>
    auto lp_norm(Container const & c, unsigned p);

    template<class ForwardIterator>
    auto lp_norm(ForwardIterator first, ForwardIterator last, unsigned p);

    template<class Container>
    auto lp_distance(Container const & v1, Container const & v2, unsigned p);

    template<class ForwardIterator>
   auto lp_distance(ForwardIterator first1, ForwardIterator last1, ForwardIterator first2, un↲
signed p);

    template<class Container>
    auto total_variation(Container const & c);

    template<class ForwardIterator>
    auto total_variation(ForwardIterator first, ForwardIterator last);

}}}
```

Description

The file `boost/math/tools/norms.hpp` is a set of facilities for computing scalar values traditionally useful in numerical analysis from vectors.

Our examples use `std::vector<double>` to hold the data, but this not required. In general, you can store your data in an Eigen array, an Armadillo vector, `std::array`, and for many of the routines, a `std::forward_list`. These routines are usable in float, double, long double, and Boost.Multiprecision precision, as well as their complex extensions whenever the computation is well-defined. Integral datatypes are supported for most routines.

∞ norm

Computes the supremum norm of a dataset:

```
std::vector<double> v{-3, 2, 1};
double sup = boost::math::tools::sup_norm(v.cbegin(), v.cend());
// sup = 3

std::vector<std::complex<double>> v{{0, -8}, {1,1}, {-3,2}};
// Range call:
double sup = boost::math::tools::sup_norm(v);
// sup = 8
```

Supports real, integral, and complex arithmetic. Container must be forward iterable and is not modified.

∞ distance

Computes the supremum norm distance between two vectors:

```
std::vector<double> v{-3, 2, 1};
std::vector<double> w{6, -2, 1};
double sup = boost::math::tools::sup_distance(w, v);
// sup = 9
```

Supports real, integral, and complex arithmetic. Container must be forward iterable and is not modified. If the input it integral, the output is a double precision float.

^{p} norm

```
std::vector<double> v{-8, 0, 0};
double sup = boost::math::tools::lp_norm(v.cbegin(), v.cend(), 7);
// sup = 8

std::vector<std::complex<double>> v{{1, 0}, {0,1}, {0,-1}};
double sup = boost::math::tools::lp_norm(v.cbegin(), v.cend(), 3);
// sup = cbrt(3)
```

Supports both real, integral, and complex arithmetic. If the input is integral, the output is a double precision float. The container must be forward iterable and the contents are not modified.

Only supports integral *p* for two reasons: The computation is much slower for real *p*, and the non-integral ^{p} norm is rarely used.

^{p} distance

```
std::vector<double> v{-8, 0, 0};
std::vector<double> w{8, 0, 0};
double dist = boost::math::tools::lp_distance(v, w, 7);
// dist = 16

std::vector<std::complex<double>> v{{1, 0}, {0,1}, {0,-1}};
double dist = boost::math::tools::lp_distance(v, v, 3);
// dist = 0
```

Supports both real, integral, and complex arithmetic. If the input is integral, the output is a double precision float. The container must be forward iterable and the contents are not modified.

Only supports integer *p*.

^{0} pseudo-norm

Counts the number of non-zero elements in a container.

```
std::vector<double> v{0,0,1};
size_t count = boost::math::tools::l0_pseudo_norm(v.begin(), v.end());
// count = 1
```

Supports real, integral, and complex numbers. The container must be forward iterable and the contents are not modified. Note that this measure is not robust against numerical noise and is therefore not as useful as (say) the Hoyer sparsity in numerical applications. Works with real, complex, and integral inputs.

Hamming Distance

Compute the number of non-equal elements between two vectors *w* and *v*:

```
std::vector<double> v{0,0,1};
std::vector<double> w{1,0,0};
size_t count = boost::math::tools::hamming_distance(w, v);
// count = 2
```

Works for any datatype for which the operator `!=` is defined.

^{1} norm

The ^{1} norm is a special case of the ^{p} norm, but is much faster:

```
std::vector<double> v{1,1,1};
double l1 = boost::math::tools::l1_norm(v.begin(), v.end());
// l1 = 3
```

Requires a forward iterable input, does not modify input data, and works with real, integral, and complex numbers.

^{1} distance

Computes the ^{1} distance between two vectors:

```
std::vector<double> v{1,1,1};
std::vector<double> w{1,1,1};
double dist = boost::math::tools::l1_distance(w, v);
// dist = 0
```

Requires a forward iterable inputs, does not modify input data, and works with real, integral, and complex numbers. If the input type is integral, the output is a double precision float.

^{2} norm

The ^{2} norm is again a special case of the ^{p} norm, but is much faster:

```
std::vector<double> v{1,1,1};
double l2 = boost::math::tools::l2_norm(v.begin(), v.end());
// l2 = sqrt(3)
```

Requires a forward iterable input, does not modify input data, and works with real, complex and integral data. If the input is integral, the output is a double precision float.

^{2} distance

Compute the ^{2} distance between two vectors *w* and *v*:

```
std::vector<double> v{1,1,1};
std::vector<double> w{1,2,1};
double dist = boost::math::tools::l2_distance(w, v);
// dist = 1
```

Requires a forward iterable input, does not modify input data, and works with real, complex numbers, and integral data. If the input type is integral, the output is a double precision float.

Total Variation

```
std::vector<double> v{1,1,1};
double tv = boost::math::tools::total_variation(v.begin(), v.end());
// no variation in v, so tv = 0.
v = {0,1};
double tv = boost::math::tools::total_variation(v.begin(), v.end());
// variation is 1, so tv = 1.
std::vector<int> v{1,1,1};
double tv = boost::math::tools::total_variation(v);
```

The total variation only supports real numbers and integers. If the input is integral, the output is a double precision float.

All the constituent operations to compute the total variation are well-defined for complex numbers, but the computed result is not meaningful; a 2D total variation is more appropriate. The container must be forward iterable, and the contents are not modified.

As an aside, the total variation is not technically a norm, since *TV(v) = 0* does not imply *v = 0*. However, it satisfies the triangle inequality and is absolutely 1-homogeneous, so it is a seminorm, and hence is grouped with the other norms here.

References

- Higham, Nicholas J. *Accuracy and stability of numerical algorithms.* Vol. 80. Siam, 2002.
- Mallat, Stephane. *A wavelet tour of signal processing: the sparse way.* Academic press, 2008.
- Hurley, Niall, and Scott Rickard. *Comparing measures of sparsity.* IEEE Transactions on Information Theory 55.10 (2009): 4723-4741.

Special Functions

Number Series

Bernoulli Numbers

Bernoulli numbers are a sequence of rational numbers useful for the Taylor series expansion, Euler-Maclaurin formula, and the Riemann zeta function.

Bernoulli numbers are used in evaluation of some Boost.Math functions, including the tgamma, lgamma and polygamma functions.

Single Bernoulli number

Synopsis

```
#include <boost/math/special_functions/bernoulli.hpp>
```

```
namespace boost { namespace math {

template <class T>
T bernoulli_b2n(const int n);  // Single Bernoulli number (default policy).

template <class T, class Policy>
T bernoulli_b2n(const int n, const Policy &pol); // User policy for errors etc.

}} // namespaces
```

Description

Both return the (2 * n)th Bernoulli number B_{2n}.

Note that since all odd numbered Bernoulli numbers are zero (apart from B_1 which is -½) the interface will only return the even numbered Bernoulli numbers.

This function uses fast table lookup for low-indexed Bernoulli numbers, while larger values are calculated as needed and then cached. The caching mechanism requires a certain amount of thread safety code, so `unchecked_bernoulli_b2n` may provide a better interface for performance critical code.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use, etc.

Refer to Policies for more details.

Examples

A simple example computes the value of B_4 where the return type is `double`, note that the argument to bernoulli_b2n is *2* not *4* since it computes B_{2N}.

```
try
{ // It is always wise to use try'n'catch blocks around Boost.Math functions
  // so that any informative error messages can be displayed in the catch block.
std::cout
  << std::setprecision(std::numeric_limits<double>::digits10)
  << boost::math::bernoulli_b2n<double>(2) << std::endl;
```

So B_4 == -1/30 == -0.0333333333333333

If we use Boost.Multiprecision and its 50 decimal digit floating-point type `cpp_dec_float_50`, we can calculate the value of much larger numbers like B_{200} and also obtain much higher precision.

```
std::cout
  << std::setprecision(std::numeric_limits<boost::multiprecision::cpp_dec_float_50>::digits10)
  << boost::math::bernoulli_b2n<boost::multiprecision::cpp_dec_float_50>(100) << std::endl;
```

```
-3.6470772645191354362138308865549944904868234686191e+215
```

Single (unchecked) Bernoulli number

Synopsis

```
#include <boost/math/special_functions/bernoulli.hpp>
```

```
template <>
struct max_bernoulli_b2n<T>;

template<class T>
inline T unchecked_bernoulli_b2n(unsigned n);
```

`unchecked_bernoulli_b2n` provides access to Bernoulli numbers **without any checks for overflow or invalid parameters**. It is implemented as a direct (and very fast) table lookup, and while not recommended for general use it can be useful inside inner loops where the ultimate performance is required, and error checking is moved outside the loop.

The largest value you can pass to `unchecked_bernoulli_b2n<>` is `max_bernoulli_b2n<>::value`: passing values greater than that will result in a buffer overrun error, so it's clearly important to place the error handling in your own code when using this direct interface.

The value of `boost::math::max_bernoulli_b2n<T>::value` varies by the type T, for types `float`/`double`/`long double` it's the largest value which doesn't overflow the target type: for example, `boost::math::max_bernoulli_b2n<double>::value` is 129. However, for multiprecision types, it's the largest value for which the result can be represented as the ratio of two 64-bit integers, for example `boost::math::max_bernoulli_b2n<boost::multiprecision::cpp_dec_float_50>::value` is just 17. Of course larger indexes can be passed to `bernoulli_b2n<T>(n)`, but then you lose fast table lookup (i.e. values may need to be calculated).

```
/*For example:
*/
   std::cout << "boost::math::max_bernoulli_b2n<float>::value = " ↵
 << boost::math::max_bernoulli_b2n<float>::value << std::endl;
   std::cout << "Maximum Bernoulli number using float is ↵
" << boost::math::bernoulli_b2n<float>( boost::math::max_bernoulli_b2n<float>::value) << std::endl;
   std::cout << "boost::math::max_bernoulli_b2n<double>::value = " ↵
 << boost::math::max_bernoulli_b2n<double>::value << std::endl;
   std::cout << "Maximum Bernoulli number using double is ↵
" << boost::math::bernoulli_b2n<double>( boost::math::max_bernoulli_b2n<double>::value) << std::endl;
```

```
boost::math::max_bernoulli_b2n<float>::value = 32
Maximum Bernoulli number using float is -2.0938e+038
boost::math::max_bernoulli_b2n<double>::value = 129
Maximum Bernoulli number using double is 1.33528e+306
```

Multiple Bernoulli Numbers

Synopsis

```
#include <boost/math/special_functions/bernoulli.hpp>
```

```
namespace boost { namespace math {

// Multiple Bernoulli numbers (default policy).
template <class T, class OutputIterator>
OutputIterator bernoulli_b2n(
  int start_index,
  unsigned number_of_bernoullis_b2n,
  OutputIterator out_it);

// Multiple Bernoulli numbers (user policy).
template <class T, class OutputIterator, class Policy>
OutputIterator bernoulli_b2n(
  int start_index,
  unsigned number_of_bernoullis_b2n,
  OutputIterator out_it,
  const Policy& pol);
}} // namespaces
```

Description

Two versions of the Bernoulli number function are provided to compute multiple Bernoulli numbers with one call (one with default policy and the other allowing a user-defined policy).

These return a series of Bernoulli numbers:

$$B_{2*\text{start\_index}}, B_{2*(\text{start\_index}+1)}, \ldots, B_{2*(\text{start\_index}+\text{number\_of\_bernoullis\_b2n}-1)}$$

Examples

We can compute and save all the float-precision Bernoulli numbers from one call.

```
std::vector<float> bn; // Space for 32-bit `float` precision Bernoulli numbers.

// Start with Bernoulli number 0.
boost::math::bernoulli_b2n<float>(0, 32, std::back_inserter(bn)); // Fill vector with even ↲
Bernoulli numbers.

for(size_t i = 0; i < bn.size(); i++)
{ // Show vector of even Bernoulli numbers, showing all significant decimal digits.
    std::cout << std::setprecision(std::numeric_limits<float>::digits10)
        << i*2 << ' '
        << bn[i]
        << std::endl;
}
```

```
0 1
2 0.166667
4 -0.0333333
6 0.0238095
8 -0.0333333
10 0.0757576
12 -0.253114
14 1.16667
16 -7.09216
18 54.9712
20 -529.124
22 6192.12
24 -86580.3
26 1.42552e+006
28 -2.72982e+007
30 6.01581e+008
32 -1.51163e+010
34 4.29615e+011
36 -1.37117e+013
38 4.88332e+014
40 -1.92966e+016
42 8.41693e+017
44 -4.03381e+019
46 2.11507e+021
48 -1.20866e+023
50 7.50087e+024
52 -5.03878e+026
54 3.65288e+028
56 -2.84988e+030
58 2.38654e+032
60 -2.14e+034
62 2.0501e+036
```

Of course, for any floating-point type, there is a maximum Bernoulli number that can be computed before it overflows the exponent. By default policy, if we try to compute too high a Bernoulli number, an exception will be thrown.

```
try
{
  std::cout
  << std::setprecision(std::numeric_limits<float>::digits10)
  << "Bernoulli number " << 33 * 2 <<std::endl;

  std::cout << boost::math::bernoulli_b2n<float>(33) << std::endl;
}
catch (std::exception ex)
{
  std::cout << "Thrown Exception caught: " << ex.what() << std::endl;
}
```

and we will get a helpful error message (provided try'n'catch blocks are used).

```
Bernoulli number 66
Thrown Exception caught: Error in function boost::math::bernoulli_b2n<float>(n):
Overflow evaluating function at 33
```

The source of this example is at bernoulli_example.cpp

Accuracy

All the functions usually return values within one ULP (unit in the last place) for the floating-point type.

Implementation

The implementation details are in bernoulli_details.hpp and unchecked_bernoulli.hpp.

For `i <= max_bernoulli_index<T>::value` this is implemented by simple table lookup from a statically initialized table; for larger values of `i`, this is implemented by the Tangent Numbers algorithm as described in the paper: Fast Computation of Bernoulli, Tangent and Secant Numbers, Richard P. Brent and David Harvey, http://arxiv.org/pdf/1108.0286v3.pdf (2011).

Tangent (or Zag) numbers (an even alternating permutation number) are defined and their generating function is also given therein.

The relation of Tangent numbers with Bernoulli numbers B_i is given by Brent and Harvey's equation 14:

$$T_j = (-1)^{j-1} 2^{2j} (2^{2n} - 1) \frac{B_{2n}}{2n}$$

Their relation with Bernoulli numbers B_i are defined by

if i > 0 and i is even then

$$B_j = (-1)^{\frac{j}{2}-1} \frac{j T_{j/2}}{(4^j - 2^j)}$$

elseif i == 0 then $B_i = 1$
elseif i == 1 then $B_i = -1/2$
elseif i < 0 or i is odd then $B_i = 0$

Note that computed values are stored in a fixed-size table, access is thread safe via atomic operations (i.e. lock free programming), this imparts a much lower overhead on access to cached values than might otherwise be expected - typically for multiprecision types the cost of thread synchronisation is negligible, while for built in types this code is not normally executed anyway. For very large arguments which cannot be reasonably computed or stored in our cache, an asymptotic expansion due to Luschny is used:

$$\begin{aligned}\ln(B_n) &= (\tfrac{1}{2}+n)\ln(n) + (\tfrac{1}{2}-n)\ln(\pi) + (\tfrac{3}{2}-n)\ln(2) - R(n)\\ R(n) &= n(1 - \tfrac{1}{12}(1 - \tfrac{1}{30}(1 - \tfrac{2}{7}n^{-2})n^{-2})n^{-2})\end{aligned}$$

Tangent Numbers

Tangent numbers, also called a zag function. See also Tangent number.

The first few values are 1, 2, 16, 272, 7936, 353792, 22368256, 1903757312 ... (sequence A000182 in OEIS). They are called tangent numbers because they appear as numerators in the Maclaurin series of `tan(x)`.

Important: there are two competing definitions of Tangent numbers in common use (depending on whether you take the even or odd numbered values as non-zero), we use:

$$T_n = \frac{2^{2n}(2^{2n}-1)\,|\,B_{2n}\,|}{2n}$$

Which gives:

$$\tan(x) = \sum_{k=1}^{\infty} \frac{T_k}{(2k-1)!} x^{2k-1}$$

Tangent numbers are used in the computation of Bernoulli numbers, but are also made available here.

Synopsis

```
#include <boost/math/special_functions/detail/bernoulli.hpp>
```

```
template <class T>
T tangent_t2n(const int i);  // Single tangent number (default policy).

template <class T, class Policy>
T tangent_t2n(const int i, const Policy &pol); // Single tangent number (user policy).

// Multiple tangent numbers (default policy).
template <class T, class OutputIterator>
OutputIterator tangent_t2n(const int start_index,
                                    const unsigned number_of_tangent_t2n,
                                    OutputIterator out_it);

// Multiple tangent numbers (user policy).
template <class T, class OutputIterator, class Policy>
OutputIterator tangent_t2n(const int start_index,
                                    const unsigned number_of_tangent_t2n,
                                    OutputIterator out_it,
                                    const Policy& pol);
```

Examples

We can compute and save a few Tangent numbers.

```
std::vector<float> tn; // Space for some `float` precision Tangent numbers.

// Start with Bernoulli number 0.
boost::math::tangent_t2n<float>(1, 6, std::back_inserter(tn)); // Fill vector with even Tan↵
gent numbers.

for(size_t i = 0; i < tn.size(); i++)
{ // Show vector of even Tangent numbers, showing all significant decimal digits.
    std::cout << std::setprecision(std::numeric_limits<float>::digits10)
        << " "
        << tn[i];
}
std::cout << std::endl;
```

The output is:

```
1 2 16 272 7936 353792
```

The source of this example is at bernoulli_example.cpp

Implementation

Tangent numbers are calculated as intermediates in the calculation of the Bernoulli numbers: refer to the Bernoulli numbers documentation for details.

Prime Numbers

Synopsis

```
#include <boost/math/special_functions/prime.hpp>
```

```
namespace boost { namespace math {

template <class Policy>
constexpr boost::uint32_t prime(unsigned n, const Policy& pol);

constexpr boost::uint32_t prime(unsigned n);

static const unsigned max_prime = 10000;

}} // namespaces
```

Description

The function `prime` provides fast table lookup to the first 10000 prime numbers (starting from 2 as the zeroth prime: as 1 isn't terribly useful in practice). There are two function signatures one of which takes an optional Policy as the second parameter to control error handling.

The constant `max_prime` is the largest value you can pass to `prime` without incurring an error.

Passing a value greater than `max_prime` results in a domain_error being raised.

This function is `constexpr` only if the compiler supports C++14 constexpr functions.

Gamma Functions

Gamma

Synopsis

```
#include <boost/math/special_functions/gamma.hpp>
```

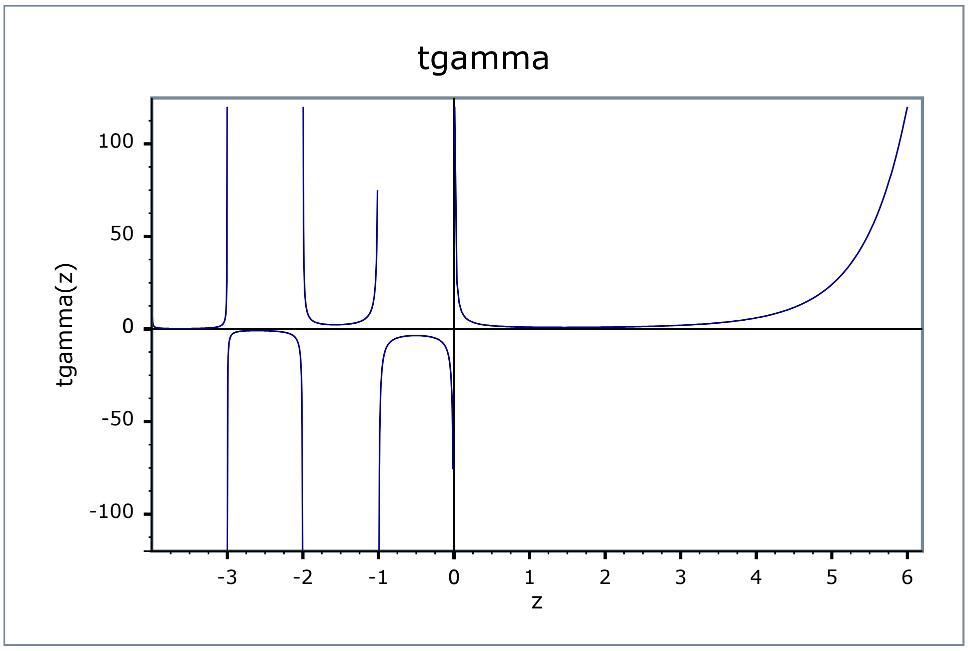
```
namespace boost{ namespace math{

template <class T>
calculated-result-type tgamma(T z);

template <class T, class Policy>
calculated-result-type tgamma(T z, const Policy&);

template <class T>
calculated-result-type tgamma1pm1(T dz);

template <class T, class Policy>
calculated-result-type tgamma1pm1(T dz, const Policy&);

}} // namespaces
```

Description

```
template <class T>
calculated-result-type tgamma(T z);

template <class T, class Policy>
calculated-result-type tgamma(T z, const Policy&);
```

Returns the "true gamma" (hence name tgamma) of value z:

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



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

There are effectively two versions of the tgamma function internally: a fully generic version that is slow, but reasonably accurate, and a much more efficient approximation that is used where the number of digits in the significand of T correspond to a certain Lanczos approximation. In practice any built in floating point type you will encounter has an appropriate Lanczos approximation defined for it. It is also possible, given enough machine time, to generate further Lanczos approximation's using the program libs/math/tools/lanczos_generator.cpp.

The return type of this function is computed using the *result type calculation rules*: the result is `double` when T is an integer type, and T otherwise.

```
template <class T>
calculated-result-type tgamma1pm1(T dz);

template <class T, class Policy>
calculated-result-type tgamma1pm1(T dz, const Policy&);
```

Returns `tgamma(dz + 1) - 1`. Internally the implementation does not make use of the addition and subtraction implied by the definition, leading to accurate results even for very small `dz`. However, the implementation is capped to either 35 digit accuracy, or to the precision of the Lanczos approximation associated with type T, whichever is more accurate.

The return type of this function is computed using the *result type calculation rules*: the result is `double` when T is an integer type, and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to other common libraries. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

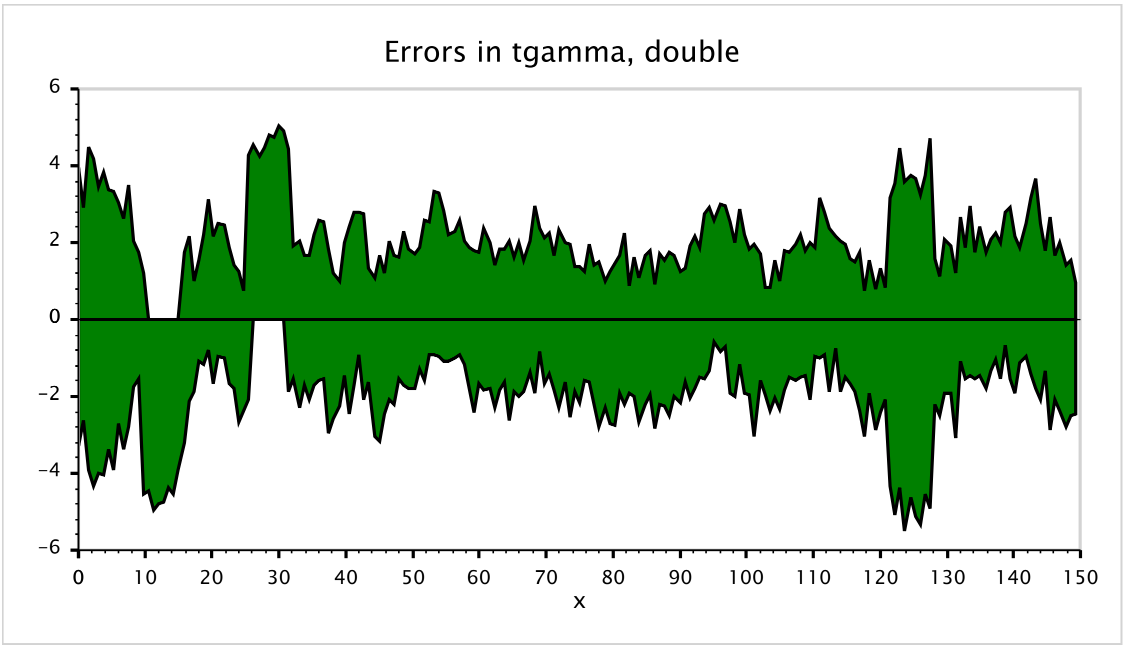
Table 24. Error rates for tgamma

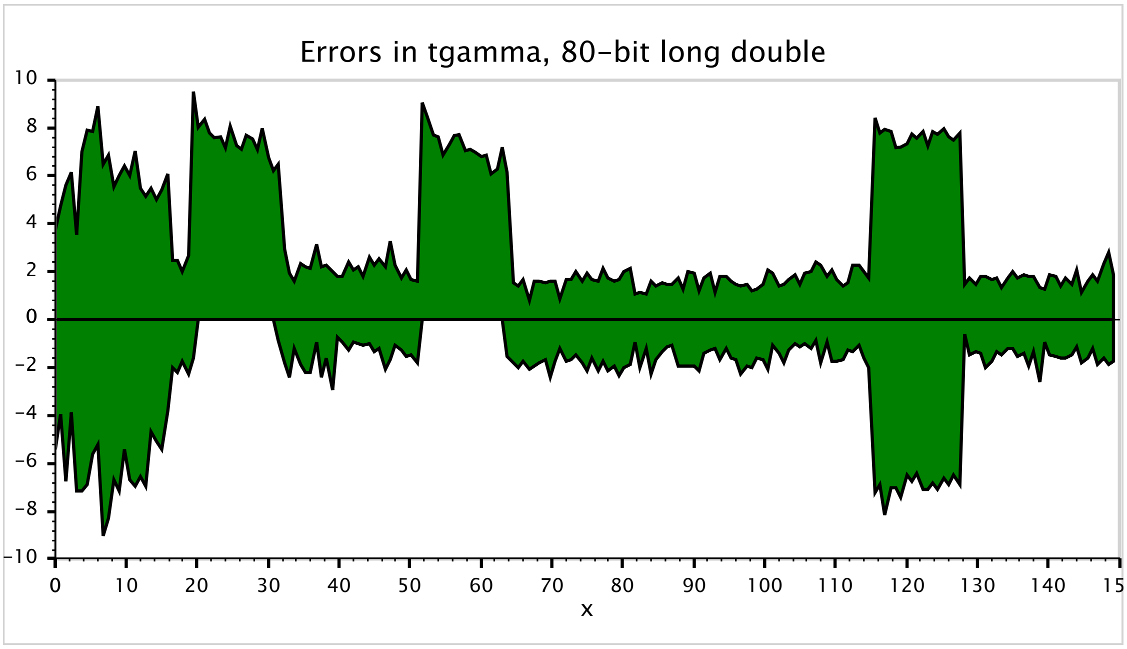
	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
factorials	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.95ε (Mean = 0.783ε)) (*Rmath 3.2.3:* Max = 314ε (Mean = 93.4ε))	Max = 2.67ε (Mean = 0.617ε) (*<cmath>:* Max = 1.66ε (Mean = 0.584ε)) (*<math.h>:* Max = 1.66ε (Mean = 0.584ε))	Max = 172ε (Mean = 41ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 1.85ε (Mean = 0.566ε) (*<math.h>:* Max = 3.17ε (Mean = 0.928ε))
near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.51ε (Mean = 1.92ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.335ε))	Max = 2ε (Mean = 0.608ε) (*<cmath>:* Max = 1ε (Mean = 0.376ε)) (*<math.h>:* Max = 1ε (Mean = 0.376ε))	Max = 2ε (Mean = 0.647ε) (*<math.h>:* Max = 0.5ε (Mean = 0.0791ε))	Max = 1.5ε (Mean = 0.635ε) (*<math.h>:* Max = 1ε (Mean = 0.405ε))
near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.41ε (Mean = 1.81ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.32ε))	Max = 2.51ε (Mean = 1.02ε) (*<cmath>:* Max = 0.918ε (Mean = 0.203ε)) (*<math.h>:* Max = 0.918ε (Mean = 0.203ε))	Max = 3.01ε (Mean = 1.06ε) (*<math.h>:* Max = 1ε (Mean = 0.175ε))	Max = 1.1ε (Mean = 0.59ε) (*<math.h>:* Max = 1ε (Mean = 0.4ε))
near 2	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.95ε (Mean = 3.12ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.191ε))	Max = 4.1ε (Mean = 1.55ε) (*<cmath>:* Max = 0.558ε (Mean = 0.298ε)) (*<math.h>:* Max = 0.558ε (Mean = 0.298ε))	Max = 5.01ε (Mean = 1.89ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 2ε (Mean = 0.733ε) (*<math.h>:* Max = 0ε (Mean = 0ε))
near -10	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.6ε (Mean = 1.05ε)) (*Rmath 3.2.3:* Max = 34.9ε (Mean = 9.2ε))	Max = 1.75ε (Mean = 0.895ε) (*<cmath>:* Max = 2.26ε (Mean = 1.08ε)) (*<math.h>:* Max = 2.26ε (Mean = 1.08ε))	Max = 1.75ε (Mean = 0.819ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 1.86ε (Mean = 0.881ε) (*<math.h>:* Max = 0.866ε (Mean = 0.445ε))
near -55	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.8ε (Mean = 0.782ε)) (*Rmath 3.2.3:* Max = 3.89e+04ε (Mean = 9.52e+03ε))	Max = 2.69ε (Mean = 1.09ε) (*<cmath>:* Max = 1.79ε (Mean = 0.75ε)) (*<math.h>:* Max = 1.79ε (Mean = 0.75ε))	Max = 98.5ε (Mean = 53.4ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 2.7ε (Mean = 1.35ε) (*<math.h>:* Max = 3.87e+04ε (Mean = 6.71e+03ε))

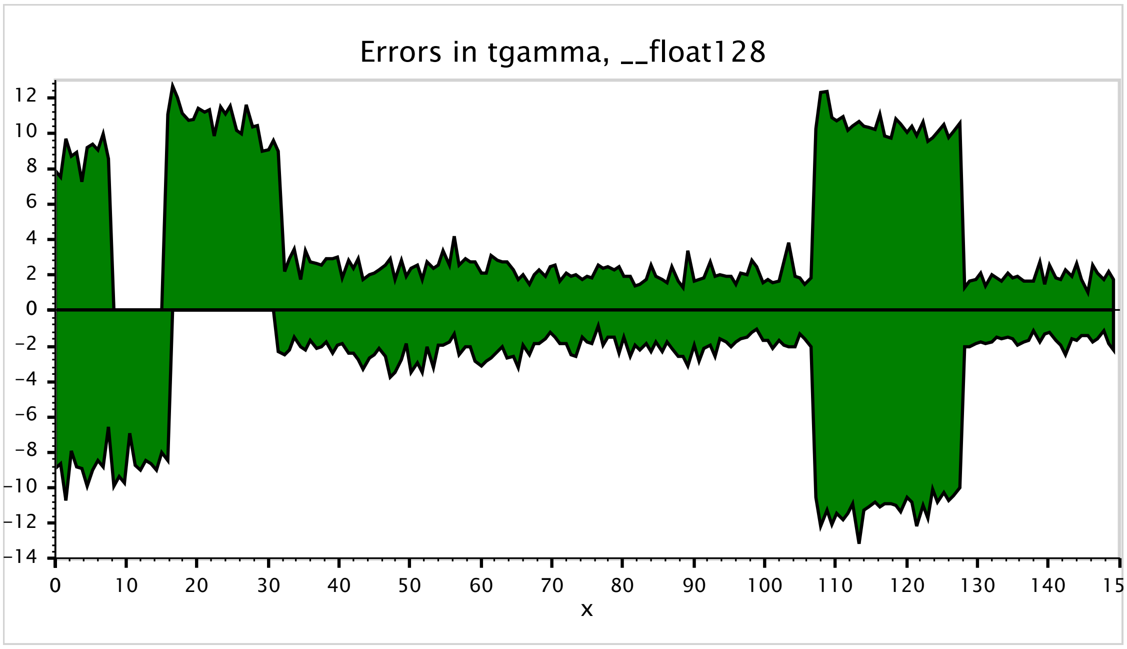
Table 25. Error rates for tgamma1pm1

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma1pm1(dz)	Max = 0ε (Mean = 0ε)	Max = 1.12ε (Mean = 0.49ε)	Max = 6.61ε (Mean = 0.84ε)	Max = 3.31ε (Mean = 0.517ε)

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The gamma is relatively easy to test: factorials and half-integer factorials can be calculated exactly by other means and compared with the gamma function. In addition, some accuracy tests in known tricky areas were computed at high precision using the generic version of this function.

The function `tgamma1pm1` is tested against values calculated very naively using the formula `tgamma(1+dz)-1` with a lanczos approximation accurate to around 100 decimal digits.

Implementation

The generic version of the `tgamma` function is implemented Sterling's approximation for lgamma for large z:

$$\ln \Gamma(z) \approx (z - \tfrac{1}{2})\ln(z) - z + \tfrac{1}{2}\ln(2\pi) + \sum_{k=1}^{\infty} \frac{B_{2k}}{2k(2k-1)z^{2k-1}}$$

Following exponentiation, downward recursion is then used for small values of z.

For types of known precision the Lanczos approximation is used, a traits class `boost::math::lanczos::lanczos_traits` maps type T to an appropriate approximation.

For z in the range -20 < z < 1 then recursion is used to shift to z > 1 via:

$$\Gamma(z) = \frac{\Gamma(z+1)}{z}$$

For very small z, this helps to preserve the identity:

$$\lim_{z \to 0}(\Gamma(z)) = \frac{1}{z}$$

For z < -20 the reflection formula:

$$\Gamma(-z) = -\frac{\pi}{\Gamma(z)\sin(\pi z)z}$$

is used. Particular care has to be taken to evaluate the `z * sin(π * z)` part: a special routine is used to reduce z prior to multiplying by π to ensure that the result in is the range [0, π/2]. Without this an excessive amount of error occurs in this region (which is hard enough already, as the rate of change near a negative pole is *exceptionally* high).

Finally if the argument is a small integer then table lookup of the factorial is used.

The function `tgamma1pm1` is implemented using rational approximations devised by JM in the region `-0.5 < dz < 2`. These are the same approximations (and internal routines) that are used for lgamma, and so aren't detailed further here. The result of the approximation is `log(tgamma(dz+1))` which can fed into expm1 to give the desired result. Outside the range `-0.5 < dz < 2` then the naive formula `tgamma1pm1(dz) = tgamma(dz+1)-1` can be used directly.

Log Gamma

Synopsis

```
#include <boost/math/special_functions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type lgamma(T z);

template <class T, class Policy>
calculated-result-type lgamma(T z, const Policy&);

template <class T>
calculated-result-type lgamma(T z, int* sign);

template <class T, class Policy>
calculated-result-type lgamma(T z, int* sign, const Policy&);

}} // namespaces
```

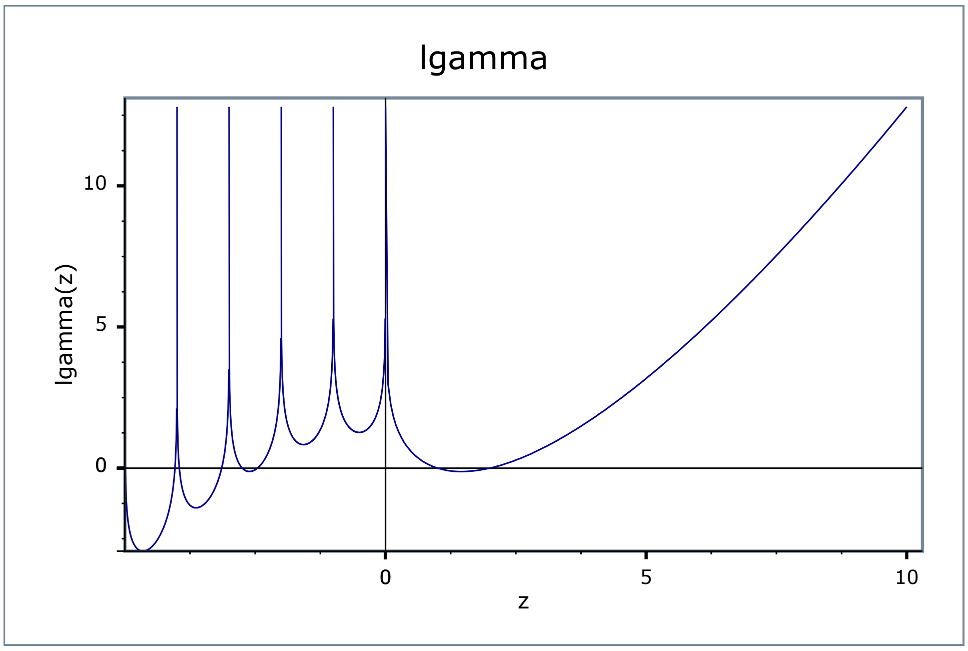
Description

The lgamma function is defined by:

$$\operatorname{lgamma}(z) = \ln|\Gamma(z)|$$

The second form of the function takes a pointer to an integer, which if non-null is set on output to the sign of tgamma(z).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.



There are effectively two versions of this function internally: a fully generic version that is slow, but reasonably accurate, and a much more efficient approximation that is used where the number of digits in the significand of T correspond to a certain Lanczos approximation. In practice, any built-in floating-point type you will encounter has an appropriate Lanczos approximation defined for it. It is also possible, given enough machine time, to generate further Lanczos approximation's using the program libs/math/tools/lanczos_generator.cpp.

The return type of these functions is computed using the *result type calculation rules*: the result is of type `double` if T is an integer type, or type T otherwise.

Accuracy

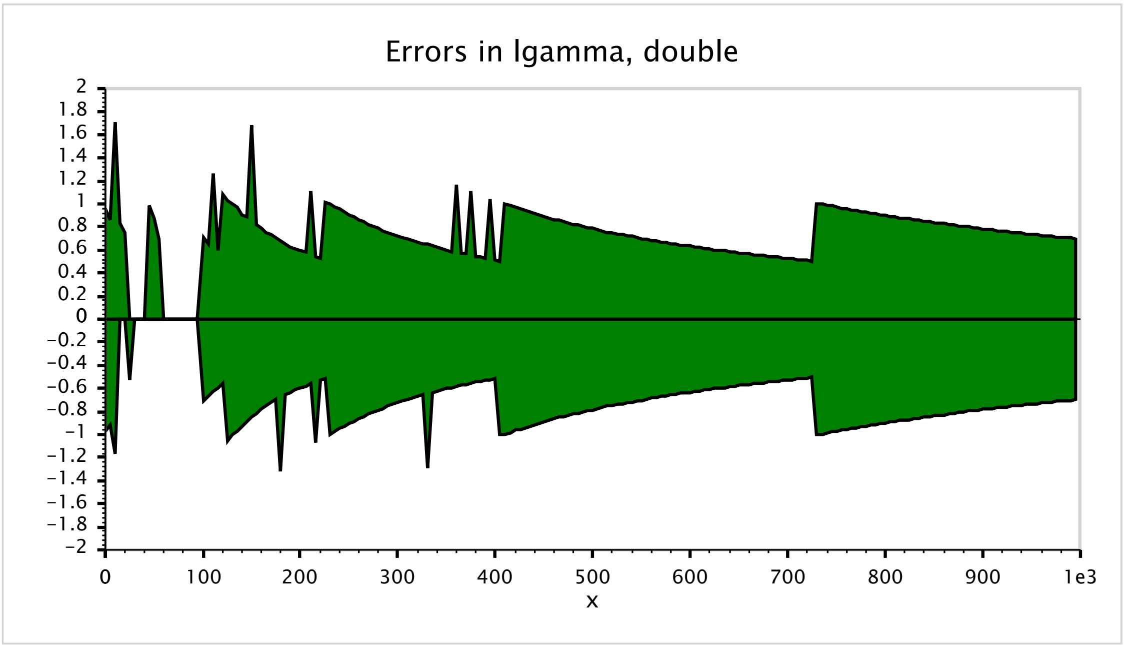
The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to various other libraries. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

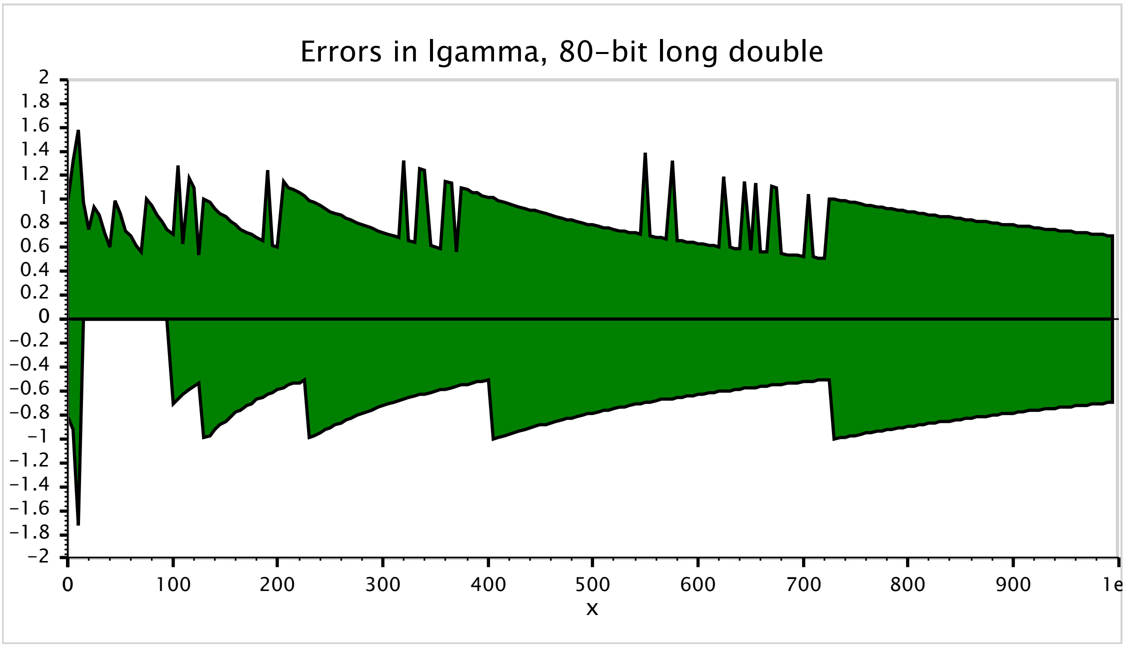
Note that while the relative errors near the positive roots of lgamma are very low, the lgamma function has an infinite number of irrational roots for negative arguments: very close to these negative roots only a low absolute error can be guaranteed.

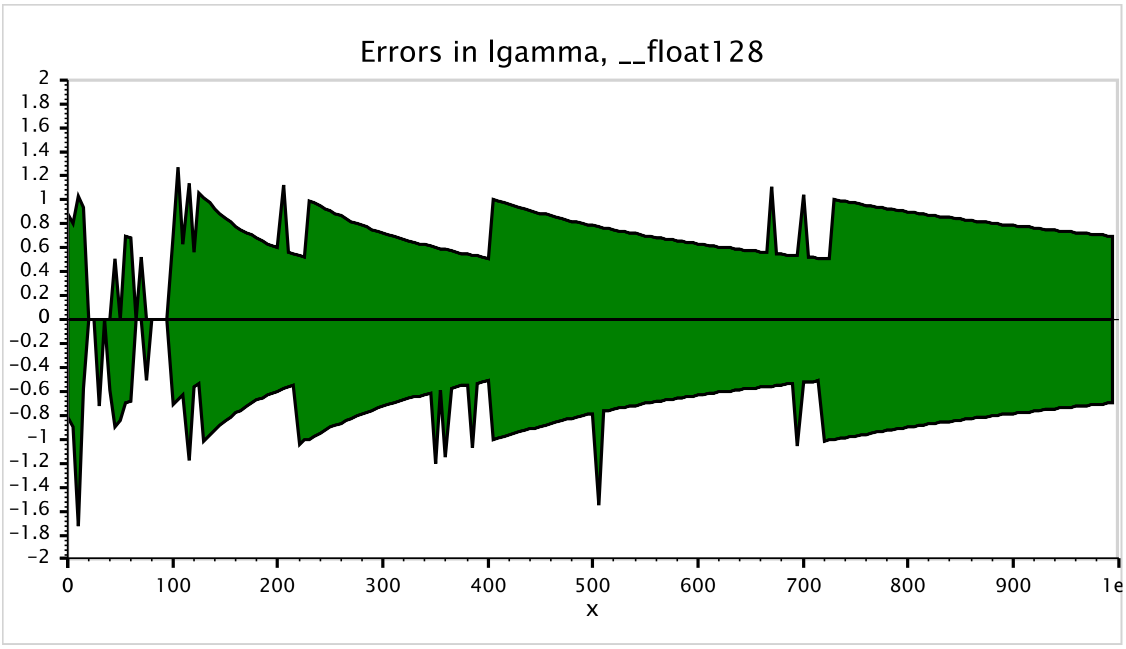
Table 26. Error rates for lgamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
factorials	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 33.6ε (Mean = 2.78ε)) (*Rmath 3.2.3:* Max = 1.55ε (Mean = 0.592ε))	Max = 0.991ε (Mean = 0.308ε) (*<cmath>:* Max = 1.67ε (Mean = 0.487ε)) (*<math.h>:* Max = 1.67ε (Mean = 0.487ε))	Max = 0.991ε (Mean = 0.383ε) (*<math.h>:* Max = 1.36ε (Mean = 0.476ε))	Max = 0.914ε (Mean = 0.175ε) (*<math.h>:* Max = 0.958ε (Mean = 0.38ε))
near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.21ε (Mean = 1.57ε)) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε))	Max = 1.42ε (Mean = 0.566ε) (*<cmath>:* Max = 0.964ε (Mean = 0.543ε)) (*<math.h>:* Max = 0.964ε (Mean = 0.543ε))	Max = 1.42ε (Mean = 0.566ε) (*<math.h>:* Max = 0.964ε (Mean = 0.543ε))	Max = 0.964ε (Mean = 0.462ε) (*<math.h>:* Max = 0.962ε (Mean = 0.372ε))
near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 442ε (Mean = 88.8ε)) (*Rmath 3.2.3:* Max = 7.99e+04ε (Mean = 1.68e+04ε))	Max = 0.948ε (Mean = 0.36ε) (*<cmath>:* Max = 0.615ε (Mean = 0.096ε)) (*<math.h>:* Max = 0.615ε (Mean = 0.096ε))	Max = 0.948ε (Mean = 0.36ε) (*<math.h>:* Max = 1.71ε (Mean = 0.581ε))	Max = 0.867ε (Mean = 0.468ε) (*<math.h>:* Max = 0.906ε (Mean = 0.565ε))
near 2	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.17e+03ε (Mean = 274ε)) (*Rmath 3.2.3:* Max = 2.63e+05ε (Mean = 5.84e+04ε))	Max = 0.878ε (Mean = 0.242ε) (*<cmath>:* Max = 0.741ε (Mean = 0.263ε)) (*<math.h>:* Max = 0.741ε (Mean = 0.263ε))	Max = 0.878ε (Mean = 0.242ε) (*<math.h>:* Max = 0.598ε (Mean = 0.235ε))	Max = 0.591ε (Mean = 0.159ε) (*<math.h>:* Max = 0.741ε (Mean = 0.473ε))
near -10	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 24.9ε (Mean = 4.6ε)) (*Rmath 3.2.3:* Max = 4.22ε (Mean = 1.26ε))	Max = 3.81ε (Mean = 1.01ε) (*<cmath>:* Max = 0.997ε (Mean = 0.412ε)) (*<math.h>:* Max = 0.997ε (Mean = 0.412ε))	Max = 3.81ε (Mean = 1.01ε) (*<math.h>:* Max = 3.04ε (Mean = 1.01ε))	Max = 4.22ε (Mean = 1.33ε) (*<math.h>:* Max = 0.997ε (Mean = 0.444ε))
near -55	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.02ε (Mean = 1.47ε)) (*Rmath 3.2.3:* Max = 250ε (Mean = 60.9ε))	Max = 0.821ε (Mean = 0.513ε) (*<cmath>:* Max = 1.58ε (Mean = 0.672ε)) (*<math.h>:* Max = 1.58ε (Mean = 0.672ε))	Max = 1.59ε (Mean = 0.587ε) (*<math.h>:* Max = 0.821ε (Mean = 0.674ε))	Max = 0.821ε (Mean = 0.419ε) (*<math.h>:* Max = 249ε (Mean = 43.1ε))

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The main tests for this function involve comparisons against the logs of the factorials which can be independently calculated to very high accuracy.

Random tests in key problem areas are also used.

Implementation

The generic version of this function is implemented using Sterling's approximation for large arguments:

$$\ln \Gamma\ (z) \approx (z - \tfrac{1}{2})\ln(z) - z + \tfrac{1}{2}\ln(2\ \pi\ ) + \sum_{k=1}^{\infty} \frac{B_{2k}}{2k(2k-1)z^{2k-1}}$$

For small arguments, the logarithm of tgamma is used.

For negative *z* the logarithm version of the reflection formula is used:

$$\ln(\ |\ \Gamma(-z)\ |\ ) = \ln(\pi) - \ln(\Gamma(z)) - \ln(z\sin(\pi z))$$

For types of known precision, the Lanczos approximation is used, a traits class `boost::math::lanczos::lanczos_traits` maps type T to an appropriate approximation. The logarithmic version of the Lanczos approximation is:

$$\ln(\ |\ \Gamma(z)\ |\ ) = (z - 0.5)\ln\left(\frac{z + g - 0.5}{e}\right) + \ln(L_{e,g}(z))$$

Where $L_{e,g}$ is the Lanczos sum, scaled by e^g.

As before the reflection formula is used for *z < 0*.

When z is very near 1 or 2, then the logarithmic version of the Lanczos approximation suffers very badly from cancellation error: indeed for values sufficiently close to 1 or 2, arbitrarily large relative errors can be obtained (even though the absolute error is tiny).

For types with up to 113 bits of precision (up to and including 128-bit long doubles), root-preserving rational approximations devised by JM are used over the intervals [1,2] and [2,3]. Over the interval [2,3] the approximation form used is:

```
lgamma(z) = (z-2)(z+1)(Y + R(z-2));
```

Where Y is a constant, and R(z-2) is the rational approximation: optimised so that it's absolute error is tiny compared to Y. In addition small values of z greater than 3 can handled by argument reduction using the recurrence relation:

```
lgamma(z+1) = log(z) + lgamma(z);
```

Over the interval [1,2] two approximations have to be used, one for small z uses:

```
lgamma(z) = (z-1)(z-2)(Y + R(z-1));
```

Once again Y is a constant, and R(z-1) is optimised for low absolute error compared to Y. For z > 1.5 the above form wouldn't converge to a minimax solution but this similar form does:

```
lgamma(z) = (2-z)(1-z)(Y + R(2-z));
```

Finally for z < 1 the recurrence relation can be used to move to z > 1:

```
lgamma(z) = lgamma(z+1) - log(z);
```

Note that while this involves a subtraction, it appears not to suffer from cancellation error: as z decreases from 1 the `-log(z)` term grows positive much more rapidly than the `lgamma(z+1)` term becomes negative. So in this specific case, significant digits are preserved, rather than cancelled.

For other types which do have a Lanczos approximation defined for them the current solution is as follows: imagine we balance the two terms in the Lanczos approximation by dividing the power term by its value at *z = 1*, and then multiplying the Lanczos coefficients by the same value. Now each term will take the value 1 at *z = 1* and we can rearrange the power terms in terms of log1p. Likewise if we subtract 1 from the Lanczos sum part (algebraically, by subtracting the value of each term at *z = 1*), we obtain a new summation that can be also be fed into log1p. Crucially, all of the terms tend to zero, as *z -> 1*:

$$\begin{aligned}\ln(\,|\,\Gamma(z)\,|\,) &= \Delta z\ln\left(\frac{\Delta z+g+0.5}{e}\right)+\frac{1}{2}\ln\left(1+\frac{\Delta z}{g+0.5}\right)+\ln\left(1+\sum_{k=0}^{N-1}\frac{\Delta z d_k}{k(\Delta z+k)}\right)\\ \Delta z &= 1-z \quad\wedge\quad d_k=\sqrt{\frac{g+0.5}{e}}\left(\frac{C_k}{e^g}\right)\end{aligned}$$

The C_k terms in the above are the same as in the Lanczos approximation.

A similar rearrangement can be performed at *z = 2*:

$$\begin{aligned}\ln(\,|\,\Gamma(z)\,|\,) &= \frac{3}{2}\ln\left(1+\frac{\Delta z}{g+1.5}\right)+\Delta z\ln\left(\frac{z+g-0.5}{e}\right)+\ln\left(1+\sum_{k=1}^{N-1}\frac{-d_k\Delta z}{z+kz+k^2-1}\right)\\ \Delta z &= z-2 \quad\wedge\quad d_k=\left(\frac{g+1.5}{e}\right)^{1.5}\left(\frac{C_k}{e^g}\right)\end{aligned}$$

Digamma

Synopsis

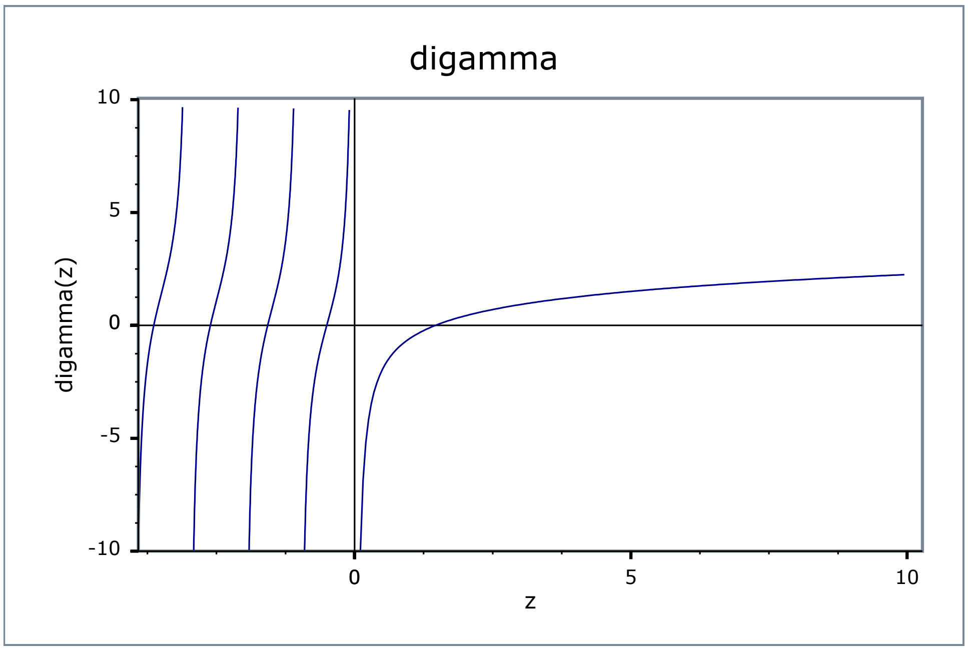
```
#include <boost/math/special_functions/digamma.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type digamma(T z);

template <class T, class Policy>
calculated-result-type digamma(T z, const Policy&);

}} // namespaces
```

Description

Returns the digamma or psi function of *x*. Digamma is defined as the logarithmic derivative of the gamma function:

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



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The return type of this function is computed using the *result type calculation rules*: the result is of type `double` when T is an integer type, and type T otherwise.

Accuracy

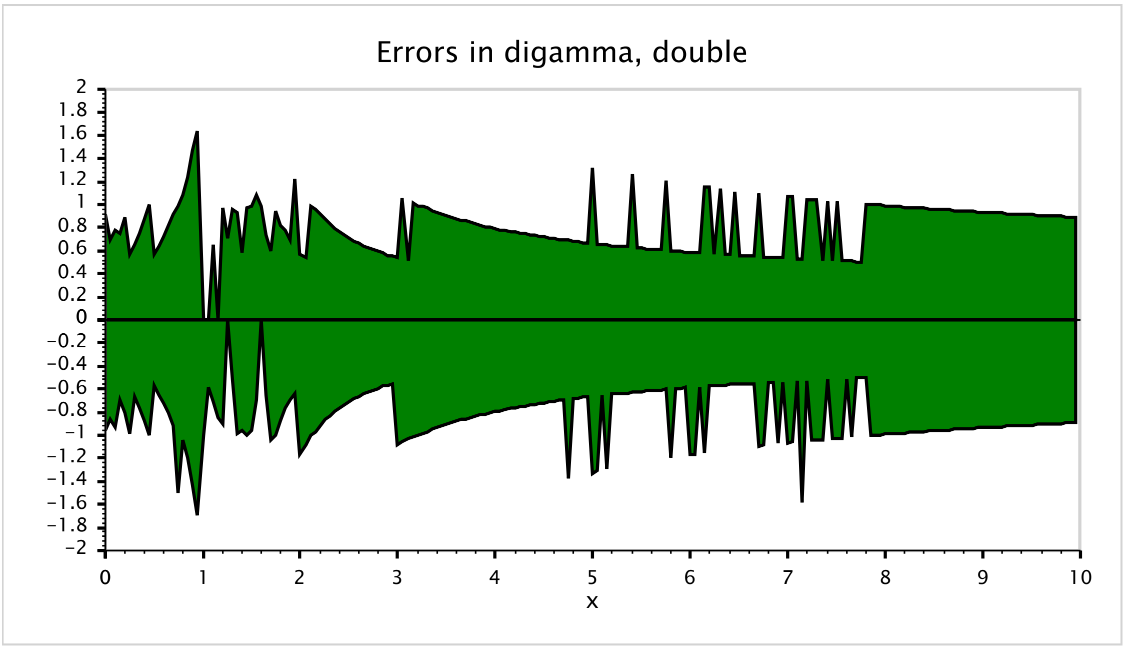
The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

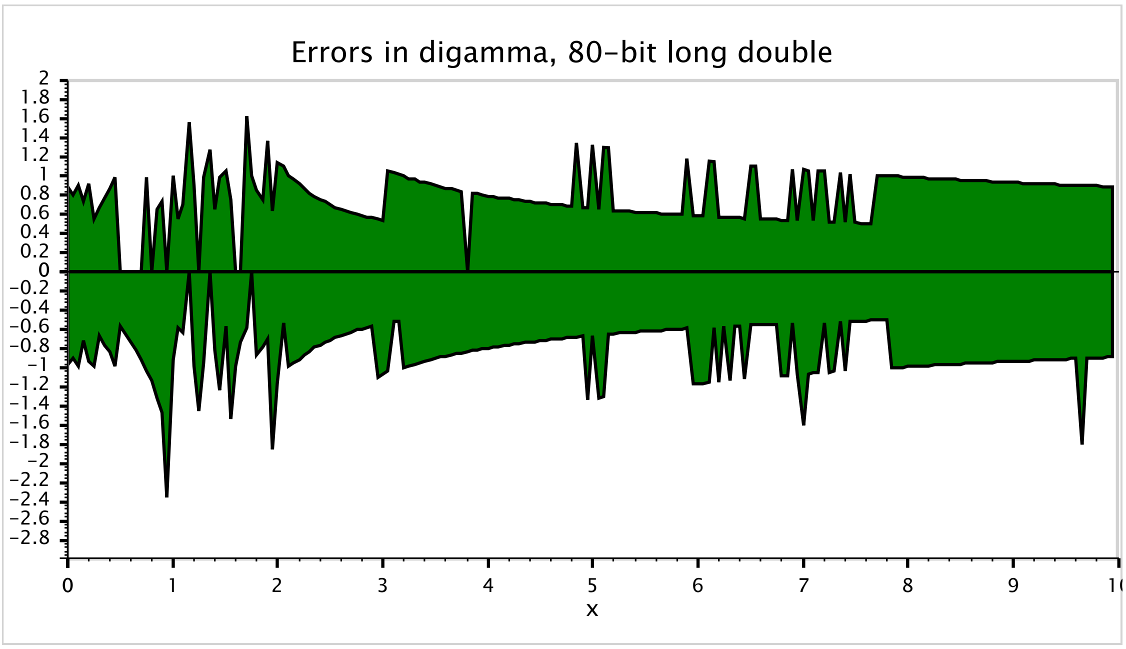
Table 27. Error rates for digamma

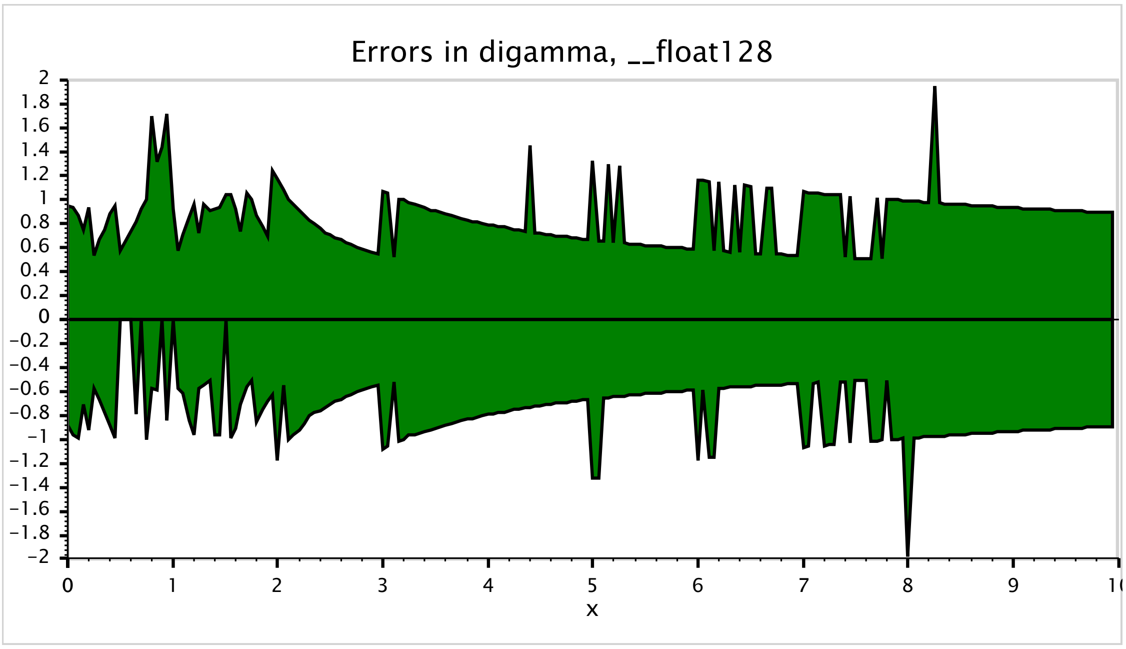
	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Digamma Function: Large Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.84ε (Mean = 0.71ε)) (*Rmath 3.2.3:* Max = 1.18ε (Mean = 0.331ε))	Max = 1.39ε (Mean = 0.413ε)	Max = 1.39ε (Mean = 0.413ε)	Max = 0.98ε (Mean = 0.369ε)
Digamma Function: Near the Positive Root	Max = 0.891ε (Mean = 0.0995ε) (*GSL 2.1:* Max = 135ε (Mean = 11.9ε)) (*Rmath 3.2.3:* Max = 2.02e+03ε (Mean = 256ε))	Max = 1.37ε (Mean = 0.477ε)	Max = 1.31ε (Mean = 0.471ε)	Max = 0.997ε (Mean = 0.527ε)
Digamma Function: Near Zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.953ε (Mean = 0.348ε)) (*Rmath 3.2.3:* Max = 1.17ε (Mean = 0.564ε))	Max = 0.984ε (Mean = 0.361ε)	Max = 0.984ε (Mean = 0.361ε)	Max = 0.953ε (Mean = 0.337ε)
Digamma Function: Negative Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.56e+04ε (Mean = 3.91e+03ε)) (*Rmath 3.2.3:* Max = 4.6e+04ε (Mean = 3.94e+03ε))	Max = 180ε (Mean = 13ε)	Max = 180ε (Mean = 13ε)	Max = 214ε (Mean = 16.1ε)
Digamma Function: Values near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.866ε (Mean = 0.387ε)) (*Rmath 3.2.3:* Max = 3.58e+05ε (Mean = 1.6e+05ε))	Max = 1ε (Mean = 0.592ε)	Max = 1ε (Mean = 0.592ε)	Max = 0ε (Mean = 0ε)
Digamma Function: Integer arguments	Max = 0.992ε (Mean = 0.215ε) (*GSL 2.1:* Max = 1.18ε (Mean = 0.607ε)) (*Rmath 3.2.3:* Max = 4.33ε (Mean = 0.982ε))	Max = 0.888ε (Mean = 0.403ε)	Max = 0.888ε (Mean = 0.403ε)	Max = 0.992ε (Mean = 0.452ε)
Digamma Function: Half integer arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.09ε (Mean = 0.531ε)) (*Rmath 3.2.3:* Max = 46.2ε (Mean = 7.24ε))	Max = 0.906ε (Mean = 0.409ε)	Max = 0.906ε (Mean = 0.409ε)	Max = 0.78ε (Mean = 0.314ε)

As shown above, error rates for positive arguments are generally very low. For negative arguments there are an infinite number of irrational roots: relative errors very close to these can be arbitrarily large, although absolute error will remain very low.

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

There are two sets of tests: spot values are computed using the online calculator at functions.wolfram.com, while random test values are generated using the high-precision reference implementation (a differentiated Lanczos approximation see below).

Implementation

The implementation is divided up into the following domains:

For Negative arguments the reflection formula:

```
digamma(1-x) = digamma(x) + pi/tan(pi*x);
```

is used to make *x* positive.

For arguments in the range [0,1] the recurrence relation:

```
digamma(x) = digamma(x+1) - 1/x
```

is used to shift the evaluation to [1,2].

For arguments in the range [1,2] a rational approximation devised by JM is used (see below).

For arguments in the range [2,BIG] the recurrence relation:

```
digamma(x+1) = digamma(x) + 1/x;
```

is used to shift the evaluation to the range [1,2].

For arguments > BIG the asymptotic expansion:

$$\psi(x) = \ln(x) + \frac{1}{2x} - \sum_{n=1}^{\infty} \frac{B_{2n}}{2nx^{2n}}$$

can be used. However, this expansion is divergent after a few terms: exactly how many terms depends on the size of *x*. Therefore the value of *BIG* must be chosen so that the series can be truncated at a term that is too small to have any effect on the result when evaluated at *BIG*. Choosing BIG=10 for up to 80-bit reals, and BIG=20 for 128-bit reals allows the series to truncated after a suitably small number of terms and evaluated as a polynomial in `1/(x*x)`.

The arbitrary precision version of this function uses recurrence relations until x > BIG, and then evaluation via the asymptotic expansion above. As special cases integer and half integer arguments are handled via:

$$\psi(n) = \sum_{k=1}^{n-1} \frac{1}{k} - \gamma \quad ; \quad n \in \mathbb{N}$$

$$\psi\left(\frac{1}{2} - n\right) = \sum_{k=1}^{n-1} \frac{1}{k} + \sum_{k=n}^{2n-1} \frac{2}{k} - \log(4) - \gamma \quad ; \quad n \in \mathbb{N}$$

The rational approximation devised by JM in the range [1,2] is derived as follows.

First a high precision approximation to digamma was constructed using a 60-term differentiated Lanczos approximation, the form used is:

$$\psi(x) \quad = \quad \frac{z - \frac{1}{2}}{x + g - \frac{1}{2}} + \ln\left(x + g - \frac{1}{2}\right) + \frac{P'(x)}{P(x)} - \frac{Q'(x)}{Q(x)} - 1$$

Where P(x) and Q(x) are the polynomials from the rational form of the Lanczos sum, and P'(x) and Q'(x) are their first derivatives. The Lanzos part of this approximation has a theoretical precision of ~100 decimal digits. However, cancellation in the above sum will reduce that to around `99-(1/y)` digits if *y* is the result. This approximation was used to calculate the positive root of digamma, and was found to agree with the value used by Cody to 25 digits (See Math. Comp. 27, 123-127 (1973) by Cody, Strecok and Thacher) and with the value used by Morris to 35 digits (See TOMS Algorithm 708).

Likewise a few spot tests agreed with values calculated using functions.wolfram.com to >40 digits. That's sufficiently precise to insure that the approximation below is accurate to double precision. Achieving 128-bit long double precision requires that the location of the root is known to ~70 digits, and it's not clear whether the value calculated by this method meets that requirement: the difficulty lies in independently verifying the value obtained.

The rational approximation devised by JM was optimised for absolute error using the form:

```
digamma(x) = (x - X0)(Y + R(x - 1));
```

Where X0 is the positive root of digamma, Y is a constant, and R(x - 1) is the rational approximation. Note that since X0 is irrational, we need twice as many digits in X0 as in x in order to avoid cancellation error during the subtraction (this assumes that *x* is an exact value, if it's not then all bets are off). That means that even when x is the value of the root rounded to the nearest representable value, the result of digamma(x) ***will not be zero***.

Trigamma

Synopsis

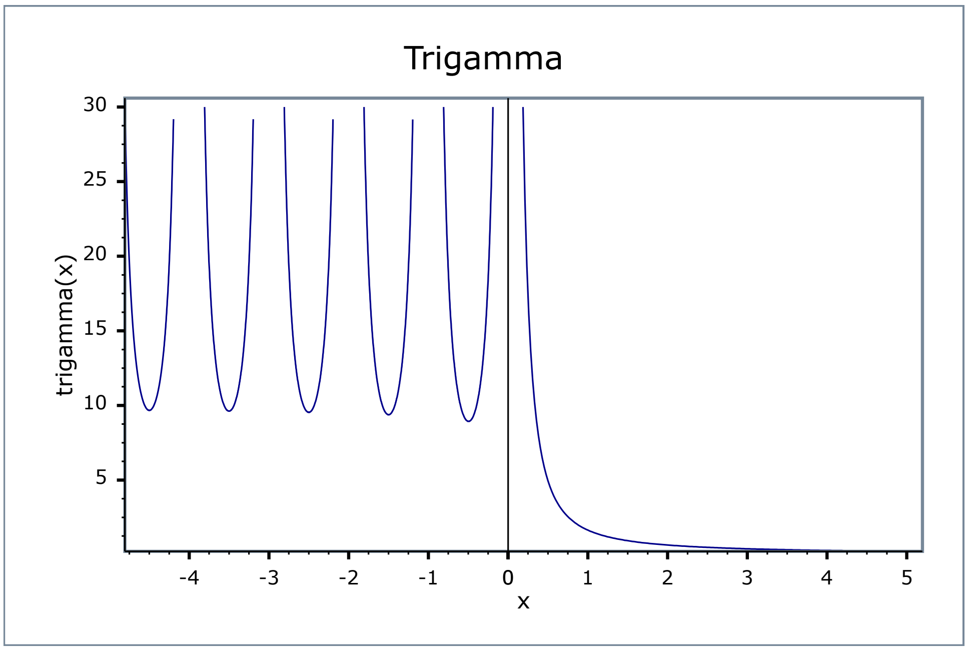
```
#include <boost/math/special_functions/trigamma.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type trigamma(T z);

template <class T, class Policy>
calculated-result-type trigamma(T z, const Policy&);

}} // namespaces
```

Description

Returns the trigamma function of *x*. Trigamma is defined as the derivative of the digamma function:

$$\psi^{(1)}(x) = \sum_{k=0}^{\infty} \frac{1}{(k+x)^2} = \frac{\partial \psi(x)}{\partial x}$$



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The return type of this function is computed using the *result type calculation rules*: the result is of type `double` when T is an integer type, and type T otherwise.

Accuracy

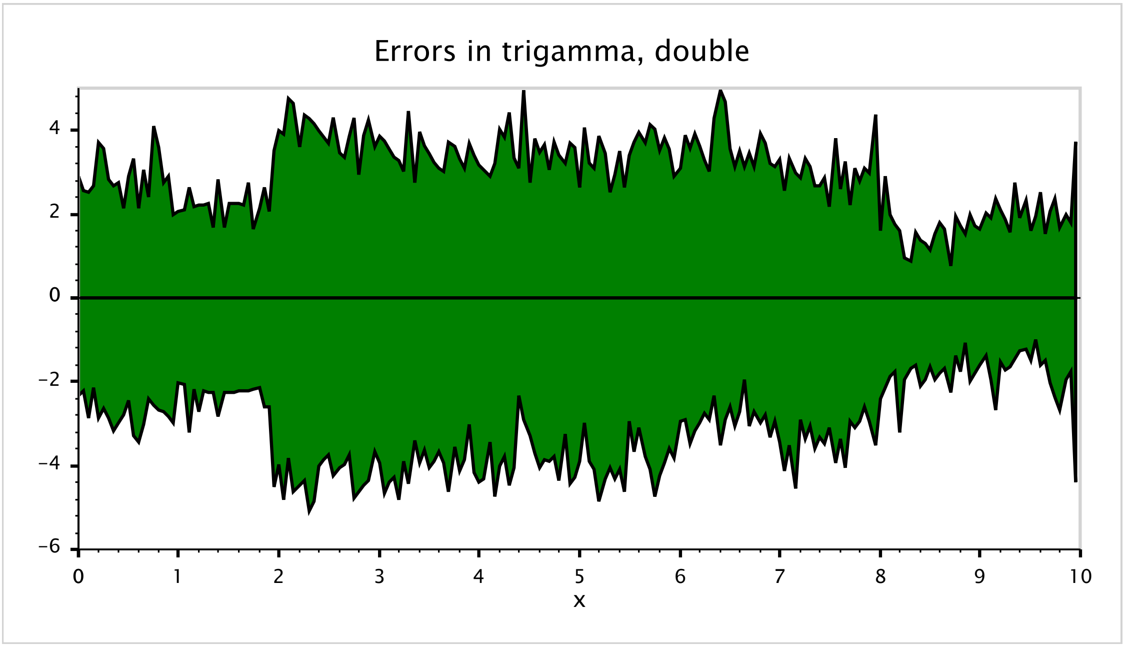
The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

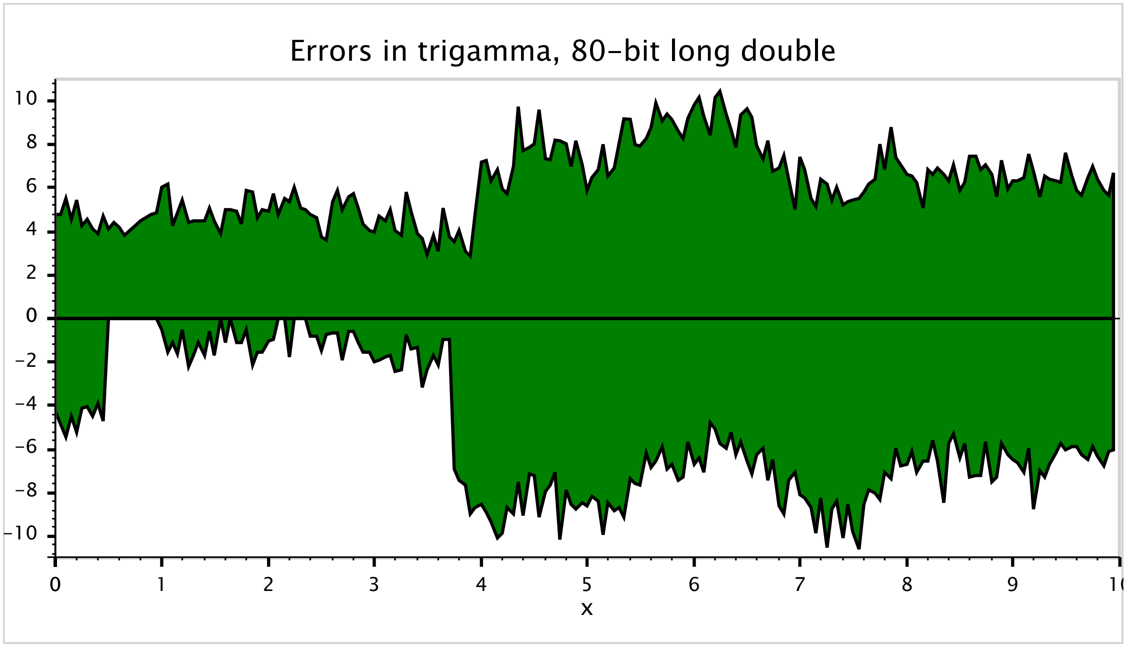
Table 28. Error rates for trigamma

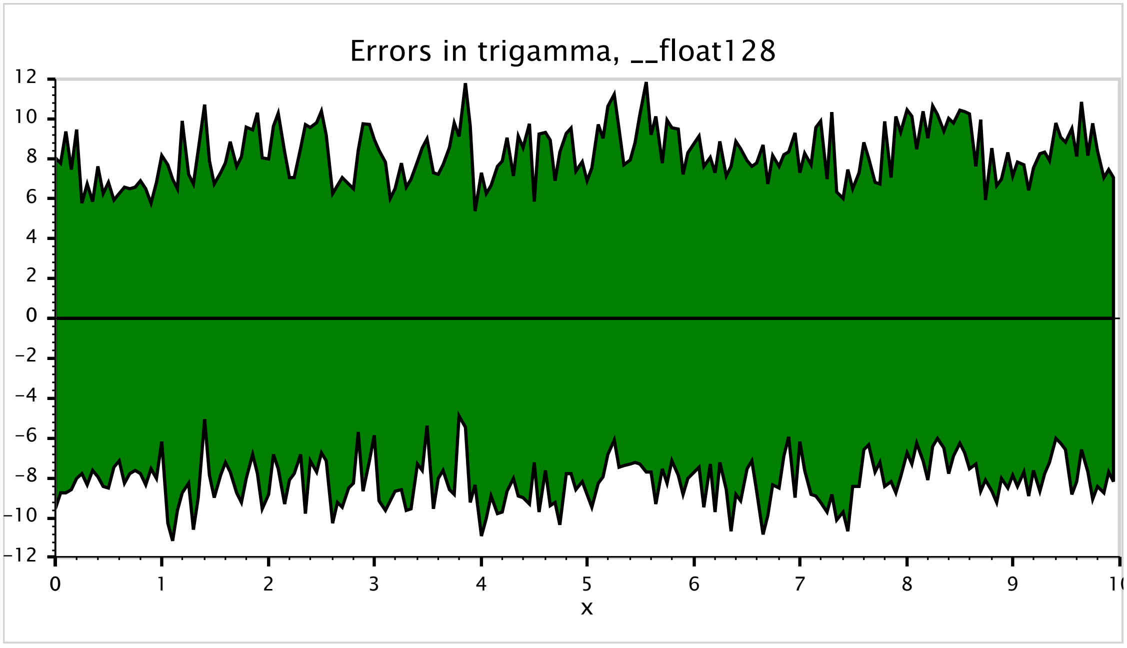
	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Mathematica Data	Max = 0.998ε (Mean = 0.105ε) (*GSL 2.1:* Max = 1.34e+04ε (Mean = 1.49e+03ε)) (*Rmath 3.2.3:* Max = 1.34e+04ε (Mean = 1.51e+03ε))	Max = 1.28ε (Mean = 0.449ε)	Max = 1.28ε (Mean = 0.449ε)	Max = 1ε (Mean = 0.382ε)

As shown above, error rates are generally very low for built in types. For multiprecision types, error rates are typically in the order of a few epsilon.

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

Testing is against Mathematica generated spot values to 35 digit precision.

Implementation

The arbitrary precision version of this function simply calls polygamma.

For built in fixed precision types, negative arguments are first made positive via:

$$\psi^{(1)}(1-x) = \frac{\pi^2}{\sin^2(\pi x)} - \psi^{(1)}(x)$$

Then arguments in the range [0, 1) are shifted to >= 1 via:

$$\psi^{(1)}(x) = \psi^{(1)}(1 + x) + \frac{1}{x^2}$$

Then evaluation is via one of a number of rational approximations, for small x these are of the form:

$$\psi^{(1)}(x) = \frac{(C + R(x))}{x^2}$$

and for large x of the form:

$$\psi^{(1)}(x) = \frac{\left(1 + R\left(\frac{1}{x}\right)\right)}{x}$$

Polygamma

Synopsis

```
#include <boost/math/special_functions/polygamma.hpp>
```

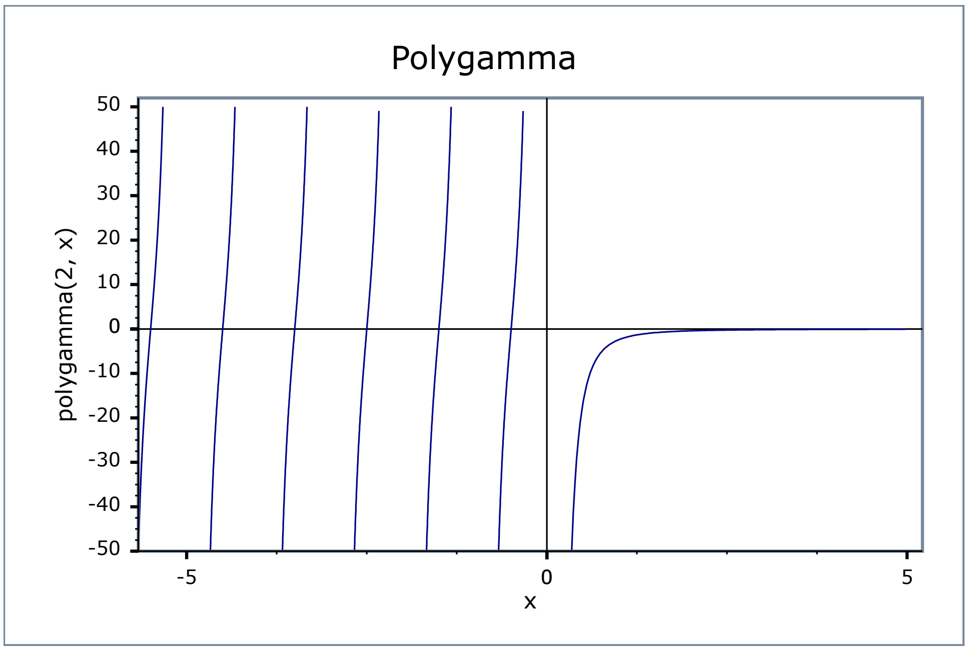
```
namespace boost{ namespace math{

template <class T>
calculated-result-type polygamma(int n, T z);

template <class T, class Policy>
calculated-result-type polygamma(int n, T z, const Policy&);

}} // namespaces
```

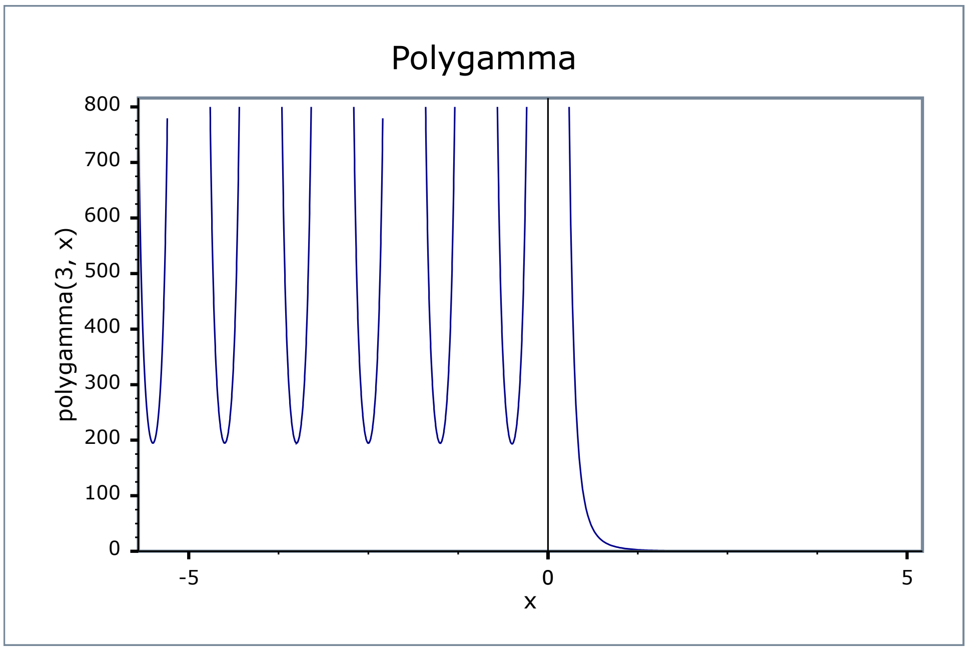
Description

Returns the polygamma function of *x*. Polygamma is defined as the n'th derivative of the digamma function:

$$\psi^{(n)}(x) = (-1)^{n+1} n! \sum_{k=0}^{\infty} \frac{1}{(k + x)^{n+1}} = \frac{\partial^n \psi(x)}{\partial^n x}$$

The following graphs illustrate the behaviour of the function for odd and even order:





The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The return type of this function is computed using the *result type calculation rules*: the result is of type `double` when T is an integer type, and type T otherwise.

Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

Table 29. Error rates for polygamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Mathematica Data	Max = 0.824ε (Mean = 0.0574ε) (*GSL 2.1:* Max = 62.9ε (Mean = 12.8ε)) (*Rmath 3.2.3:* Max = 108ε (Mean = 15.2ε))	Max = 7.38ε (Mean = 1.84ε)	Max = 34.3ε (Mean = 7.65ε)	Max = 9.32ε (Mean = 1.95ε)
Mathematica Data - large arguments	Max = 0.998ε (Mean = 0.0592ε) (*GSL 2.1:* Max = 244ε (Mean = 32.8ε) And other failures.) (*Rmath 3.2.3:* Max = 1.71e+56ε (Mean = 1.01e+55ε) And other failures.)	Max = 2.23ε (Mean = 0.323ε)	Max = 11.1ε (Mean = 0.848ε)	Max = 150ε (Mean = 13.9ε)
Mathematica Data - negative arguments	Max = 0.516ε (Mean = 0.022ε) (*GSL 2.1:* Max = 36.6ε (Mean = 3.04ε) And other failures.) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 269ε (Mean = 87.7ε)	Max = 269ε (Mean = 88.4ε)	Max = 497ε (Mean = 129ε)
Mathematica Data - large negative arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.79ε (Mean = 0.197ε) And other failures.) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 155ε (Mean = 96.4ε)	Max = 155ε (Mean = 96.4ε)	Max = 162ε (Mean = 101ε)
Mathematica Data - small arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 15.2ε (Mean = 5.03ε)) (*Rmath 3.2.3:* Max = 106ε (Mean = 20ε))	Max = 3.33ε (Mean = 0.75ε)	Max = 3.33ε (Mean = 0.75ε)	Max = 3ε (Mean = 0.496ε)
Mathematica Data - Large orders and other bug cases	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 151ε (Mean = 39.3ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 54.5ε (Mean = 13.3ε)	Max = 145ε (Mean = 55.9ε)	Max = 200ε (Mean = 57.2ε)

As shown above, error rates are generally very acceptable for moderately sized arguments. Error rates should stay low for exact inputs, however, please note that the function becomes exceptionally sensitive to small changes in input for large n and negative x, indeed for cases where *n!* would overflow, the function changes directly from -∞ to +∞ somewhere between each negative integer *- these cases are not handled correctly*.

For these reasons results should be treated with extreme caution when n is large and x negative.

Testing

Testing is against Mathematica generated spot values to 35 digit precision.

Implementation

For x < 0 the following reflection formula is used:

$$\psi^{(n)}(1-x) = (-1)^n \psi^{(n)}(x) + (-1)^n \pi \frac{\partial^n \cot(\pi x)}{\partial x^n}$$

The n'th derivative of *cot(x)* is tabulated for small *n*, and for larger n has the general form:

$$\frac{\partial^n \cot(\pi x)}{\partial x^n} = \frac{\pi^n}{\sin^{n+1}(\pi x)} \sum_{k=0}^{n-1} C_{k,n} \cos^k(\pi n)$$

The coefficients of the cosine terms can be calculated iteratively starting from $C_{1,0} = -1$ and then using

$$\frac{\partial}{\partial x} \frac{\cos^k(\theta)}{\sin^n(\theta)} = \frac{1}{\sin^{n+1}(\theta)} \left((k-n)\cos^{n+1}(\theta) - k\cos^{n-1}(\theta) \right)$$

to generate coefficients for n+1.

Note that every other coefficient is zero, and therefore what we have are even or odd polynomials depending on whether n is even or odd.

Once x is positive then we have two methods available to us, for small x we use the series expansion:

$$\psi^{(n)}(x) = \frac{(-1)^{n-1} n!}{x^{n+1}} + \sum_{k=0}^{\infty} \frac{(-1)^{k+n+1} (k+n)! \zeta(k+n+1) x^k}{k!}$$

Note that the evaluation of zeta functions at integer values is essentially a table lookup as zeta is optimized for those cases.

For large x we use the asymptotic expansion:

$$\psi^{(n)}(x) \propto \frac{(-1)^{n-1}(n-1)!(n+2x)}{2x^{n+1}} - (-1)^n \sum_{k=1}^{\infty} \frac{(2k+n-1)!}{(2k)! x^{2k+n}} B_{2k}$$

For x in-between the two extremes we use the relation:

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

to make x large enough for the asymptotic expansion to be used.

There are also two special cases:

$$\psi^{(n)}(1) = (-1)^{n+1} n!\varsigma(n+1)$$

$$\psi^{(n)}\left(\frac{1}{2}\right) = (-1)^{n+1} n!\left(2^{n+1} - 1\right)\varsigma(n+1)$$

Ratios of Gamma Functions

```
#include <boost/math/special_functions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type tgamma_ratio(T1 a, T2 b);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_ratio(T1 a, T2 b, const Policy&);

template <class T1, class T2>
calculated-result-type tgamma_delta_ratio(T1 a, T2 delta);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_delta_ratio(T1 a, T2 delta, const Policy&);

}} // namespaces
```

Description

```
template <class T1, class T2>
calculated-result-type tgamma_ratio(T1 a, T2 b);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_ratio(T1 a, T2 b, const Policy&);
```

Returns the ratio of gamma functions:

$$\text{tgamma\_ratio(a,b)} \quad = \quad \frac{\Gamma(a)}{\Gamma(b)}$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Internally this just calls `tgamma_delta_ratio(a, b-a)`.

```
template <class T1, class T2>
calculated-result-type tgamma_delta_ratio(T1 a, T2 delta);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_delta_ratio(T1 a, T2 delta, const Policy&);
```

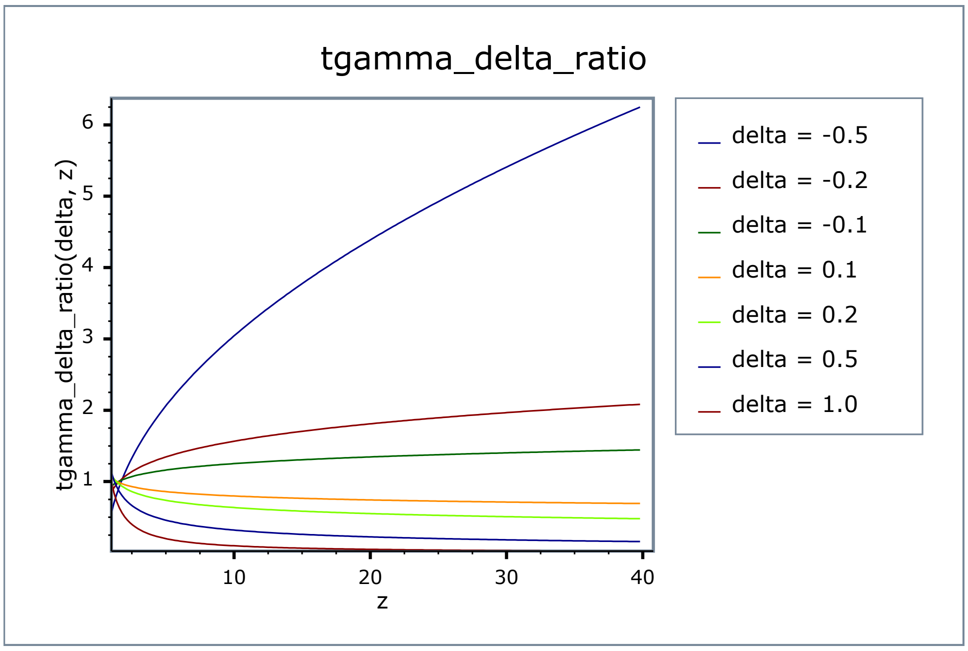
Returns the ratio of gamma functions:

$$\text{tgamma\_delta\_ratio(a,delta)} \quad = \quad \frac{\Gamma(a)}{\Gamma(a+delta)}$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that the result is calculated accurately even when *delta* is small compared to *a*: indeed even if *a+delta* ~ *a*. The function is typically used when *a* is large and *delta* is very small.

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types, otherwise the result type is simple T1.



Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

Table 30. Error rates for tgamma_delta_ratio

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
tgamma + small delta ratios	Max = 0ε (Mean = 0ε)	Max = 5.83ε (Mean = 1.3ε)	Max = 15.4ε (Mean = 2.09ε)	Max = 7.56ε (Mean = 1.31ε)
tgamma + small delta ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 7.94ε (Mean = 1.4ε)	Max = 18.3ε (Mean = 2.03ε)	Max = 7.43ε (Mean = 1.42ε)
tgamma + small integer ratios	Max = 0ε (Mean = 0ε)	Max = 1.96ε (Mean = 0.677ε)	Max = 1.96ε (Mean = 0.677ε)	Max = 2.74ε (Mean = 0.736ε)
tgamma + small integer ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 1.62ε (Mean = 0.451ε)	Max = 1.62ε (Mean = 0.451ε)	Max = 2.15ε (Mean = 0.685ε)
integer tgamma ratios	Max = 0ε (Mean = 0ε)	Max = 0.997ε (Mean = 0.4ε)	Max = 0.997ε (Mean = 0.4ε)	Max = 0.968ε (Mean = 0.386ε)
integer tgamma ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 0.853ε (Mean = 0.176ε)	Max = 0.853ε (Mean = 0.176ε)	Max = 0.974ε (Mean = 0.175ε)

Table 31. Error rates for tgamma_ratio

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma ratios	Max = 0.694ε (Mean = 0.0347ε)	Max = 2.99ε (Mean = 1.15ε)	Max = 174ε (Mean = 61.2ε)	Max = 3.28ε (Mean = 1.12ε)

Testing

Accuracy tests use data generated at very high precision (with NTL RR class set at 1000-bit precision: about 300 decimal digits) and a deliberately naive calculation of Γ(x)/Γ(y).

Implementation

The implementation of these functions is very similar to that of beta, and is based on combining similar power terms to improve accuracy and avoid spurious overflow/underflow.

In addition there are optimisations for the situation where *delta* is a small integer: in which case this function is basically the reciprocal of a rising factorial, or where both arguments are smallish integers: in which case table lookup of factorials can be used to calculate the ratio.

Incomplete Gamma Functions

Synopsis

```
#include <boost/math/special_functions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type gamma_p(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p(T1 a, T2 z, const Policy&);

template <class T1, class T2>
calculated-result-type gamma_q(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q(T1 a, T2 z, const Policy&);

template <class T1, class T2>
calculated-result-type tgamma_lower(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_lower(T1 a, T2 z, const Policy&);

template <class T1, class T2>
calculated-result-type tgamma(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type tgamma(T1 a, T2 z, const Policy&);

}} // namespaces
```

Description

There are four incomplete gamma functions: two are normalised versions (also known as *regularized* incomplete gamma functions) that return values in the range [0, 1], and two are non-normalised and return values in the range [0, Γ(a)]. Users interested in statistical applications should use the normalised versions (gamma_p and gamma_q).

All of these functions require *a > 0* and *z >= 0*, otherwise they return the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

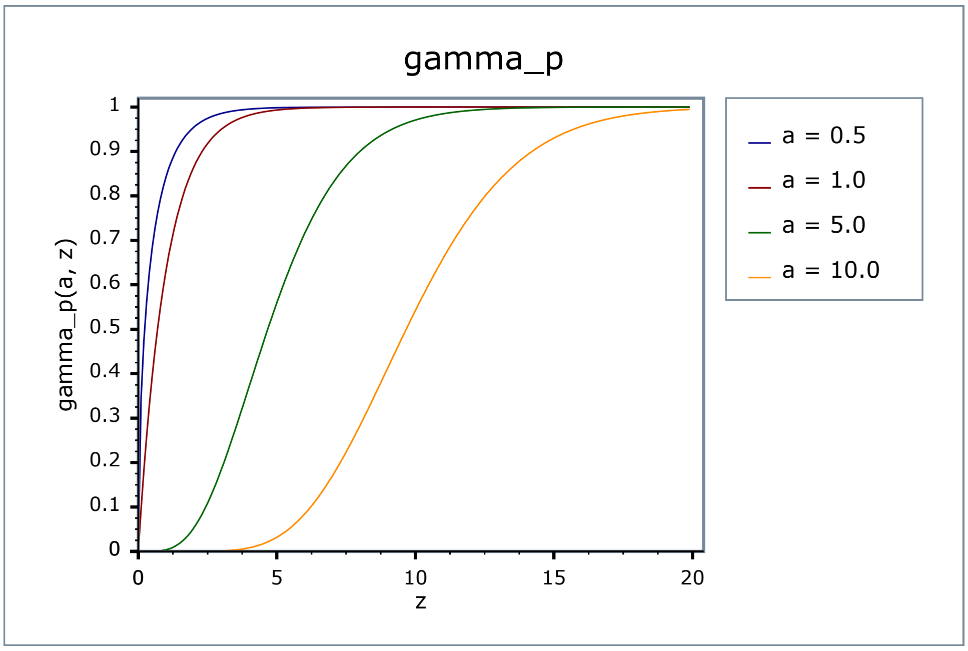
The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types, otherwise the return type is simply T1.

```
template <class T1, class T2>
calculated-result-type gamma_p(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p(T1 a, T2 z, const Policy&);
```

Returns the normalised lower incomplete gamma function of a and z:

$$\text{gamma\_p}(a, z) = P(a, z) = \frac{\gamma(a, z)}{\Gamma(a)} = \frac{1}{\Gamma(a)}\int_0^z t^{a-1}e^{-t}dt$$

This function changes rapidly from 0 to 1 around the point z == a:

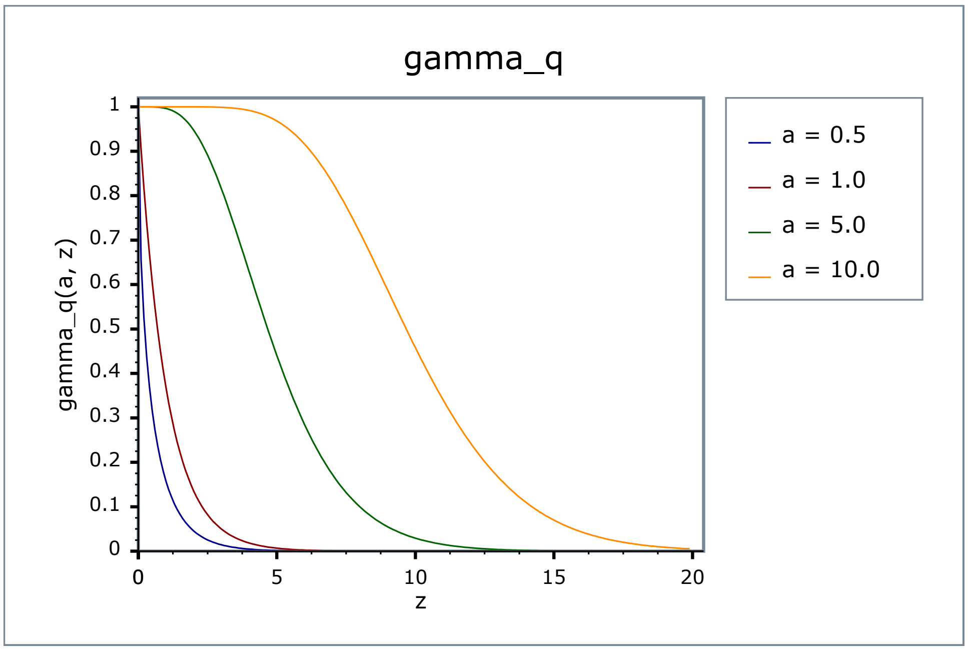


```
template <class T1, class T2>
calculated-result-type gamma_q(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q(T1 a, T2 z, const Policy&);
```

Returns the normalised upper incomplete gamma function of a and z:

$$\text{gamma\_q}(a, z) = Q(a, z) = \frac{\Gamma(a, z)}{\Gamma(a)} = \frac{1}{\Gamma(a)}\int_z^\infty t^{a-1}e^{-t}dt$$

This function changes rapidly from 1 to 0 around the point z == a:



```
template <class T1, class T2>
calculated-result-type tgamma_lower(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type tgamma_lower(T1 a, T2 z, const Policy&);
```

Returns the full (non-normalised) lower incomplete gamma function of a and z:

$$\text{tgamma\_lower}\,(a, z) \quad = \quad \gamma(a, z) \quad = \quad \int_0^z t^{a-1} e^{-t} dt$$

```
template <class T1, class T2>
calculated-result-type tgamma(T1 a, T2 z);

template <class T1, class T2, class Policy>
calculated-result-type tgamma(T1 a, T2 z, const Policy&);
```

Returns the full (non-normalised) upper incomplete gamma function of a and z:

$$\text{tgamma}\,(a, z) \quad = \quad \Gamma(a, z) \quad = \quad \int_z^\infty t^{a-1} e^{-t} dt$$

Accuracy

The following tables give peak and mean relative errors in over various domains of a and z, along with comparisons to the GSL-1.9 and Cephes libraries. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error.

Note that errors grow as *a* grows larger.

Note also that the higher error rates for the 80 and 128 bit long double results are somewhat misleading: expected results that are zero at 64-bit double precision may be non-zero - but exceptionally small - with the larger exponent range of a long double. These results therefore reflect the more extreme nature of the tests conducted for these types.

All values are in units of epsilon.

Table 32. Error rates for gamma_p

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma(a, z) medium values	Max = 0.955ε (Mean = 0.05ε) (*GSL 2.1:* Max = 342ε (Mean = 45.8ε)) (*Rmath 3.2.3:* Max = 389ε (Mean = 44ε))	Max = 41.6ε (Mean = 8.09ε)	Max = 239ε (Mean = 30.2ε)	Max = 35.1ε (Mean = 6.98ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.82ε (Mean = 0.758ε)) (*Rmath 3.2.3:* Max = 1.01ε (Mean = 0.306ε))	Max = 2ε (Mean = 0.464ε)	Max = 2ε (Mean = 0.461ε)	Max = 1.54ε (Mean = 0.439ε)
tgamma(a, z) large values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.02e+03ε (Mean = 105ε)) (*Rmath 3.2.3:* Max = 1.11e+03ε (Mean = 67.5ε))	Max = 3.08e+04ε (Mean = 1.86e+03ε)	Max = 3.02e+04ε (Mean = 1.91e+03ε)	Max = 243ε (Mean = 20.2ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 128ε (Mean = 22.6ε)) (*Rmath 3.2.3:* Max = 66.2ε (Mean = 12.2ε))	Max = 11.8ε (Mean = 2.66ε)	Max = 71.6ε (Mean = 9.47ε)	Max = 13ε (Mean = 2.97ε)

Table 33. Error rates for gamma_q

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma(a, z) medium values	Max = 0.927ε (Mean = 0.035ε) (*GSL 2.1:* Max = 201ε (Mean = 13.5ε)) (*Rmath 3.2.3:* Max = 131ε (Mean = 12.7ε))	Max = 32.3ε (Mean = 6.61ε)	Max = 199ε (Mean = 26.6ε)	Max = 23.7ε (Mean = 4ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.38e+10ε (Mean = 1.05e+09ε)) (*Rmath 3.2.3:* Max = 65.6ε (Mean = 11ε))	Max = 2.45ε (Mean = 0.885ε)	Max = 2.45ε (Mean = 0.819ε)	Max = 2.26ε (Mean = 0.74ε)
tgamma(a, z) large values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.71e+04ε (Mean = 2.16e+03ε)) (*Rmath 3.2.3:* Max = 1.02e+03ε (Mean = 62.7ε))	Max = 6.82e+03ε (Mean = 414ε)	Max = 1.15e+04ε (Mean = 733ε)	Max = 469ε (Mean = 31.5ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 118ε (Mean = 12.5ε)) (*Rmath 3.2.3:* Max = 138ε (Mean = 16.9ε))	Max = 11.1ε (Mean = 2.07ε)	Max = 54.7ε (Mean = 6.16ε)	Max = 8.72ε (Mean = 1.48ε)

Table 34. Error rates for tgamma_lower

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma(a, z) medium values	Max = 0.833ε (Mean = 0.0315ε) (*GSL 2.1:* Max = 0.833ε (Mean = 0.0315ε))	Max = 6.79ε (Mean = 1.46ε)	Max = 363ε (Mean = 63.8ε)	Max = 5.62ε (Mean = 1.49ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 1.97ε (Mean = 0.555ε)	Max = 1.97ε (Mean = 0.558ε)	Max = 1.57ε (Mean = 0.525ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 4.83ε (Mean = 1.15ε)	Max = 84.7ε (Mean = 17.5ε)	Max = 2.69ε (Mean = 0.849ε)

Table 35. Error rates for tgamma (incomplete)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
tgamma(a, z) medium values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 200ε (Mean = 13.3ε))	Max = 8.47ε (Mean = 1.9ε)	Max = 412ε (Mean = 95.5ε)	Max = 8.14ε (Mean = 1.76ε)
tgamma(a, z) small values	Max = 0.753ε (Mean = 0.0474ε) (*GSL 2.1:* Max = 1.38e+10ε (Mean = 1.05e+09ε))	Max = 2.31ε (Mean = 0.775ε)	Max = 2.13ε (Mean = 0.717ε)	Max = 2.53ε (Mean = 0.66ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 117ε (Mean = 12.5ε))	Max = 5.52ε (Mean = 1.48ε)	Max = 79.6ε (Mean = 20.9ε)	Max = 5.16ε (Mean = 1.33ε)

Testing

There are two sets of tests: spot tests compare values taken from Mathworld's online evaluator with this implementation to perform a basic "sanity check". Accuracy tests use data generated at very high precision (using NTL's RR class set at 1000-bit precision) using this implementation with a very high precision 60-term Lanczos approximation, and some but not all of the special case handling disabled. This is less than satisfactory: an independent method should really be used, but apparently a complete lack of such methods are available. We can't even use a deliberately naive implementation without special case handling since Legendre's continued fraction (see below) is unstable for small a and z.

Implementation

These four functions share a common implementation since they are all related via:

1)

$$Q(a, x) + P(a, x) = 1$$

2)

$$\Gamma(a, z) + \gamma(a, z) = \Gamma(a)$$

3)

$$Q(a, z) = \frac{\Gamma(a, z)}{\Gamma(a)} \quad , \quad P(a, z) = \frac{\gamma(a, z)}{\Gamma(a)}$$

The lower incomplete gamma is computed from its series representation:

4)

$$\gamma(a, x) = x^a e^{-x} \sum_{k=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a+k+1)} x^n = x^a e^{-x} \sum_{k=0}^{\infty} \frac{x^n}{a^{\bar{k+1}}}$$

Or by subtraction of the upper integral from either Γ(a) or 1 when *x - (1*(3x)) > a and x > 1.1/.

The upper integral is computed from Legendre's continued fraction representation:

5)

$$\Gamma(a, x) = \frac{x^a e^{-x}}{x - a + 1 + \frac{a_k}{b_k + \frac{a_{k+1}}{b_{k+1} + \dots}}} \quad ; \quad a_k = k(a-k) \quad ; \quad b_k = x - a + 2k + 1$$

When *(x > 1.1)* or by subtraction of the lower integral from either Γ(a) or 1 when *x - (1*(3x)) < a/.

For *x < 1.1* computation of the upper integral is more complex as the continued fraction representation is unstable in this area. However there is another series representation for the lower integral:

6)

$$\gamma(a, x) = x^a \sum_{k=0}^{\infty} \frac{(-1)^k x^k}{(a+k)k!}$$

That lends itself to calculation of the upper integral via rearrangement to:

7)

$$\begin{aligned} \Gamma(a, x) &= \frac{\text{tgamma1pm1}\,(a) - \text{powm1}\,(x, a)}{a} + x^a \sum_{k=1}^{\infty} \frac{(-1)^k x^k}{(a+k)k!} \\ \text{tgamma1pm1}\,(a) &= \Gamma(a+1) - 1 \\ \text{powm1}\,(x, a) &= x^a - 1 \end{aligned}$$

Refer to the documentation for powm1 and tgamma1pm1 for details of their implementation. Note however that the precision of tgamma1pm1 is capped to either around 35 digits, or to that of the Lanczos approximation associated with type T - if there is one - whichever of the two is the greater. That therefore imposes a similar limit on the precision of this function in this region.

For *x < 1.1* the crossover point where the result is ~0.5 no longer occurs for *x ~ y*. Using *x * 0.75 < a* as the crossover criterion for *0.5 < x <= 1.1* keeps the maximum value computed (whether it's the upper or lower interval) to around 0.75. Likewise for *x <= 0.5* then using *-0.4 / log(x) < a* as the crossover criterion keeps the maximum value computed to around 0.7 (whether it's the upper or lower interval).

There are two special cases used when a is an integer or half integer, and the crossover conditions listed above indicate that we should compute the upper integral Q. If a is an integer in the range *1 <= a < 30* then the following finite sum is used:

9)

$$Q(a, x) = e^{-x} \sum_{n=0}^{a-1} \frac{x^n}{n!} \quad ; \quad a \in \mathbb{N}^+$$

While for half integers in the range *0.5 <= a < 30* then the following finite sum is used:

10)

$$Q(a, x) = \text{erfc}\,(\sqrt{x}) + \frac{e^{-x}}{\sqrt{\pi x}} \sum_{n=1}^{i} \frac{x^n}{\left(1 - \frac{1}{2}\right) \dots \left(n - \frac{1}{2}\right)} \quad ; \quad a = i + \frac{1}{2} \quad ; \quad i \in \mathbb{N}^+$$

These are both more stable and more efficient than the continued fraction alternative.

When the argument *a* is large, and *x* ~ *a* then the series (4) and continued fraction (5) above are very slow to converge. In this area an expansion due to Temme is used:

11)

$$P(a,\ x) \quad = \quad \frac{1}{2}\operatorname{erfc}\left(\sqrt{y}\right) - \frac{e^{-y}}{\sqrt{2\pi a}}T(a,\ \lambda) \quad ; \quad \lambda \leq 1$$

12)

$$Q(a,\ x) \quad = \quad \frac{1}{2}\operatorname{erfc}\left(\sqrt{y}\right) + \frac{e^{-y}}{\sqrt{2\pi a}}T(a,\ \lambda) \quad ; \quad \lambda > 1$$

13)

$$\lambda = \frac{x}{a} \quad , \quad y = a(\lambda - 1 - \ln\lambda) = -a(\ln(1+\sigma) - \sigma) \quad ; \quad \sigma = \frac{x-a}{a}$$

14)

$$T(a,\ \lambda) \quad = \quad \sum_{k=0}^{N}\left(\sum_{n=0}^{M} C_k^n z^n\right)a^{-k} \quad ; \quad z = \operatorname{sign}(\lambda - 1)\sqrt{2\sigma}$$

The double sum is truncated to a fixed number of terms - to give a specific target precision - and evaluated as a polynomial-of-polynomials. There are versions for up to 128-bit long double precision: types requiring greater precision than that do not use these expansions. The coefficients C_k^n are computed in advance using the recurrence relations given by Temme. The zone where these expansions are used is

```
(a > 20) && (a < 200) && fabs(x-a)/a < 0.4
```

And:

```
(a > 200) && (fabs(x-a)/a < 4.5/sqrt(a))
```

The latter range is valid for all types up to 128-bit long doubles, and is designed to ensure that the result is larger than 10^{-6}, the first range is used only for types up to 80-bit long doubles. These domains are narrower than the ones recommended by either Temme or Didonato and Morris. However, using a wider range results in large and inexact (i.e. computed) values being passed to the `exp` and `erfc` functions resulting in significantly larger error rates. In other words there is a fine trade off here between efficiency and error. The current limits should keep the number of terms required by (4) and (5) to no more than ~20 at double precision.

For the normalised incomplete gamma functions, calculation of the leading power terms is central to the accuracy of the function. For smallish a and x combining the power terms with the Lanczos approximation gives the greatest accuracy:

15)

$$\frac{x^a e^{-x}}{\Gamma(a)} \quad = \quad e^{x-a}\left(\frac{x}{a+g-0.5}\right)^a \sqrt{\frac{a+g-0.5}{e}}\,\frac{1}{L(a)}$$

In the event that this causes underflow/overflow then the exponent can be reduced by a factor of *a* and brought inside the power term.

When a and x are large, we end up with a very large exponent with a base near one: this will not be computed accurately via the pow function, and taking logs simply leads to cancellation errors. The worst of the errors can be avoided by using:

16)

$$e^{x-a}\left(\frac{x}{a+g-0.5}\right)^{a} = e^{\left(a\,\mathrm{log1pmx}\left(\frac{x-a-g+0.5}{a+g-0.5}\right)+\frac{x(0.5-g)}{a+g-0.5}\right)} \quad ; \quad \mathrm{log1pmx}\,(z) = \ln(1+z)-z$$

when *a-x* is small and a and x are large. There is still a subtraction and therefore some cancellation errors - but the terms are small so the absolute error will be small - and it is absolute rather than relative error that counts in the argument to the *exp* function. Note that for sufficiently large a and x the errors will still get you eventually, although this does delay the inevitable much longer than other methods. Use of *log(1+x)-x* here is inspired by Temme (see references below).

References

- N. M. Temme, A Set of Algorithms for the Incomplete Gamma Functions, Probability in the Engineering and Informational Sciences, 8, 1994.
- N. M. Temme, The Asymptotic Expansion of the Incomplete Gamma Functions, Siam J. Math Anal. Vol 10 No 4, July 1979, p757.
- A. R. Didonato and A. H. Morris, Computation of the Incomplete Gamma Function Ratios and their Inverse. ACM TOMS, Vol 12, No 4, Dec 1986, p377.
- W. Gautschi, The Incomplete Gamma Functions Since Tricomi, In Tricomi's Ideas and Contemporary Applied Mathematics, Atti dei Convegni Lincei, n. 147, Accademia Nazionale dei Lincei, Roma, 1998, pp. 203--237. http://citeseer.ist.psu.edu/gautschi98incomplete.html

Incomplete Gamma Function Inverses

Synopsis

```
#include <boost/math/special_functions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type gamma_q_inv(T1 a, T2 q);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q_inv(T1 a, T2 q, const Policy&);

template <class T1, class T2>
calculated-result-type gamma_p_inv(T1 a, T2 p);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p_inv(T1 a, T2 p, const Policy&);

template <class T1, class T2>
calculated-result-type gamma_q_inva(T1 x, T2 q);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q_inva(T1 x, T2 q, const Policy&);

template <class T1, class T2>
calculated-result-type gamma_p_inva(T1 x, T2 p);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p_inva(T1 x, T2 p, const Policy&);

}} // namespaces
```

Description

There are four incomplete gamma function inverses which either compute *x* given *a* and *p* or *q*, or else compute *a* given *x* and either *p* or *q*.

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types, otherwise the return type is simply T1.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Tip

When people normally talk about the inverse of the incomplete gamma function, they are talking about inverting on parameter *x*. These are implemented here as gamma_p_inv and gamma_q_inv, and are by far the most efficient of the inverses presented here.

The inverse on the *a* parameter finds use in some statistical applications but has to be computed by rather brute force numerical techniques and is consequently several times slower. These are implemented here as gamma_p_inva and gamma_q_inva.

```
template <class T1, class T2>
calculated-result-type gamma_q_inv(T1 a, T2 q);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q_inv(T1 a, T2 q, const Policy&);
```

Returns a value x such that: `q = gamma_q(a, x);`

Requires: *a > 0* and *1 >= p,q >= 0*.

```
template <class T1, class T2>
calculated-result-type gamma_p_inv(T1 a, T2 p);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p_inv(T1 a, T2 p, const Policy&);
```

Returns a value x such that: `p = gamma_p(a, x);`

Requires: *a > 0* and *1 >= p,q >= 0*.

```
template <class T1, class T2>
calculated-result-type gamma_q_inva(T1 x, T2 q);

template <class T1, class T2, class Policy>
calculated-result-type gamma_q_inva(T1 x, T2 q, const Policy&);
```

Returns a value a such that: `q = gamma_q(a, x);`

Requires: *x > 0* and *1 >= p,q >= 0*.

```
template <class T1, class T2>
calculated-result-type gamma_p_inva(T1 x, T2 p);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p_inva(T1 x, T2 p, const Policy&);
```

Returns a value a such that: `p = gamma_p(a, x);`

Requires: *x > 0* and *1 >= p,q >= 0*.

Accuracy

The accuracy of these functions doesn't vary much by platform or by the type T. Given that these functions are computed by iterative methods, they are deliberately "detuned" so as not to be too accurate: it is in any case impossible for these function to be more accurate than the regular forward incomplete gamma functions. In practice, the accuracy of these functions is very similar to that of gamma_p and gamma_q functions:

Table 36. Error rates for gamma_p_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
incomplete gamma inverse(a, z) medium values	Max = 0.993ε (Mean = 0.15ε) (*Rmath 3.2.3:* Max = 4.88ε (Mean = 0.868ε))	Max = 1.8ε (Mean = 0.406ε)	Max = 1.89ε (Mean = 0.466ε)	Max = 1.71ε (Mean = 0.34ε)
incomplete gamma inverse(a, z) large values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 0.816ε (Mean = 0.0874ε))	Max = 0.509ε (Mean = 0.0447ε)	Max = 0.509ε (Mean = 0.0447ε)	Max = 0.924ε (Mean = 0.108ε)
incomplete gamma inverse(a, z) small values	Max = 441ε (Mean = 53.9ε) (*Rmath 3.2.3:* Max = 547ε (Mean = 61.6ε))	Max = 9.17e+03ε (Mean = 1.45e+03ε)	Max = 1.09e+04ε (Mean = 1.3e+03ε)	Max = 1.1e+03ε (Mean = 131ε)

Table 37. Error rates for gamma_q_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
incomplete gamma inverse(a, z) medium values	Max = 0.912ε (Mean = 0.154ε) (*Rmath 3.2.3:* Max = 4.66ε (Mean = 0.792ε))	Max = 6.2ε (Mean = 0.627ε)	Max = 6.2ε (Mean = 0.683ε)	Max = 2.88ε (Mean = 0.469ε)
incomplete gamma inverse(a, z) large values	Max = 0.894ε (Mean = 0.0915ε) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.106ε))	Max = 0ε (Mean = 0ε)	Max = 0ε (Mean = 0ε)	Max = 0.814ε (Mean = 0.0856ε)
incomplete gamma inverse(a, z) small values	Max = 292ε (Mean = 36.4ε) (*Rmath 3.2.3:* Max = 415ε (Mean = 48.7ε))	Max = 8.28e+03ε (Mean = 1.09e+03ε)	Max = 8.98e+03ε (Mean = 877ε)	Max = 451ε (Mean = 64.7ε)

Table 38. Error rates for gamma_p_inva

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Incomplete gamma inverses.	Max = 0ε (Mean = 0ε)	Max = 7.87ε (Mean = 1.15ε)	Max = 4.08ε (Mean = 1.12ε)	Max = 4.92ε (Mean = 1.03ε)

Table 39. Error rates for gamma_q_inva

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Incomplete gamma inverses.	Max = 0ε (Mean = 0ε)	Max = 8.42ε (Mean = 1.3ε)	Max = 7.86ε (Mean = 1.24ε)	Max = 5.05ε (Mean = 1.08ε)

Testing

There are two sets of tests:

- Basic sanity checks attempt to "round-trip" from *a* and *x* to *p* or *q* and back again. These tests have quite generous tolerances: in general both the incomplete gamma, and its inverses, change so rapidly that round tripping to more than a couple of significant digits isn't possible. This is especially true when *p* or *q* is very near one: in this case there isn't enough "information content" in the input to the inverse function to get back where you started.
- Accuracy checks using high precision test values. These measure the accuracy of the result, given exact input values.

Implementation

The functions gamma_p_inv and gamma_q_inv share a common implementation.

First an initial approximation is computed using the methodology described in:

A. R. Didonato and A. H. Morris, Computation of the Incomplete Gamma Function Ratios and their Inverse, ACM Trans. Math. Software 12 (1986), 377-393.

Finally, the last few bits are cleaned up using Halley iteration, the iteration limit is set to 2/3 of the number of bits in T, which by experiment is sufficient to ensure that the inverses are at least as accurate as the normal incomplete gamma functions. In testing, no more than 3 iterations are required to produce a result as accurate as the forward incomplete gamma function, and in many cases only one iteration is required.

The functions gamma_p_inva and gamma_q_inva also share a common implementation but are handled separately from gamma_p_inv and gamma_q_inv.

An initial approximation for *a* is computed very crudely so that *gamma_p(a, x) ~ 0.5*, this value is then used as a starting point for a generic derivative-free root finding algorithm. As a consequence, these two functions are rather more expensive to compute than the gamma_p_inv or gamma_q_inv functions. Even so, the root is usually found in fewer than 10 iterations.

Derivative of the Incomplete Gamma Function

Synopsis

```
#include <boost/math/special_functions/gamma.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type gamma_p_derivative(T1 a, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type gamma_p_derivative(T1 a, T2 x, const Policy&);

}} // namespaces
```

Description

This function find some uses in statistical distributions: it implements the partial derivative with respect to *x* of the incomplete gamma function.

$$\text{gamma\_p\_derivative}(a, x) \quad = \quad \frac{\partial}{\partial x}P(a, x) \quad = \quad \frac{e^{-x}x^{a-1}}{\Gamma(a)}$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that the derivative of the function gamma_q can be obtained by negating the result of this function.

The return type of this function is computed using the *result type calculation rules* when T1 and T2 are different types, otherwise the return type is simply T1.

Accuracy

Almost identical to the incomplete gamma function gamma_p: refer to the documentation for that function for more information.

Implementation

This function just expose some of the internals of the incomplete gamma function gamma_p: refer to the documentation for that function for more information.

Factorials and Binomial Coefficients

Factorial

Synopsis

```
#include <boost/math/special_functions/factorials.hpp>
```

```
namespace boost{ namespace math{

template <class T>
T factorial(unsigned i);

template <class T, class Policy>
T factorial(unsigned i, const Policy&);

template <class T>
constexpr T unchecked_factorial(unsigned i);

template <class T>
struct max_factorial;

}} // namespaces
```

Description

Important

The functions described below are templates where the template argument T CANNOT be deduced from the arguments passed to the function. Therefore if you write something like:

`boost::math::factorial(2);`

You will get a (perhaps perplexing) compiler error, ususally indicating that there is no such function to be found. Instead you need to specify the return type explicity and write:

`boost::math::factorial<double>(2);`

So that the return type is known.

Furthermore, the template argument must be a real-valued type such as `float` or `double` and not an integer type - that would overflow far too easily for quite small values of parameter `i`!

The source code `static_assert` and comment just after the will be:

```
BOOST_STATIC_ASSERT(!boost::is_integral<T>::value);
// factorial<unsigned int>(n) is not implemented
// because it would overflow integral type T for too small n
// to be useful. Use instead a floating-point type,
// and convert to an unsigned type if essential, for example:
// unsigned int nfac = static_cast<unsigned int>(factorial<double>(n));
// See factorial documentation for more detail.
```

```
template <class T>
T factorial(unsigned i);

template <class T, class Policy>
T factorial(unsigned i, const Policy&);
```

Returns `i!`.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

For `i <= max_factorial<T>::value` this is implemented by table lookup, for larger values of `i`, this function is implemented in terms of tgamma.

If `i` is so large that the result can not be represented in type T, then calls overflow_error.

```
template <class T>
constexpr T unchecked_factorial(unsigned i);
```

Returns `i!`.

Internally this function performs table lookup of the result. Further it performs no range checking on the value of i: it is up to the caller to ensure that `i <= max_factorial<T>::value`. This function is intended to be used inside inner loops that require fast table lookup of factorials, but requires care to ensure that argument `i` never grows too large.

```
template <class T>
struct max_factorial
{
   static const unsigned value = X;
};
```

This traits class defines the largest value that can be passed to `unchecked_factorial`. The member `value` can be used where integral constant expressions are required: for example to define the size of further tables that depend on the factorials.

This function is `constexpr` only if the compiler supports C++14 constexpr functions.

Accuracy

For arguments smaller than `max_factorial<T>::value` the result should be correctly rounded. For larger arguments the accuracy will be the same as for tgamma.

Testing

Basic sanity checks and spot values to verify the data tables: the main tests for the tgamma function handle those cases already.

Implementation

The factorial function is table driven for small arguments, and is implemented in terms of tgamma for larger arguments.

Double Factorial

```
#include <boost/math/special_functions/factorials.hpp>
```

```
namespace boost{ namespace math{

template <class T>
T double_factorial(unsigned i);

template <class T, class Policy>
T double_factorial(unsigned i, const Policy&);

}} // namespaces
```

Returns `i!!`.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

May return the result of overflow_error if the result is too large to represent in type T. The implementation is designed to be optimised for small *i* where table lookup of i! is possible.

Important

The functions described above are templates where the template argument T can not be deduced from the arguments passed to the function. Therefore if you write something like:

`boost::math::double_factorial(2);`

You will get a (possibly perplexing) compiler error, ususally indicating that there is no such function to be found. Instead you need to specifiy the return type explicity and write:

`boost::math::double_factorial<double>(2);`

So that the return type is known. Further, the template argument must be a real-valued type such as `float` or `double` and not an integer type - that would overflow far too easily!

The source code `static_assert` and comment just after the will be:

```
BOOST_STATIC_ASSERT(!boost::is_integral<T>::value);
// factorial<unsigned int>(n) is not implemented
// because it would overflow integral type T for too small n
// to be useful. Use instead a floating-point type,
// and convert to an unsigned type if essential, for example:
// unsigned int nfac = static_cast<unsigned int>(factorial<double>(n));
// See factorial documentation for more detail.
```

Note

The argument to `double_factorial` is type `unsigned` even though technically -1!! is defined.

Accuracy

The implementation uses a trivial adaptation of the factorial function, so error rates should be no more than a couple of epsilon higher.

Testing

The spot tests for the double factorial use data generated by functions.wolfram.com.

Implementation

The double factorial is implemented in terms of the factorial and gamma functions using the relations:

(2n)!! = 2^n * n!

(2n+1)!! = (2n+1)! / (2^n n!)

and

(2n-1)!! = Γ((2n+1)/2) * 2^n / sqrt(pi)

Rising Factorial

```
#include <boost/math/special_functions/factorials.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type rising_factorial(T x, int i);

template <class T, class Policy>
calculated-result-type rising_factorial(T x, int i, const Policy&);

}} // namespaces
```

Returns the rising factorial of *x* and *i*:

rising_factorial(x, i) = Γ(x + i) / Γ(x);

or

rising_factorial(x, i) = x(x+1)(x+2)(x+3)...(x+i-1)

Note that both *x* and *i* can be negative as well as positive.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

May return the result of overflow_error if the result is too large to represent in type T.

The return type of these functions is computed using the *result type calculation rules*: the type of the result is `double` if T is an integer type, otherwise the type of the result is T.

Accuracy

The accuracy will be the same as the tgamma_delta_ratio function.

Testing

The spot tests for the rising factorials use data generated by functions.wolfram.com.

Implementation

Rising and falling factorials are implemented as ratios of gamma functions using tgamma_delta_ratio. Optimisations for small integer arguments are handled internally by that function.

Falling Factorial

```
#include <boost/math/special_functions/factorials.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type falling_factorial(T x, unsigned i);

template <class T, class Policy>
calculated-result-type falling_factorial(T x, unsigned i, const Policy&);

}} // namespaces
```

Returns the falling factorial of *x* and *i*:

falling_factorial(x, i) = x(x-1)(x-2)(x-3)...(x-i+1)

Note that this function is only defined for positive *i*, hence the `unsigned` second argument. Argument *x* can be either positive or negative however.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

May return the result of overflow_error if the result is too large to represent in type T.

The return type of these functions is computed using the *result type calculation rules*: the type of the result is `double` if T is an integer type, otherwise the type of the result is T.

Accuracy

The accuracy will be the same as the tgamma_delta_ratio function.

Testing

The spot tests for the falling factorials use data generated by functions.wolfram.com.

Implementation

Rising and falling factorials are implemented as ratios of gamma functions using tgamma_delta_ratio. Optimisations for small integer arguments are handled internally by that function.

Binomial Coefficients

```
#include <boost/math/special_functions/binomial.hpp>
```

```
namespace boost{ namespace math{

template <class T>
T binomial_coefficient(unsigned n, unsigned k);

template <class T, class Policy>
T binomial_coefficient(unsigned n, unsigned k, const Policy&);

}} // namespaces
```

Returns the binomial coefficient: ${}_nC_k$.

Requires k <= n.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

May return the result of overflow_error if the result is too large to represent in type T.

Important

The functions described above are templates where the template argument T can not be deduced from the arguments passed to the function. Therefore if you write something like:

```
boost::math::binomial_coefficient(10, 2);
```

You will get a compiler error, ususally indicating that there is no such function to be found. Instead you need to specifiy the return type explicity and write:

```
boost::math::binomial_coefficient<double>(10, 2);
```

So that the return type is known. Further, the template argument must be a real-valued type such as `float` or `double` and not an integer type - that would overflow far too easily!

Accuracy

The accuracy will be the same as for the factorials for small arguments (i.e. no more than one or two epsilon), and the beta function for larger arguments.

Testing

The spot tests for the binomial coefficients use data generated by functions.wolfram.com.

Implementation

Binomial coefficients are calculated using table lookup of factorials where possible using:

${}_nC_k = n! / (k!(n-k)!)$

Otherwise it is implemented in terms of the beta function using the relations:

${}_nC_k = 1 / (k * \text{beta}(k, n-k+1))$

and

${}_{n}C_{k}$ = 1 / ((n-k) * beta(k+1, n-k))

Beta Functions

Beta

Synopsis

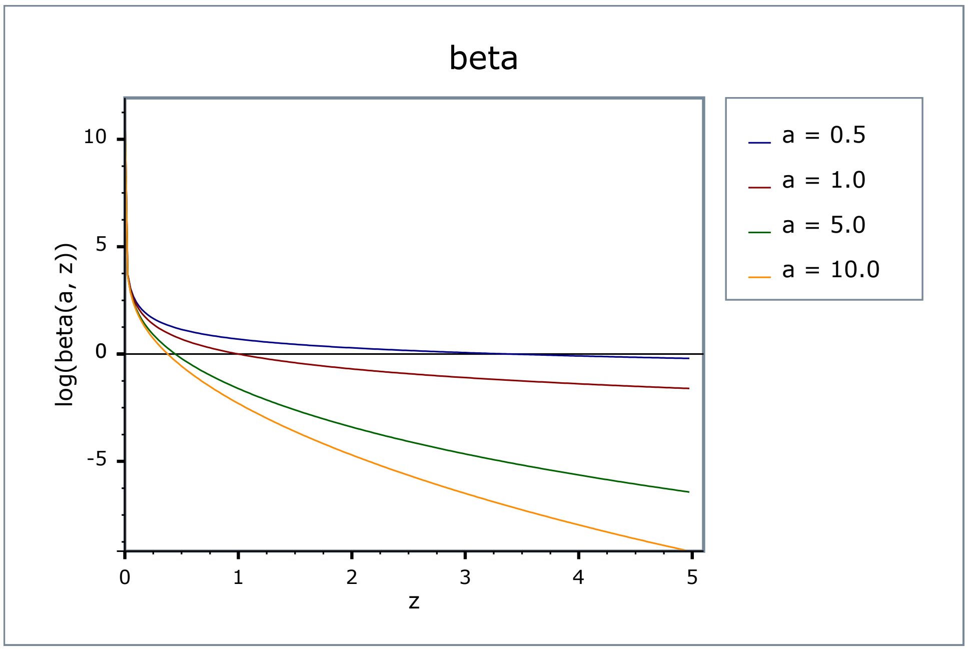
```
#include <boost/math/special_functions/beta.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type beta(T1 a, T2 b);

template <class T1, class T2, class Policy>
calculated-result-type beta(T1 a, T2 b, const Policy&);

}} // namespaces
```

Description

The beta function is defined by:

$$\text{beta}\,(a,\,b) = B(a,\,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

There are effectively two versions of this function internally: a fully generic version that is slow, but reasonably accurate, and a much more efficient approximation that is used where the number of digits in the significand of T correspond to a certain Lanczos approximation. In practice any built-in floating-point type you will encounter has an appropriate Lanczos approximation defined for it. It is also possible, given enough machine time, to generate further Lanczos approximation's using the program libs/math/tools/lanczos_generator.cpp.

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types.

Accuracy

The following table shows peak errors for various domains of input arguments, along with comparisons to the GSL-1.9 and Cephes libraries. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error.

Table 40. Error rates for beta

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = +INFε (Mean = +INFε) And other failures.) (*Rmath 3.2.3:* Max = 1.14ε (Mean = 0.574ε))	Max = 2.86ε (Mean = 1.22ε) (*<cmath>:* Max = 364ε (Mean = 76.2ε))	Max = 2.86ε (Mean = 1.22ε)	Max = 2.23ε (Mean = 1.14ε)
Beta Function: Medium Values	Max = 0.978ε (Mean = 0.0595ε) (*GSL 2.1:* Max = 1.18e+03ε (Mean = 238ε)) (*Rmath 3.2.3:* Max = 1.09e+03ε (Mean = 265ε))	Max = 61.4ε (Mean = 19.4ε) (*<cmath>:* Max = 1.07e+03ε (Mean = 264ε))	Max = 107ε (Mean = 24.5ε)	Max = 96.5ε (Mean = 22.4ε)
Beta Function: Divergent Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 12.1ε (Mean = 1.99ε)) (*Rmath 3.2.3:* Max = 176ε (Mean = 28ε))	Max = 8.99ε (Mean = 2.44ε) (*<cmath>:* Max = 128ε (Mean = 23.8ε))	Max = 18.8ε (Mean = 2.71ε)	Max = 11.4ε (Mean = 2.19ε)

Note that the worst errors occur when a or b are large, and that when this is the case the result is very close to zero, so absolute errors will be very small.

Testing

A mixture of spot tests of exact values, and randomly generated test data are used: the test data was computed using NTL::RR at 1000-bit precision.

Implementation

Traditional methods of evaluating the beta function either involve evaluating the gamma functions directly, or taking logarithms and then exponentiating the result. However, the former is prone to overflows for even very modest arguments, while the latter is prone to cancellation errors. As an alternative, if we regard the gamma function as a white-box containing the Lanczos approximation, then we can combine the power terms:

$$\mathrm{beta}\,(a,\ b) = \left(\frac{a+g-0.5}{a+b+g-0.5}\right)^{a-0.5}\left(\frac{b+g-0.5}{a+b+g-0.5}\right)^{b}\sqrt{\frac{e}{b+g-0.5}}\,\frac{L(a)L(b)}{L(c)}$$

which is almost the ideal solution, however almost all of the error occurs in evaluating the power terms when *a* or *b* are large. If we assume that *a* > *b* then the larger of the two power terms can be reduced by a factor of *b*, which immediately cuts the maximum error in half:

$$\operatorname{beta}(a,\ b) = \left(\frac{a+g-0.5}{a+b+g-0.5}\right)^{a-b-0.5}\left(\frac{(a+g-0.5)(b+g-0.5)}{(a+b+g-0.5)^2}\right)^{b}\sqrt{\frac{e}{b+g-0.5}}\frac{L(a)L(b)}{L(c)}$$

This may not be the final solution, but it is very competitive compared to other implementation methods.

The generic implementation - where no Lanczos approximation approximation is available - is implemented in a very similar way to the generic version of the gamma function. Again in order to avoid numerical overflow the power terms that prefix the series and continued fraction parts are collected together into:

$$e^{lc-la-lb}\left(\frac{la}{lc}\right)^{a}\left(\frac{lb}{lc}\right)^{b}$$

where la, lb and lc are the integration limits used for a, b, and a+b.

There are a few special cases worth mentioning:

When *a* or *b* are less than one, we can use the recurrence relations:

$$\operatorname{beta}(a,\ b) = \frac{(a+b)}{b}\operatorname{beta}(a,\ b+1)$$

$$\operatorname{beta}(a,\ b) = \frac{(a+b)(a+b+1)}{ab}\operatorname{beta}(a+1, b+1)$$

to move to a more favorable region where they are both greater than 1.

In addition:

$$\text{if}\quad a = 1 \qquad \text{then:}\qquad \operatorname{beta}(a,\ b) = \frac{1}{b}$$

Incomplete Beta Functions

Synopsis

```
#include <boost/math/special_functions/beta.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2, class T3>
calculated-result-type ibeta(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta(T1 a, T2 b, T3 x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibetac(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac(T1 a, T2 b, T3 x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type beta(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type beta(T1 a, T2 b, T3 x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type betac(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type betac(T1 a, T2 b, T3 x, const Policy&);

}} // namespaces
```

Description

There are four incomplete beta functions : two are normalised versions (also known as *regularized* beta functions) that return values in the range [0, 1], and two are non-normalised and return values in the range [0, beta(a, b)]. Users interested in statistical applications should use the normalised (or regularized) versions (ibeta and ibetac).

All of these functions require *0 <= x <= 1*.

The normalized functions ibeta and ibetac require *a,b >= 0*, and in addition that not both *a* and *b* are zero.

The functions beta and betac require *a,b > 0*.

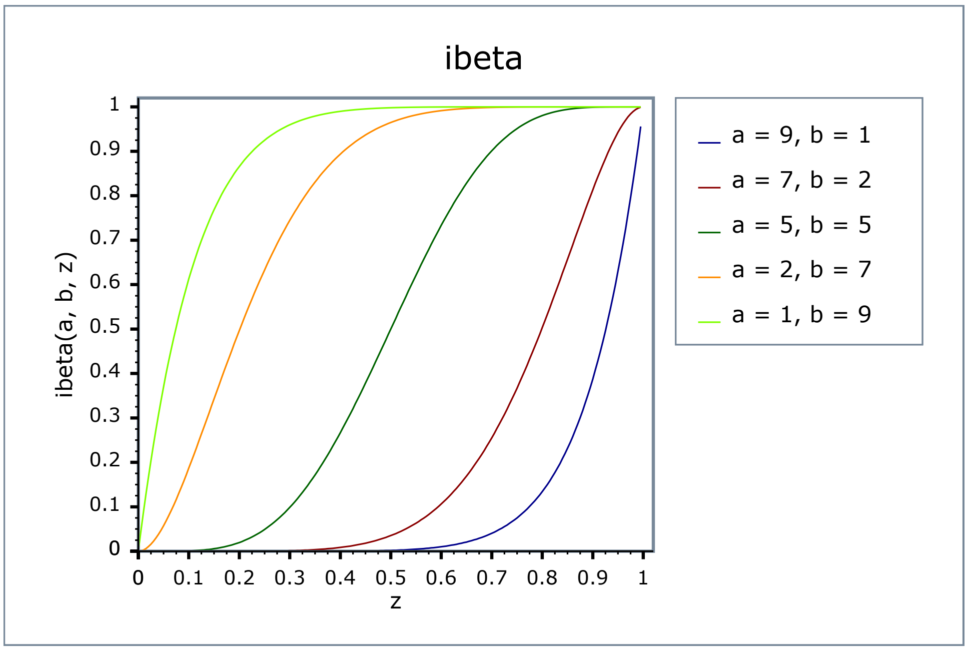
The return type of these functions is computed using the *result type calculation rules* when T1, T2 and T3 are different types.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibeta(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta(T1 a, T2 b, T3 x, const Policy&);
```

Returns the normalised incomplete beta function of a, b and x:

$$\text{ibeta}(a, b, x) = I_x(a, b) = \frac{1}{\text{beta}(a, b)}\int_0^x t^{a-1}(1-t)^{b-1}dt$$



```
template <class T1, class T2, class T3>
calculated-result-type ibetac(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac(T1 a, T2 b, T3 x, const Policy&);
```

Returns the normalised complement of the incomplete beta function of a, b and x:

$$\text{ibetac}(a, b, x) = 1 - I_x(a, b) = I_{1-x}(b, a)$$

```
template <class T1, class T2, class T3>
calculated-result-type beta(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type beta(T1 a, T2 b, T3 x, const Policy&);
```

Returns the full (non-normalised) incomplete beta function of a, b and x:

$$\text{beta}(a, b, x) = B_x(a, b) = \int_0^x t^{a-1}(1-t)^{b-1}dt$$

```
template <class T1, class T2, class T3>
calculated-result-type betac(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type betac(T1 a, T2 b, T3 x, const Policy&);
```

Returns the full (non-normalised) complement of the incomplete beta function of a, b and x:

$$\text{betac}(a, b, x) = 1 - B_x(a, b) = B_{1-x}(b, a)$$

Accuracy

The following tables give peak and mean relative errors in over various domains of a, b and x, along with comparisons to the GSL-1.9 and Cephes libraries. Note that only results for the widest floating-point type on the system are given as narrower types have effectively zero error.

Note that the results for 80 and 128-bit long doubles are noticeably higher than for doubles: this is because the wider exponent range of these types allow more extreme test cases to be tested. For example expected results that are zero at double precision, may be finite but exceptionally small with the wider exponent range of the long double types.

Table 41. Error rates for ibeta

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 682ε (Mean = 32.6ε)) (*Rmath 3.2.3:* Max = 22.9ε (Mean = 3.35ε))	Max = 8.97ε (Mean = 2.09ε)	Max = 21.3ε (Mean = 2.75ε)	Max = 8.4ε (Mean = 1.93ε)
Incomplete Beta Function: Medium Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 690ε (Mean = 151ε)) (*Rmath 3.2.3:* Max = 232ε (Mean = 27.9ε))	Max = 50ε (Mean = 12.1ε)	Max = 124ε (Mean = 18.4ε)	Max = 106ε (Mean = 16.3ε)
Incomplete Beta Function: Large and Diverse Values	Max = 1.26ε (Mean = 0.063ε) (*GSL 2.1:* Max = 3.9e+05ε (Mean = 1.82e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 574ε (Mean = 49.4ε))	Max = 1.96e+04ε (Mean = 997ε)	Max = 4.98e+04ε (Mean = 2.07e+03ε)	Max = 1.32e+03ε (Mean = 68.5ε)
Incomplete Beta Function: Small Integer Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 254ε (Mean = 50.9ε)) (*Rmath 3.2.3:* Max = 62.2ε (Mean = 8.95ε))	Max = 4.45ε (Mean = 0.814ε)	Max = 44.5ε (Mean = 10.1ε)	Max = 3.85ε (Mean = 0.791ε)

Table 42. Error rates for ibetac

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 22.4ε (Mean = 3.67ε))	Max = 10.6ε (Mean = 2.22ε)	Max = 13.8ε (Mean = 2.68ε)	Max = 6.94ε (Mean = 1.71ε)
Incomplete Beta Function: Medium Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 204ε (Mean = 25.8ε))	Max = 73.9ε (Mean = 11.2ε)	Max = 132ε (Mean = 19.8ε)	Max = 56.7ε (Mean = 14.3ε)
Incomplete Beta Function: Large and Diverse Values	Max = 0.981ε (Mean = 0.0573ε) (*Rmath 3.2.3:* Max = 889ε (Mean = 68.4ε))	Max = 3.45e+04ε (Mean = 1.32e+03ε)	Max = 6.31e+04ε (Mean = 2.04e+03ε)	Max = 1.88e+03ε (Mean = 82.7ε)
Incomplete Beta Function: Small Integer Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 84.6ε (Mean = 18ε))	Max = 5.34ε (Mean = 1.11ε)	Max = 107ε (Mean = 17.1ε)	Max = 6.37ε (Mean = 1.03ε)

Table 43. Error rates for beta (incomplete)

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε)	Max = 11.1ε (Mean = 2.32ε)	Max = 18.7ε (Mean = 3.19ε)	Max = 9.94ε (Mean = 2.17ε)
Incomplete Beta Function: Medium Values	Max = 0.568ε (Mean = 0.0254ε)	Max = 69.2ε (Mean = 13.4ε)	Max = 174ε (Mean = 25ε)	Max = 90ε (Mean = 12.7ε)
Incomplete Beta Function: Large and Diverse Values	Max = 0.999ε (Mean = 0.0325ε)	Max = 6.84e+04ε (Mean = 2.76e+03ε)	Max = 6.86e+04ε (Mean = 2.79e+03ε)	Max = 633ε (Mean = 29.7ε)
Incomplete Beta Function: Small Integer Values	Max = 0.786ε (Mean = 0.0323ε)	Max = 11.6ε (Mean = 3.6ε)	Max = 51.8ε (Mean = 11ε)	Max = 26ε (Mean = 6.28ε)

Table 44. Error rates for betac

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Incomplete Beta Function: Small Values	Max = 0.676ε (Mean = 0.0302ε)	Max = 9.92ε (Mean = 2.3ε)	Max = 11.2ε (Mean = 2.94ε)	Max = 8.94ε (Mean = 2.06ε)
Incomplete Beta Function: Medium Values	Max = 0.949ε (Mean = 0.098ε)	Max = 63.5ε (Mean = 13.5ε)	Max = 97.6ε (Mean = 24.3ε)	Max = 90.6ε (Mean = 14.8ε)
Incomplete Beta Function: Large and Diverse Values	Max = 1.12ε (Mean = 0.0458ε)	Max = 1.05e+05ε (Mean = 5.45e+03ε)	Max = 1.04e+05ε (Mean = 5.46e+03ε)	Max = 3.72e+03ε (Mean = 113ε)
Incomplete Beta Function: Small Integer Values	Max = 0.586ε (Mean = 0.0314ε)	Max = 11.1ε (Mean = 3.65ε)	Max = 103ε (Mean = 17.4ε)	Max = 26.2ε (Mean = 6.36ε)

Testing

There are two sets of tests: spot tests compare values taken from Mathworld's online function evaluator with this implementation: they provide a basic "sanity check" for the implementation, with one spot-test in each implementation-domain (see implementation notes below).

Accuracy tests use data generated at very high precision (with NTL RR class set at 1000-bit precision), using the "textbook" continued fraction representation (refer to the first continued fraction in the implementation discussion below). Note that this continued fraction is *not* used in the implementation, and therefore we have test data that is fully independent of the code.

Implementation

This implementation is closely based upon "Algorithm 708; Significant digit computation of the incomplete beta function ratios", DiDonato and Morris, ACM, 1992.

All four of these functions share a common implementation: this is passed both x and y, and can return either p or q where these are related by:

$$p \quad = \quad 1 - q \quad = \quad I_x(a, b) \quad = \quad 1 - I_y(b, a) \quad ; \quad y = 1 - x$$

so at any point we can swap a for b, x for y and p for q if this results in a more favourable position. Generally such swaps are performed so that we always compute a value less than 0.9: when required this can then be subtracted from 1 without undue cancellation error.

The following continued fraction representation is found in many textbooks but is not used in this implementation - it's both slower and less accurate than the alternatives - however it is used to generate test data:

$$\begin{aligned} I_x(a, b) &= \frac{x^a y^b}{aB(a, b)}\left(\cfrac{1}{1 + \cfrac{d_1}{1 + \cfrac{d_2}{1 + \ldots}}}\right) \\ d_{2m+1} &= \frac{-(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} \end{aligned}$$

The following continued fraction is due to Didonato and Morris, and is used in this implementation when a and b are both greater than 1:

$$
\begin{aligned}
I_x(a,b) &= \frac{x^a y^b}{B(a,b)}\left(\cfrac{\alpha_1}{\beta_1+\cfrac{\alpha_2}{\beta_2+\cfrac{\alpha_3}{\beta_3+\ \ldots}}}\right)\\
\alpha_1 = 1 \quad , \quad \alpha_{m+1} &= \frac{(a+m-1)(a+b+m-1)m(b-m)x^2}{(a+2m-1)^2}\\
\beta_{m+1} &= m + \frac{m(b-m)x}{a+2m-1} + \frac{(a+m)(a-(a+b)x+1+m(2-x))}{a+2m+1}
\end{aligned}
$$

For smallish b and x then a series representation can be used:

$$
I_x(a,b) = \frac{x^a}{B(a,b)}\sum_{n=1}^{\infty}\frac{(1-b)^{\overline{n}}x^n}{(a+n)n!}
$$

When b << a then the transition from 0 to 1 occurs very close to x = 1 and some care has to be taken over the method of computation, in that case the following series representation is used:

$$
\begin{aligned}
I_x(a,b) &\approx M\sum_{n=0}^{\infty}p_n J_n(b,u) \quad ; \quad a > b\\
M &= \frac{H(b,u)\Gamma(a+b)}{\Gamma(a)T^b}\\
H(c,u) &= \frac{e^{-u}u^c}{\Gamma(a)}\\
T &= a + \frac{b-1}{2}\\
u &= -T\ln(x)\\
p_0 &= 1\\
p_n &= \frac{(b-1)}{(2n-1)!} + \frac{1}{n}\sum_{m=1}^{n-1}\frac{(mb-n)}{(2m+1)!}p_{n-m}
\end{aligned}
$$

Where Q(a,x) is an incomplete gamma function. Note that this method relies on keeping a table of all the p_n previously computed, which does limit the precision of the method, depending upon the size of the table used.

When *a* and *b* are both small integers, then we can relate the incomplete beta to the binomial distribution and use the following finite sum:

$$
I_x(a,b) = \sum_{i=k+1}^{N}\frac{N!}{i!(N-i)!}x^i y^{(N-i)} \quad ; \quad k = a-1, N = a+b-1
$$

Finally we can sidestep difficult areas, or move to an area with a more efficient means of computation, by using the duplication formulae:

$$
\begin{aligned}
I_x(a,b) &= I_x(a+n,b) + x^a(1-x)^b\sum_{j=1}^{n}\frac{\Gamma(a+b+j-1)}{\Gamma(b)\Gamma(a+j)}x^{j-1}\\
&= I_x(a+n,b) + \frac{x^a(1-x)^b}{a}\sum_{j=0}^{n-1}\frac{(a+b)^{\overline{j}}}{(a+1)^{\overline{j}}}x^j
\end{aligned}
$$

$$B_x(a,b) = \frac{(a+b)^{\overline{n}}}{a^{\overline{n}}} B_x(a+n,b) + \frac{x^a(1-x)^b}{a}\sum_{j=0}^{n-1}\frac{(a+b)^{\overline{j}}}{(a+1)^{\overline{j}}}x^j$$

The domains of a, b and x for which the various methods are used are identical to those described in the Didonato and Morris TOMS 708 paper.

The Incomplete Beta Function Inverses

```
#include <boost/math/special_functions/beta.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2, class T3>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, const Policy&);

template <class T1, class T2, class T3, class T4>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, T4* py);

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, T4* py, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, const Policy&);

template <class T1, class T2, class T3, class T4>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, T4* py);

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, T4* py, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibeta_inva(T1 b, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_inva(T1 b, T2 x, T3 p, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibetac_inva(T1 b, T2 x, T3 q);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_inva(T1 b, T2 x, T3 q, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibeta_invb(T1 a, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_invb(T1 a, T2 x, T3 p, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type ibetac_invb(T1 a, T2 x, T3 q);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_invb(T1 a, T2 x, T3 q, const Policy&);

}} // namespaces
```

Description

There are six incomplete beta function inverses which allow you solve for any of the three parameters to the incomplete beta, starting from either the result of the incomplete beta (p) or its complement (q).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Tip

When people normally talk about the inverse of the incomplete beta function, they are talking about inverting on parameter *x*. These are implemented here as ibeta_inv and ibetac_inv, and are by far the most efficient of the inverses presented here.

The inverses on the *a* and *b* parameters find use in some statistical applications, but have to be computed by rather brute force numerical techniques and are consequently several times slower. These are implemented here as ibeta_inva and ibeta_invb, and complement versions ibetac_inva and ibetac_invb.

The return type of these functions is computed using the *result type calculation rules* when called with arguments T1...TN of different types.

```
template <class T1, class T2, class T3>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, const Policy&);

template <class T1, class T2, class T3, class T4>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, T4* py);

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ibeta_inv(T1 a, T2 b, T3 p, T4* py, const Policy&);
```

Returns a value *x* such that: `p = ibeta(a, b, x);` and sets `*py = 1 - x` when the `py` parameter is provided and is non-null. Note that internally this function computes whichever is the smaller of `x` and `1-x`, and therefore the value assigned to `*py` is free from cancellation errors. That means that even if the function returns `1`, the value stored in `*py` may be non-zero, albeit very small.

Requires: *a,b > 0* and *0 <= p <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, const Policy&);

template <class T1, class T2, class T3, class T4>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, T4* py);

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ibetac_inv(T1 a, T2 b, T3 q, T4* py, const Policy&);
```

Returns a value *x* such that: `q = ibetac(a, b, x);` and sets `*py = 1 - x` when the `py` parameter is provided and is non-null. Note that internally this function computes whichever is the smaller of `x` and `1-x`, and therefore the value assigned to `*py` is free from cancellation errors. That means that even if the function returns `1`, the value stored in `*py` may be non-zero, albeit very small.

Requires: *a,b > 0* and *0 <= q <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibeta_inva(T1 b, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_inva(T1 b, T2 x, T3 p, const Policy&);
```

Returns a value *a* such that: `p = ibeta(a, b, x);`

Requires: *b > 0*, *0 < x < 1* and *0 <= p <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibetac_inva(T1 b, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_inva(T1 b, T2 x, T3 p, const Policy&);
```

Returns a value *a* such that: `q = ibetac(a, b, x);`

Requires: *b > 0*, *0 < x < 1* and *0 <= q <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibeta_invb(T1 b, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_invb(T1 b, T2 x, T3 p, const Policy&);
```

Returns a value *b* such that: `p = ibeta(a, b, x);`

Requires: *a > 0*, *0 < x < 1* and *0 <= p <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ibetac_invb(T1 b, T2 x, T3 p);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibetac_invb(T1 b, T2 x, T3 p, const Policy&);
```

Returns a value *b* such that: `q = ibetac(a, b, x);`

Requires: *a > 0*, *0 < x < 1* and *0 <= q <= 1*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

The accuracy of these functions should closely follow that of the regular forward incomplete beta functions. However, note that in some parts of their domain, these functions can be extremely sensitive to changes in input, particularly when the argument *p* (or it's complement *q*) is very close to `0` or `1`.

Comparisons to other libraries are shown below, note that our test data exercises some rather extreme cases in the incomplete beta function which many other libraries fail to handle:

Table 45. Error rates for ibeta_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse incomplete beta	Max = 11ε (Mean = 0.345ε) (*Rmath 3.2.3:* Max = 1.14e+121ε (Mean = 3.28e+119ε) And other failures.)	Max = 3.8e+04ε (Mean = 2.66e+03ε)	Max = 4.07e+04ε (Mean = 2.86e+03ε)	Max = 8.59e+03ε (Mean = 277ε)

Table 46. Error rates for ibetac_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse incomplete beta	Max = 0.977ε (Mean = 0.0976ε) (*Rmath 3.2.3:* Max = 3.01e+132ε (Mean = 8.65e+130ε) And other failures.)	Max = 4.88e+04ε (Mean = 3.16e+03ε)	Max = 5.05e+04ε (Mean = 3.33e+03ε)	Max = 2.93e+03ε (Mean = 198ε)

Table 47. Error rates for ibeta_inva

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse incomplete beta	Max = 0.602ε (Mean = 0.0239ε)	Max = 377ε (Mean = 24.4ε)	Max = 438ε (Mean = 31.3ε)	Max = 242ε (Mean = 22.9ε)

Table 48. Error rates for ibetac_inva

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse incomplete beta	Max = 0.683ε (Mean = 0.0314ε)	Max = 382ε (Mean = 22.2ε)	Max = 315ε (Mean = 23.7ε)	Max = 408ε (Mean = 26.7ε)

Table 49. Error rates for ibeta_invb

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse incomplete beta	Max = 0.765ε (Mean = 0.0422ε)	Max = 407ε (Mean = 27.2ε)	Max = 407ε (Mean = 24.4ε)	Max = 409ε (Mean = 19.3ε)

Table 50. Error rates for ibetac_invb

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse incomplete beta	Max = 0.724ε (Mean = 0.0303ε)	Max = 317ε (Mean = 19.8ε)	Max = 369ε (Mean = 22.6ε)	Max = 271ε (Mean = 16.4ε)

Testing

There are two sets of tests:

- Basic sanity checks attempt to "round-trip" from *a*, *b* and *x* to *p* or *q* and back again. These tests have quite generous tolerances: in general both the incomplete beta and its inverses change so rapidly, that round tripping to more than a couple of significant digits isn't possible. This is especially true when *p* or *q* is very near one: in this case there isn't enough "information content" in the input to the inverse function to get back where you started.
- Accuracy checks using high precision test values. These measure the accuracy of the result, given exact input values.

Implementation of ibeta_inv and ibetac_inv

These two functions share a common implementation.

First an initial approximation to x is computed then the last few bits are cleaned up using Halley iteration. The iteration limit is set to 1/2 of the number of bits in T, which by experiment is sufficient to ensure that the inverses are at least as accurate as the normal incomplete beta functions. Up to 5 iterations may be required in extreme cases, although normally only one or two are required. Further, the number of iterations required decreases with increasing *a* and *b* (which generally form the more important use cases).

The initial guesses used for iteration are obtained as follows:

Firstly recall that:

$$p \quad = \quad 1 - q \quad = \quad I_x(a, b) \quad = \quad 1 - I_y(b, a) \quad ; \quad y = 1 - x$$

We may wish to start from either p or q, and to calculate either x or y. In addition at any stage we can exchange a for b, p for q, and x for y if it results in a more manageable problem.

For `a+b >= 5` the initial guess is computed using the methods described in:

Asymptotic Inversion of the Incomplete Beta Function, by N. M. Temme. Journal of Computational and Applied Mathematics 41 (1992) 145-157.

The nearly symmetrical case (section 2 of the paper) is used for

$$I_x(a, a + \beta) \qquad ; \qquad \beta < \sqrt{a}$$

and involves solving the inverse error function first. The method is accurate to at least 2 decimal digits when `a = 5` rising to at least 8 digits when `a = 10`^{5}.

The general error function case (section 3 of the paper) is used for

$$I_x(a, b) \qquad ; \qquad 0.2 \le \frac{a}{a+b} \le 0.8$$

and again expresses the inverse incomplete beta in terms of the inverse of the error function. The method is accurate to at least 2 decimal digits when `a+b = 5` rising to 11 digits when `a+b = 10`^{5}. However, when the result is expected to be very small, and when a+b is also small, then its accuracy tails off, in this case when $p^{1/a} < 0.0025$ then it is better to use the following as an initial estimate:

$$I_x^{-1}(a, b) \quad \approx \quad (apB(a, b))^{\frac{1}{a}}$$

Finally the for all other cases where `a+b > 5` the method of section 4 of the paper is used. This expresses the inverse incomplete beta in terms of the inverse of the incomplete gamma function, and is therefore significantly more expensive to compute than the other cases. However the method is accurate to at least 3 decimal digits when `a = 5` rising to at least 10 digits when `a = 10`^{5}. This method is limited to a > b, and therefore we need to perform an exchange a for b, p for q and x for y when this is not the case. In addition when p is close to 1 the method is inaccurate should we actually want y rather than x as output. Therefore when q is small ($q^{1/p} < 10^{-3}$) we use:

$$y \quad = \quad 1 - I_x^{-1}(a, b) \quad \approx \quad (bqB(a, b))^{\frac{1}{b}}$$

which is both cheaper to compute than the full method, and a more accurate estimate on q.

When a and b are both small there is a distinct lack of information in the literature on how to proceed. I am extremely grateful to Prof Nico Temme who provided the following information with a great deal of patience and explanation on his part. Any errors that follow are entirely my own, and not Prof Temme's.

When a and b are both less than 1, then there is a point of inflection in the incomplete beta at point `xs = (1 - a) / (2 - a - b)`. Therefore if `p > I`$_x$`(a,b)` we swap a for b, p for q and x for y, so that now we always look for a point x below the point of inflection `xs`, and on a convex curve. An initial estimate for x is made with:

$$x_0 = \frac{x_g}{1 + x_g} \quad ; \quad x_g = (apB(a, b))^{\frac{1}{a}}$$

which is provably below the true value for x: Newton iteration will therefore smoothly converge on x without problems caused by overshooting etc.

When a and b are both greater than 1, but a+b is too small to use the other methods mentioned above, we proceed as follows. Observe that there is a point of inflection in the incomplete beta at `xs = (1 - a) / (2 - a - b)`. Therefore if `p > I`$_x$`(a,b)` we swap a for b, p for q and x for y, so that now we always look for a point x below the point of inflection `xs`, and on a concave curve. An initial estimate for x is made with:

$$I_x^{-1}(a, b) \approx (apB(a, b))^{\frac{1}{a}}$$

which can be improved somewhat to:

$$I_x^{-1}(a, b) \approx (apB(a, b))^{\frac{1}{a}} + \frac{b-1}{a+1}(apB(a, b))^{\frac{2}{a}} + \frac{(b-1)\left(a^2 + 3ba - a + 5b - 4\right)}{2(a+1)^2(a+2)}(apB(a, b))^{\frac{3}{a}}$$

when b and x are both small (I've used b < a and x < 0.2). This actually under-estimates x, which drops us on the wrong side of x for Newton iteration to converge monotonically. However, use of higher derivatives and Halley iteration keeps everything under control.

The final case to be considered if when one of a and b is less than or equal to 1, and the other greater that 1. Here, if b < a we swap a for b, p for q and x for y. Now the curve of the incomplete beta is convex with no points of inflection in [0,1]. For small p, x can be estimated using

$$I_x^{-1}(a, b) \approx (apB(a, b))^{\frac{1}{a}}$$

which under-estimates x, and drops us on the right side of the true value for Newton iteration to converge monotonically. However, when p is large this can quite badly underestimate x. This is especially an issue when we really want to find y, in which case this method can be an arbitrary number of order of magnitudes out, leading to very poor convergence during iteration.

Things can be improved by considering the incomplete beta as a distorted quarter circle, and estimating y from:

$$y = \left(1 - p^{bB(a,b)}\right)^{\frac{1}{b}}$$

This doesn't guarantee that we will drop in on the right side of x for monotonic convergence, but it does get us close enough that Halley iteration rapidly converges on the true value.

Implementation of inverses on the a and b parameters

These four functions share a common implementation.

First an initial approximation is computed for *a* or *b*: where possible this uses a Cornish-Fisher expansion for the negative binomial distribution to get within around 1 of the result. However, when *a* or *b* are very small the Cornish Fisher expansion is not usable, in this case the initial approximation is chosen so that $I_x(a, b)$ is near the middle of the range [0,1].

This initial guess is then used as a starting value for a generic root finding algorithm. The algorithm converges rapidly on the root once it has been bracketed, but bracketing the root may take several iterations. A better initial approximation for *a* or *b* would improve these functions quite substantially: currently 10-20 incomplete beta function invocations are required to find the root.

Derivative of the Incomplete Beta Function

Synopsis

```
#include <boost/math/special_functions/beta.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2, class T3>
calculated-result-type ibeta_derivative(T1 a, T2 b, T3 x);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ibeta_derivative(T1 a, T2 b, T3 x, const Policy&);

}} // namespaces
```

Description

This function finds some uses in statistical distributions: it computes the partial derivative with respect to *x* of the incomplete beta function ibeta.

$$\text{ibeta\_derivative}(a, b, x) = \frac{\partial}{\partial x} I_x(a, b) = \frac{(1-x)^{b-1} x^{a-1}}{B(a, b)}$$

The return type of this function is computed using the *result type calculation rules* when T1, T2 and T3 are different types.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

Almost identical to the incomplete beta function ibeta.

Implementation

This function just expose some of the internals of the incomplete beta function ibeta: refer to the documentation for that function for more information.

Error Functions

Error Functions

Synopsis

```
#include <boost/math/special_functions/erf.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type erf(T z);

template <class T, class Policy>
calculated-result-type erf(T z, const Policy&);

template <class T>
calculated-result-type erfc(T z);

template <class T, class Policy>
calculated-result-type erfc(T z, const Policy&);

}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: the return type is `double` if T is an integer type, and T otherwise.

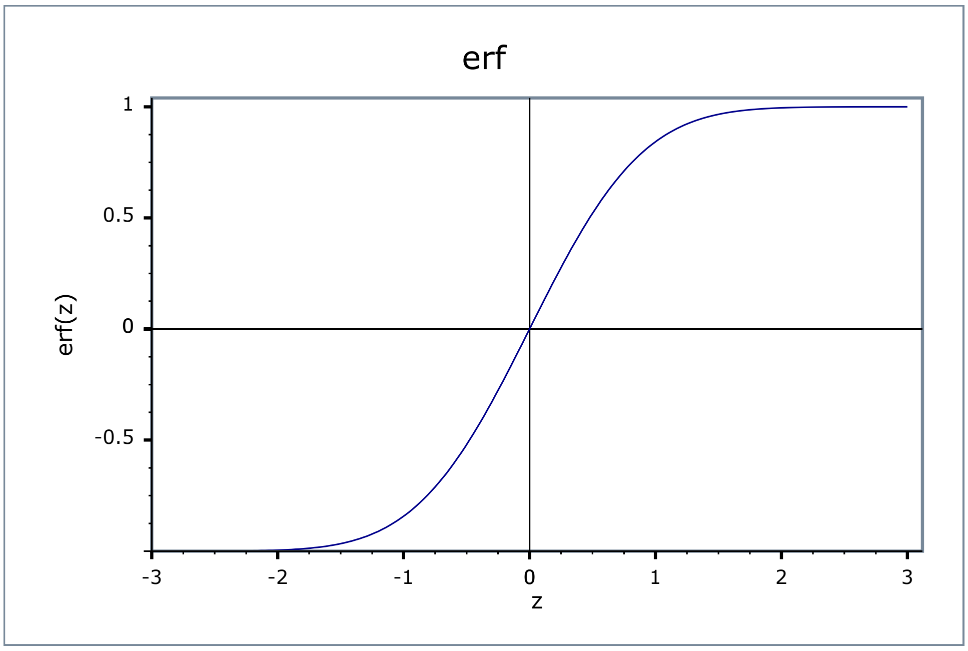
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Description

```
template <class T>
calculated-result-type erf(T z);

template <class T, class Policy>
calculated-result-type erf(T z, const Policy&);
```

Returns the error function erf of z:

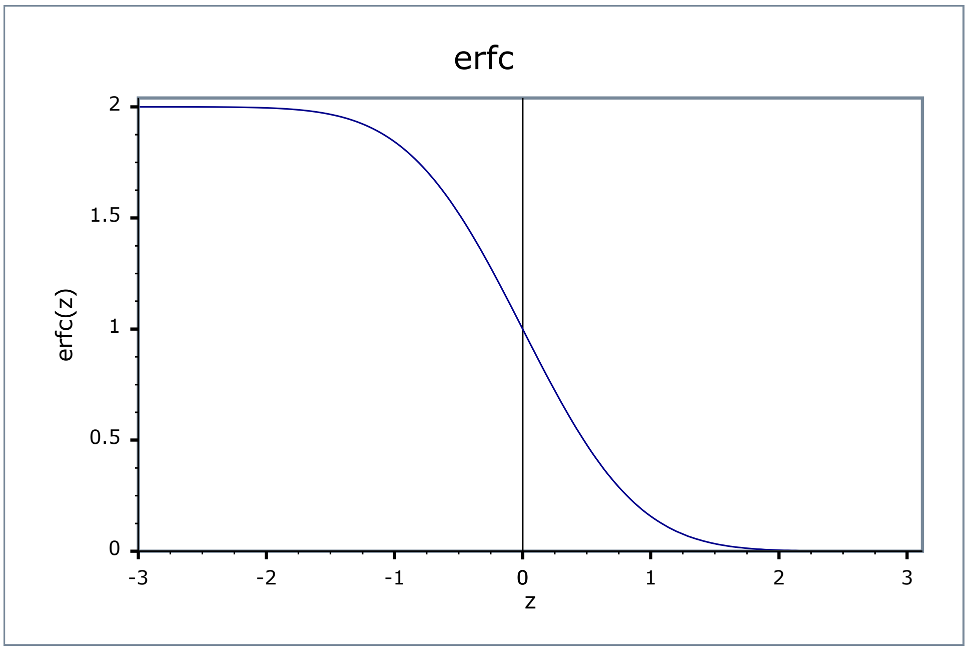
$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}\,dt$$



```
template <class T>
calculated-result-type erfc(T z);

template <class T, class Policy>
calculated-result-type erfc(T z, const Policy&);
```

Returns the complement of the error function of z:

$$\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$$



Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to the GSL-1.9, GNU C Lib, HP-UX C Library and Cephes libraries. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

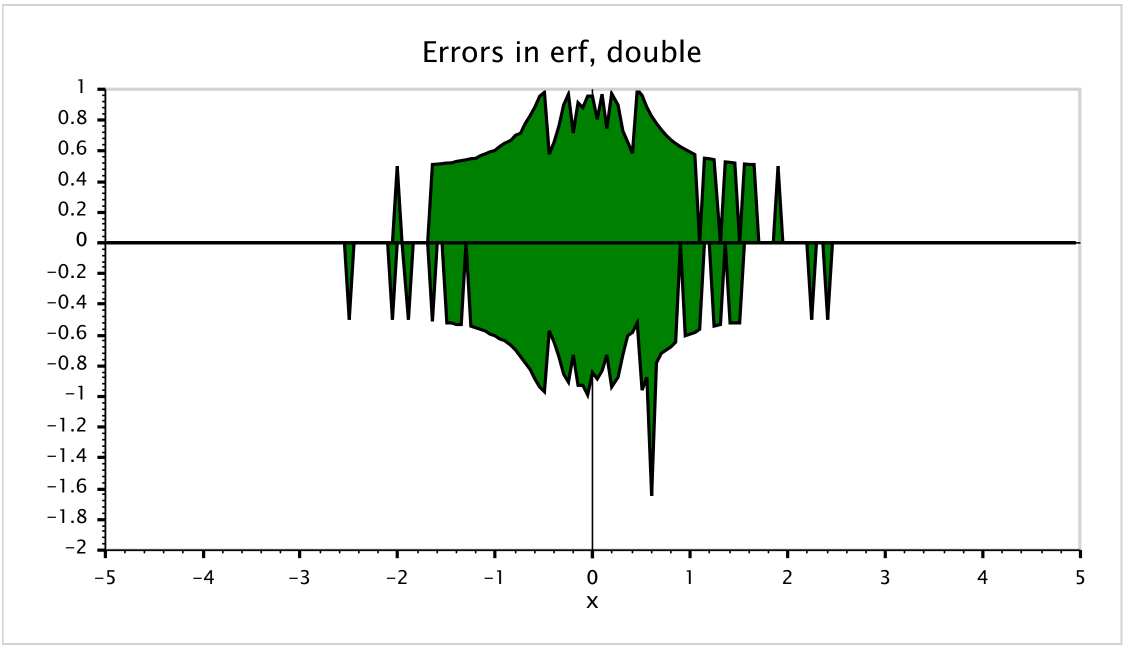
Table 51. Error rates for erf

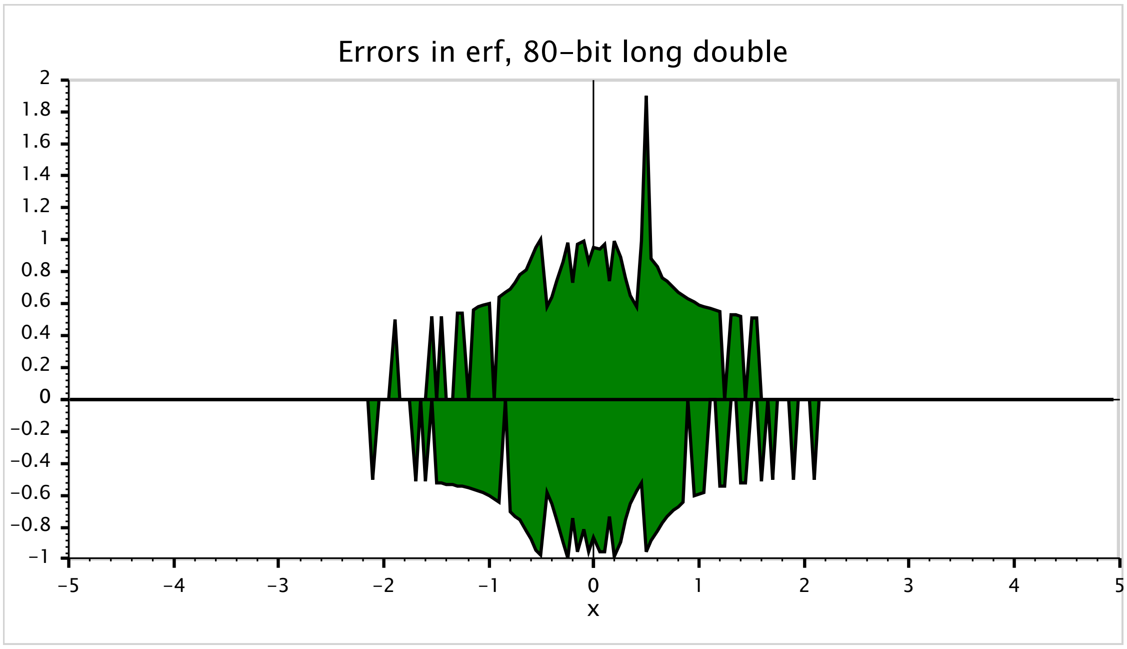
	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Erf Function: Small Values	Max = 0.925ε (Mean = 0.193ε) (*<cmath>:* Max = 0.944ε (Mean = 0.191ε)) (*<math.h>:* Max = 0.944ε (Mean = 0.191ε))	Max = 0.841ε (Mean = 0.0687ε) (*GSL 2.1:* Max = 2.06ε (Mean = 0.319ε))	Max = 0.925ε (Mean = 0.193ε) (*<math.h>:* Max = 0.944ε (Mean = 0.194ε))	Max = 0.996ε (Mean = 0.182ε) (*<math.h>:* Max = 1.57ε (Mean = 0.317ε))
Erf Function: Medium Values	Max = 1.5ε (Mean = 0.193ε) (*<cmath>:* Max = 0.921ε (Mean = 0.0723ε)) (*<math.h>:* Max = 0.921ε (Mean = 0.0723ε))	Max = 1ε (Mean = 0.119ε) (*GSL 2.1:* Max = 2.31ε (Mean = 0.368ε))	Max = 1.5ε (Mean = 0.197ε) (*<math.h>:* Max = 0.921ε (Mean = 0.071ε))	Max = 1ε (Mean = 0.171ε) (*<math.h>:* Max = 1.19ε (Mean = 0.244ε))
Erf Function: Large Values	Max = 0ε (Mean = 0ε) (*<cmath>:* Max = 0ε (Mean = 0ε)) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))

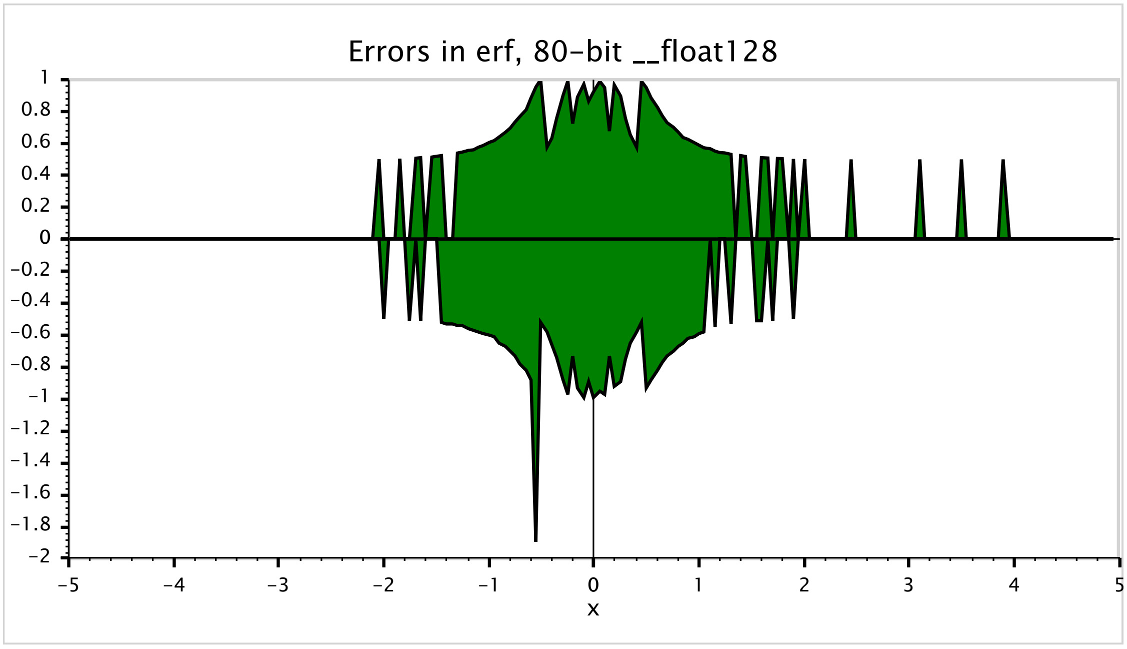
Table 52. Error rates for erfc

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Erf Function: Small Values	Max = 0ε (Mean = 0ε) (*<cmath>:* Max = 0ε (Mean = 0ε)) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0.658ε (Mean = 0.0537ε) (*GSL 2.1:* Max = 1.01ε (Mean = 0.485ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))
Erf Function: Medium Values	Max = 1.76ε (Mean = 0.365ε) (*<cmath>:* Max = 1.35ε (Mean = 0.307ε)) (*<math.h>:* Max = 1.35ε (Mean = 0.307ε))	Max = 0.983ε (Mean = 0.213ε) (*GSL 2.1:* Max = 2.64ε (Mean = 0.662ε))	Max = 1.76ε (Mean = 0.38ε) (*<math.h>:* Max = 2.81ε (Mean = 0.739ε))	Max = 1.65ε (Mean = 0.373ε) (*<math.h>:* Max = 2.36ε (Mean = 0.539ε))
Erf Function: Large Values	Max = 1.57ε (Mean = 0.542ε) (*<cmath>:* Max = 1.26ε (Mean = 0.441ε)) (*<math.h>:* Max = 1.26ε (Mean = 0.441ε))	Max = 0.868ε (Mean = 0.147ε) (*GSL 2.1:* Max = 3.9ε (Mean = 0.472ε))	Max = 1.57ε (Mean = 0.564ε) (*<math.h>:* Max = 4.91ε (Mean = 1.54ε))	Max = 1.14ε (Mean = 0.248ε) (*<math.h>:* Max = 1.84ε (Mean = 0.331ε))

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The tests for these functions come in two parts: basic sanity checks use spot values calculated using Mathworld's online evaluator, while accuracy checks use high-precision test values calculated at 1000-bit precision with NTL::RR and this implementation. Note that the generic and type-specific versions of these functions use differing implementations internally, so this gives us reasonably independent test data. Using our test data to test other "known good" implementations also provides an additional sanity check.

Implementation

All versions of these functions first use the usual reflection formulas to make their arguments positive:

```
erf(-z) = 1 - erf(z);

erfc(-z) = 2 - erfc(z);  // preferred when -z < -0.5

erfc(-z) = 1 + erf(z);   // preferred when -0.5 <= -z < 0
```

The generic versions of these functions are implemented in terms of the incomplete gamma function.

When the significand (mantissa) size is recognised (currently for 53, 64 and 113-bit reals, plus single-precision 24-bit handled via promotion to double) then a series of rational approximations devised by JM are used.

For `z <= 0.5` then a rational approximation to erf is used, based on the observation that erf is an odd function and therefore erf is calculated using:

```
erf(z) = z * (C + R(z*z));
```

where the rational approximation R(z*z) is optimised for absolute error: as long as its absolute error is small enough compared to the constant C, then any round-off error incurred during the computation of R(z*z) will effectively disappear from the result. As a result the error for erf and erfc in this region is very low: the last bit is incorrect in only a very small number of cases.

For `z > 0.5` we observe that over a small interval [a, b) then:

```
erfc(z) * exp(z*z) * z ~ c
```

for some constant c.

Therefore for `z > 0.5` we calculate erfc using:

```
erfc(z) = exp(-z*z) * (C + R(z - B)) / z;
```

Again R(z - B) is optimised for absolute error, and the constant `C` is the average of `erfc(z) * exp(z*z) * z` taken at the end-points of the range. Once again, as long as the absolute error in R(z - B) is small compared to `c` then `c + R(z - B)` will be correctly rounded, and the error in the result will depend only on the accuracy of the exp function. In practice, in all but a very small number of cases, the error is confined to the last bit of the result. The constant `B` is chosen so that the left hand end of the range of the rational approximation is 0.

For large `z` over a range [a, +∞] the above approximation is modified to:

```
erfc(z) = exp(-z*z) * (C + R(1 / z)) / z;
```

Error Function Inverses

Synopsis

```
#include <boost/math/special_functions/erf.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type erf_inv(T p);

template <class T, class Policy>
calculated-result-type erf_inv(T p, const Policy&);

template <class T>
calculated-result-type erfc_inv(T p);

template <class T, class Policy>
calculated-result-type erfc_inv(T p, const Policy&);

}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: the return type is `double` if T is an integer type, and T otherwise.

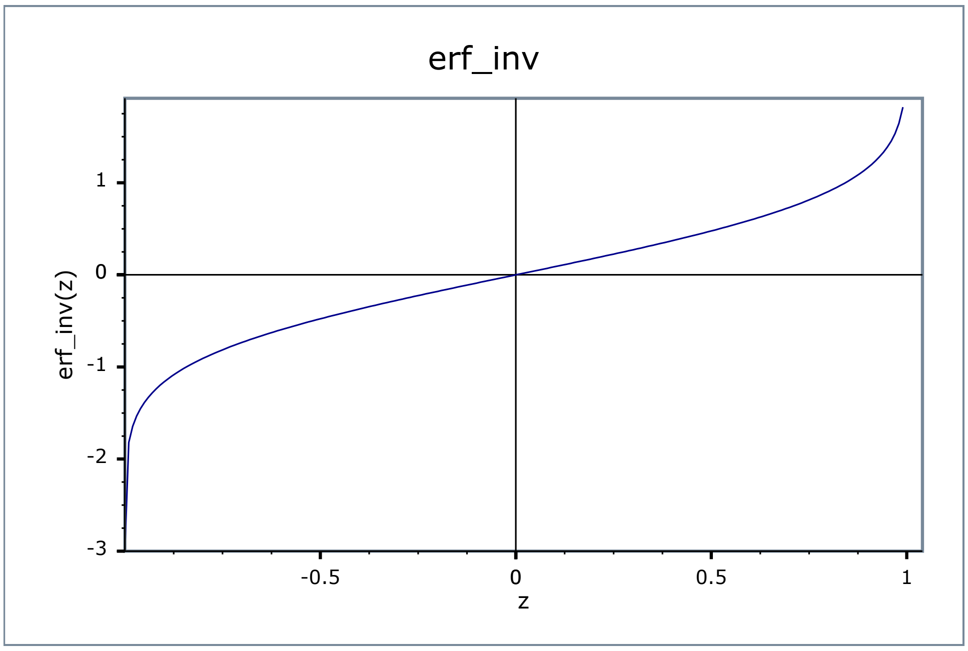
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Description

```
template <class T>
calculated-result-type erf_inv(T z);

template <class T, class Policy>
calculated-result-type erf_inv(T z, const Policy&);
```

Returns the inverse error function of z, that is a value x such that:

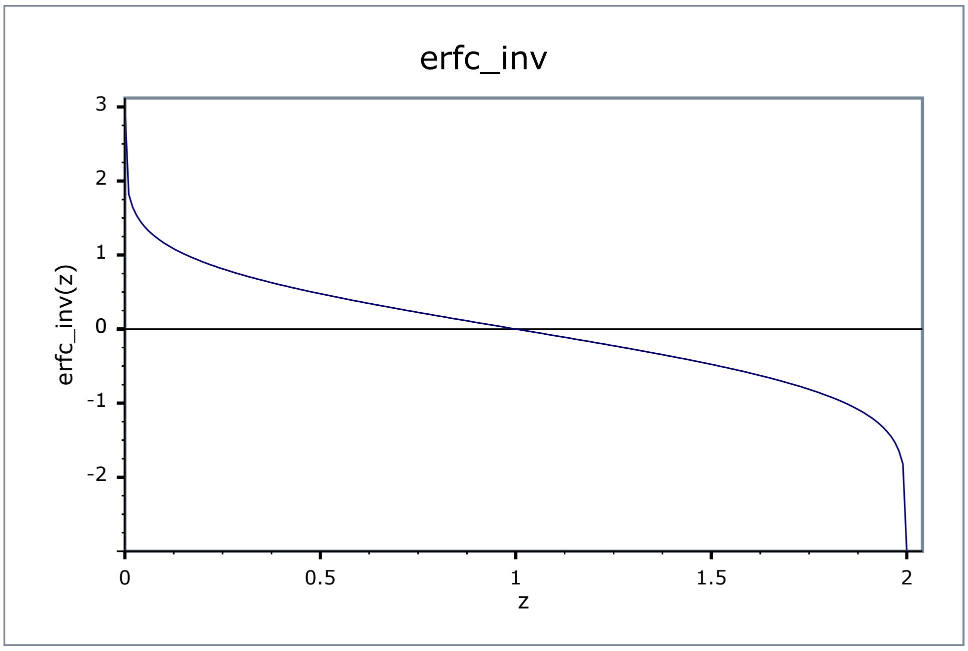
```
p = erf(x);
```



```
template <class T>
calculated-result-type erfc_inv(T z);

template <class T, class Policy>
calculated-result-type erfc_inv(T z, const Policy&);
```

Returns the inverse of the complement of the error function of z, that is a value x such that:

```
p = erfc(x);
```



Accuracy

For types up to and including 80-bit long doubles the approximations used are accurate to less than ~ 2 epsilon. For higher precision types these functions have the same accuracy as the forward error functions.

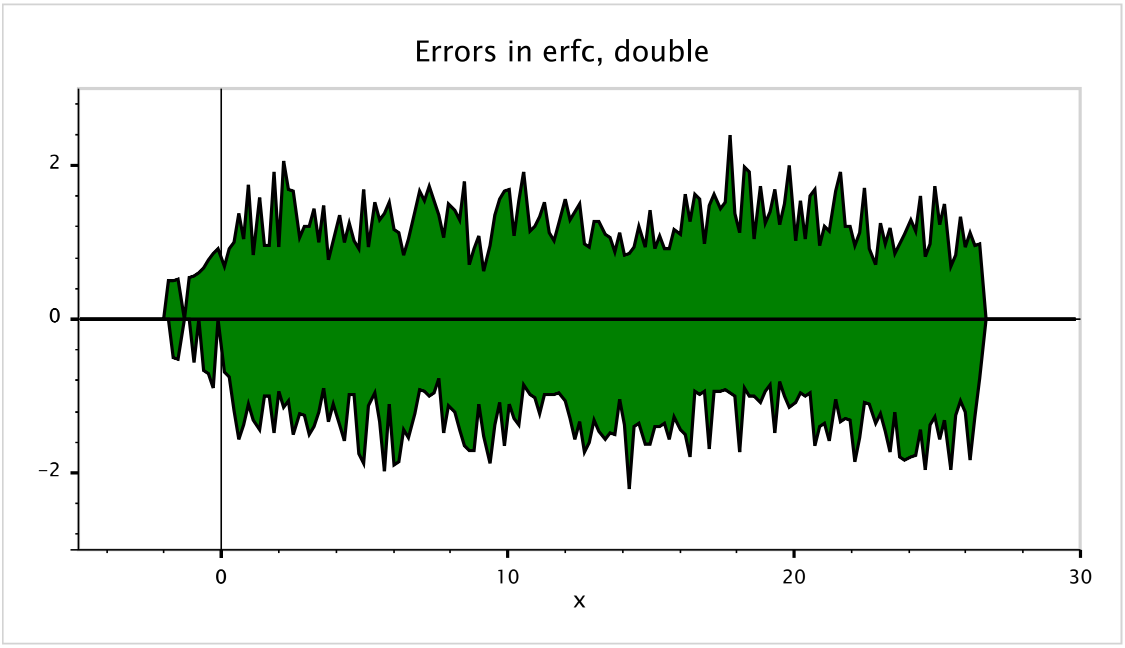
Table 53. Error rates for erf_inv

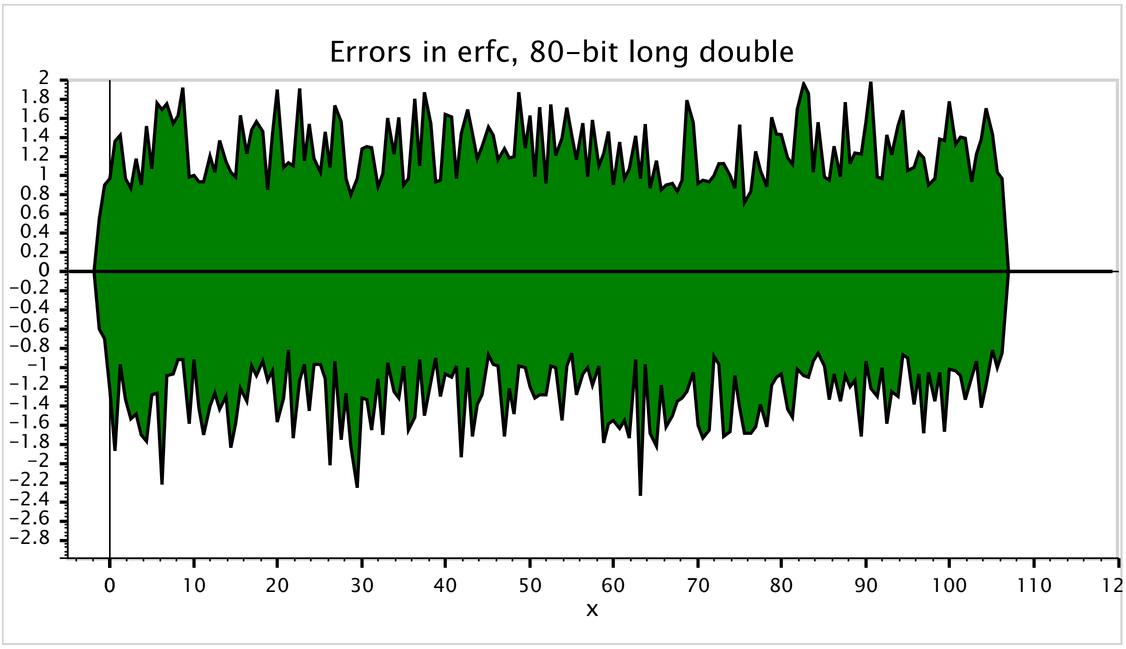
	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse Erf Function	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.389ε)	Max = 1.08ε (Mean = 0.395ε)	Max = 1.09ε (Mean = 0.502ε)

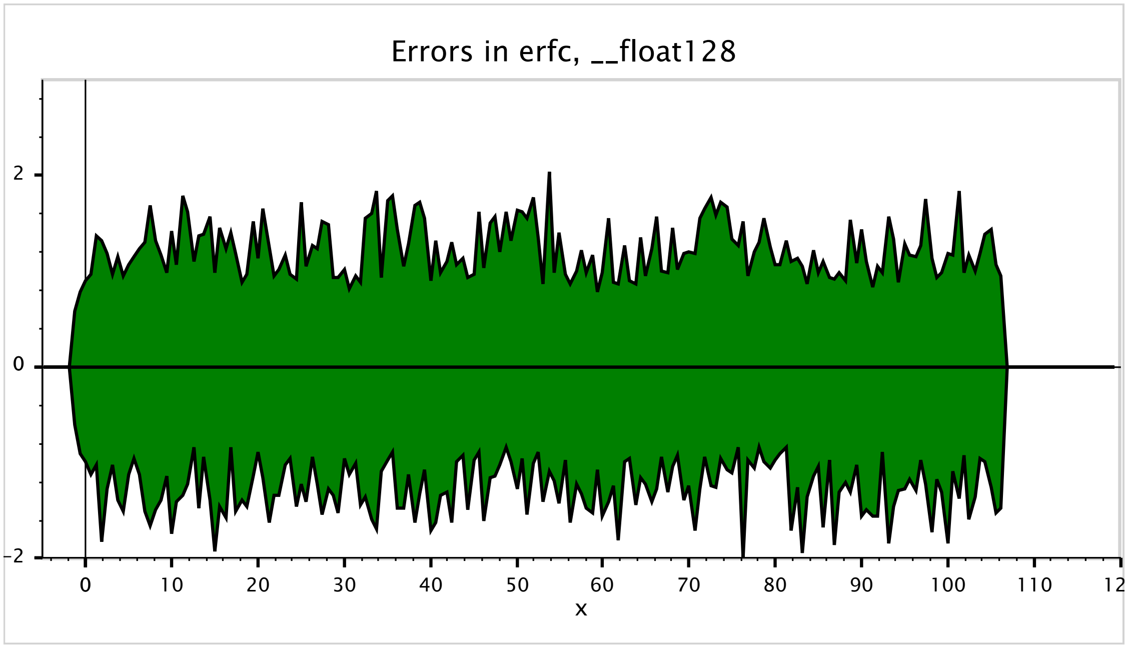
Table 54. Error rates for erfc_inv

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse Erfc Function	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.397ε)	Max = 1.08ε (Mean = 0.403ε)	Max = 1ε (Mean = 0.491ε)
Inverse Erfc Function: extreme values		Max = 1.62ε (Mean = 0.383ε)	Max = 1.62ε (Mean = 0.383ε)	

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

There are two sets of tests:

- Basic sanity checks attempt to "round-trip" from *x* to *p* and back again. These tests have quite generous tolerances: in general both the error functions and their inverses change so rapidly in some places that round tripping to more than a couple of significant digits isn't possible. This is especially true when *p* is very near one: in this case there isn't enough "information content" in the input to the inverse function to get back where you started.
- Accuracy checks using high-precision test values. These measure the accuracy of the result, given *exact* input values.

Implementation

These functions use a rational approximation devised by JM to calculate an initial approximation to the result that is accurate to ~10^{-19}, then only if that has insufficient accuracy compared to the epsilon for T, do we clean up the result using Halley iteration.

Constructing rational approximations to the erf/erfc functions is actually surprisingly hard, especially at high precision. For this reason no attempt has been made to achieve 10^{-34} accuracy suitable for use with 128-bit reals.

In the following discussion, *p* is the value passed to erf_inv, and *q* is the value passed to erfc_inv, so that $p = 1 - q$ and $q = 1 - p$ and in both cases we want to solve for the same result *x*.

For $p < 0.5$ the inverse erf function is reasonably smooth and the approximation:

```
x = p(p + 10)(Y + R(p))
```

Gives a good result for a constant Y, and R(p) optimised for low absolute error compared to |Y|.

For q < 0.5 things get trickier, over the interval $0.5 > q > 0.25$ the following approximation works well:

```
x = sqrt(-2log(q)) / (Y + R(q))
```

While for q < 0.25, let

```
z = sqrt(-log(q))
```

Then the result is given by:

```
x = z(Y + R(z - B))
```

As before Y is a constant and the rational function R is optimised for low absolute error compared to |Y|. B is also a constant: it is the smallest value of *z* for which each approximation is valid. There are several approximations of this form each of which reaches a little further into the tail of the erfc function (at `long double` precision the extended exponent range compared to `double` means that the tail goes on for a very long way indeed).

Polynomials

Legendre (and Associated) Polynomials

Synopsis

```
#include <boost/math/special_functions/legendre.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type legendre_p(int n, T x);

template <class T, class Policy>
calculated-result-type legendre_p(int n, T x, const Policy&);

template <class T>
calculated-result-type legendre_p_prime(int n, T x);

template <class T, class Policy>
calculated-result-type legendre_p_prime(int n, T x, const Policy&);

template <class T, class Policy>
std::vector<T> legendre_p_zeros(int l, const Policy&);

template <class T>
std::vector<T> legendre_p_zeros(int l);

template <class T>
calculated-result-type legendre_p(int n, int m, T x);

template <class T, class Policy>
calculated-result-type legendre_p(int n, int m, T x, const Policy&);

template <class T>
calculated-result-type legendre_q(unsigned n, T x);

template <class T, class Policy>
calculated-result-type legendre_q(unsigned n, T x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type legendre_next(unsigned l, T1 x, T2 Pl, T3 Plm1);

template <class T1, class T2, class T3>
calculated-result-type legendre_next(unsigned l, unsigned m, T1 x, T2 Pl, T3 Plm1);


}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: note than when there is a single template argument the result is the same type as that argument or `double` if the template argument is an integer type.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Description

```
template <class T>
calculated-result-type legendre_p(int l, T x);

template <class T, class Policy>
calculated-result-type legendre_p(int l, T x, const Policy&);
```

Returns the Legendre Polynomial of the first kind:

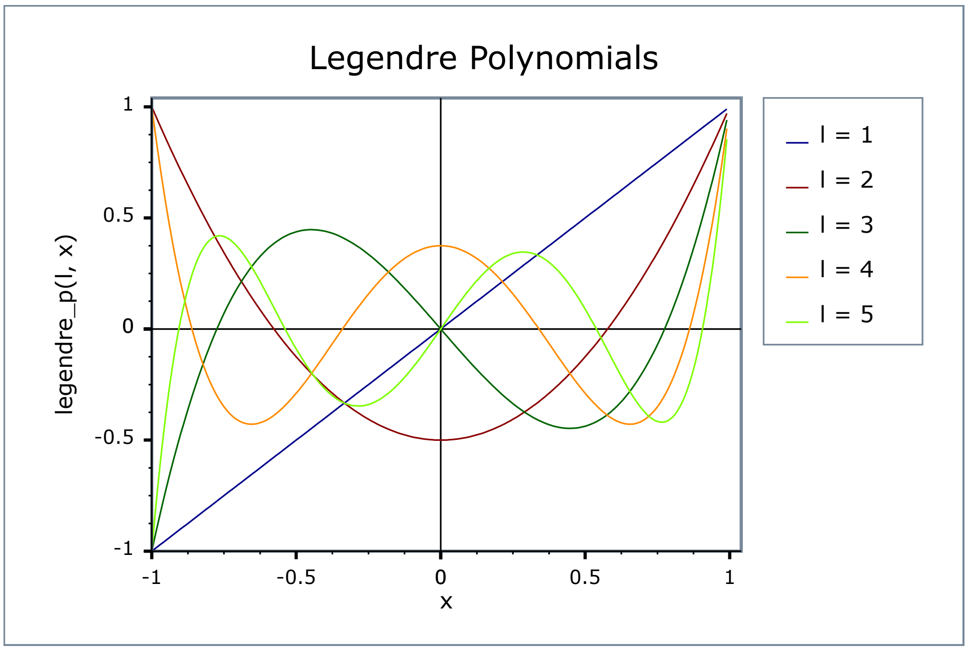
$$\text{legendre\_p}(l, x) \quad = \quad P_l(x) \quad = \quad \frac{1}{2^l l!}\frac{d^l}{dx^l}(x^2-1)^l \quad ; \quad \left|x\right| \le 1$$

Requires -1 <= x <= 1, otherwise returns the result of domain_error.

Negative orders are handled via the reflection formula:

$P_{-l-1}(x) = P_l(x)$

The following graph illustrates the behaviour of the first few Legendre Polynomials:



```
template <class T>
calculated-result-type legendre_p_prime(int n, T x);

template <class T, class Policy>
calculated-result-type legendre_p_prime(int n, T x, const Policy&);
```

Returns the derivatives of the Legendre polynomials.

```
template <class T, class Policy>
std::vector<T> legendre_p_zeros(int l, const Policy&);

template <class T>
std::vector<T> legendre_p_zeros(int l);
```

The zeros of the Legendre polynomials are calculated by Newton's method using an initial guess given by Tricomi with root bracketing provided by Szego.

Since the Legendre polynomials are alternatively even and odd, only the non-negative zeros are returned. For the odd Legendre polynomials, the first zero is always zero. The rest of the zeros are returned in increasing order.

Note that the argument to the routine is an integer, and the output is a floating-point type. Hence the template argument is mandatory. The time to extract a single root is linear in `l` (this is scaling to evaluate the Legendre polynomials), so recovering all roots is (`l`^{2}). Algorithms with linear scaling exist for recovering all roots, but requires tooling not currently built into boost.math. This implementation proceeds under the assumption that calculating zeros of these functions will not be a bottleneck for any workflow.

```
template <class T>
calculated-result-type legendre_p(int l, int m, T x);

template <class T, class Policy>
calculated-result-type legendre_p(int l, int m, T x, const Policy&);
```

Returns the associated Legendre polynomial of the first kind:

$$\text{legendre\_p}\,(l,\, m,\, x) \quad = \quad P_l^m(x) \quad = \quad (-1)^m\left(1-x^2\right)^{\frac{m}{2}}\frac{d^mP_l(x)}{dx^m}$$

Requires -1 <= x <= 1, otherwise returns the result of domain_error.

Negative values of *l* and *m* are handled via the identity relations:

$$\begin{aligned} P_l^{-m}(x) &= (-1)^m\frac{(l-m)!}{(l+m)!}P_l^m(x) \\ P_{-l-1}^m(x) &= P_l^m(x) \end{aligned}$$

Caution

The definition of the associated Legendre polynomial used here includes a leading Condon-Shortley phase term of (-1)^m. This matches the definition given by Abramowitz and Stegun (8.6.6) and that used by Mathworld and Mathematica's LegendreP function. However, uses in the literature do not always include this phase term, and strangely the specification for the associated Legendre function in the C++ TR1 (assoc_legendre) also omits it, in spite of stating that it uses Abramowitz and Stegun as the final arbiter on these matters.

See:

Weisstein, Eric W. "Legendre Polynomial." From MathWorld--A Wolfram Web Resource.

Abramowitz, M. and Stegun, I. A. (Eds.). "Legendre Functions" and "Orthogonal Polynomials." Ch. 22 in Chs. 8 and 22 in Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing. New York: Dover, pp. 331-339 and 771-802, 1972.

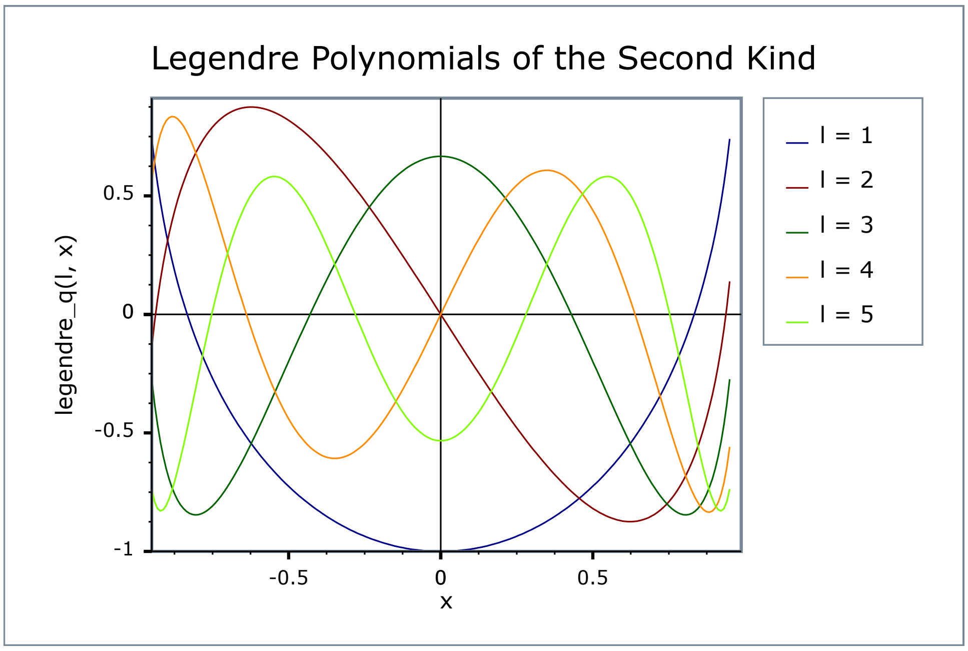
```
template <class T>
calculated-result-type legendre_q(unsigned n, T x);

template <class T, class Policy>
calculated-result-type legendre_q(unsigned n, T x, const Policy&);
```

Returns the value of the Legendre polynomial that is the second solution to the Legendre differential equation, for example:

$$\begin{aligned} \text{legendre\_q}(0,x) &= Q_0(x) = \tfrac{1}{2}\ln\left(\tfrac{1+x}{1-x}\right) \\ \text{legendre\_q}(1,x) &= Q_1(x) = \tfrac{x}{2}\ln\left(\tfrac{1+x}{1-x}\right) - 1 \end{aligned}$$

Requires -1 <= x <= 1, otherwise domain_error is called.

The following graph illustrates the first few Legendre functions of the second kind:



```
template <class T1, class T2, class T3>
calculated-result-type legendre_next(unsigned l, T1 x, T2 Pl, T3 Plm1);
```

Implements the three term recurrence relation for the Legendre polynomials, this function can be used to create a sequence of values evaluated at the same *x*, and for rising *l*. This recurrence relation holds for Legendre Polynomials of both the first and second kinds.

$$P_{l+1}(x) = \frac{(2l+1)xP_l(x) - lP_{l-1}(x)}{(l+1)}$$

For example we could produce a vector of the first 10 polynomial values using:

```
double x = 0.5;  // Abscissa value
vector<double> v;
v.push_back(legendre_p(0, x));
v.push_back(legendre_p(1, x));
for(unsigned l = 1; l < 10; ++l)
   v.push_back(legendre_next(l, x, v[l], v[l-1]));
// Double check values:
for(unsigned l = 1; l < 10; ++l)
   assert(v[l] == legendre_p(l, x));
```

Formally the arguments are:

l	The degree of the last polynomial calculated.
x	The abscissa value
Pl	The value of the polynomial evaluated at degree *l*.
Plm1	The value of the polynomial evaluated at degree *l-1*.

```
template <class T1, class T2, class T3>
calculated-result-type legendre_next(unsigned l, unsigned m, T1 x, T2 Pl, T3 Plm1);
```

Implements the three term recurrence relation for the Associated Legendre polynomials, this function can be used to create a sequence of values evaluated at the same *x*, and for rising *l*.

$$P_{l+1}^{m}(x) \quad = \quad \frac{(2l+1)xP_l^m(x) - (l+m+1)P_{l-1}^m(x)}{(l-m+1)}$$

For example we could produce a vector of the first m+10 polynomial values using:

```
double x = 0.5;  // Abscissa value
int m = 10;      // order
vector<double> v;
v.push_back(legendre_p(m, m, x));
v.push_back(legendre_p(1 + m, m, x));
for(unsigned l = 1; l < 10; ++l)
   v.push_back(legendre_next(l + 10, m, x, v[l], v[l-1]));
// Double check values:
for(unsigned l = 1; l < 10; ++l)
   assert(v[l] == legendre_p(10 + l, m, x));
```

Formally the arguments are:

l	The degree of the last polynomial calculated.
m	The order of the Associated Polynomial.
x	The abscissa value
Pl	The value of the polynomial evaluated at degree *l*.
Plm1	The value of the polynomial evaluated at degree *l-1*.

Accuracy

The following table shows peak errors (in units of epsilon) for various domains of input arguments. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error.

Table 55. Error rates for legendre_p

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Legendre Polynomials: Small Values	Max = 0.732ε (Mean = 0.0619ε) (*GSL 2.1:* Max = 211ε (Mean = 20.4ε))	Max = 69.2ε (Mean = 9.58ε) (*<cmath>:* Max = 124ε (Mean = 13.2ε))	Max = 69.2ε (Mean = 9.58ε)	Max = 211ε (Mean = 20.4ε)
Legendre Polynomials: Large Values	Max = 0.632ε (Mean = 0.0693ε) (*GSL 2.1:* Max = 300ε (Mean = 33.2ε))	Max = 699ε (Mean = 59.6ε) (*<cmath>:* Max = 343ε (Mean = 32.1ε))	Max = 699ε (Mean = 59.6ε)	Max = 300ε (Mean = 33.2ε)

Table 56. Error rates for legendre_q

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Legendre Polynomials: Small Values	Max = 0.612ε (Mean = 0.0517ε) (*GSL 2.1:* Max = 46.4ε (Mean = 7.46ε))	Max = 50.9ε (Mean = 9ε)	Max = 50.9ε (Mean = 8.98ε)	Max = 46.4ε (Mean = 7.32ε)
Legendre Polynomials: Large Values	Max = 2.49ε (Mean = 0.202ε) (*GSL 2.1:* Max = 4.6e+03ε (Mean = 366ε))	Max = 5.98e+03ε (Mean = 478ε)	Max = 5.98e+03ε (Mean = 478ε)	Max = 4.6e+03ε (Mean = 366ε)

Table 57. Error rates for legendre_p (associated)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Associated Legendre Polynomials: Small Values	Max = 0.999ε (Mean = 0.05ε) (*GSL 2.1:* Max = 121ε (Mean = 6.75ε) And other failures.)	Max = 175ε (Mean = 9.88ε) (*<cmath>:* Max = 175ε (Mean = 9.36ε) And other failures.)	Max = 77.7ε (Mean = 5.59ε)	Max = 121ε (Mean = 7.14ε)

Note that the worst errors occur when the order increases, values greater than ~120 are very unlikely to produce sensible results, especially in the associated polynomial case when the degree is also large. Further the relative errors are likely to grow arbitrarily large when the function is very close to a root.

Testing

A mixture of spot tests of values calculated using functions.wolfram.com, and randomly generated test data are used: the test data was computed using NTL::RR at 1000-bit precision.

Implementation

These functions are implemented using the stable three term recurrence relations. These relations guarantee low absolute error but cannot guarantee low relative error near one of the roots of the polynomials.

Legendre-Stieltjes Polynomials

Synopsis

```
#include <boost/math/special_functions/legendre_stieltjes.hpp>


namespace boost{ namespace math{

template <class T>
class legendre_stieltjes
{
public:
    legendre_stieltjes(size_t m);

    Real norm_sq() const;

    Real operator()(Real x) const;

    Real prime(Real x) const;

    std::vector<Real> zeros() const;
}

}}
```

Description

The Legendre-Stieltjes polynomials are a family of polynomials used to generate Gauss-Konrod quadrature formulas. Gauss-Konrod quadratures are algorithms which extend a Gaussian quadrature in such a way that all abscissas are reused when computed a higher-order estimate of the integral, allowing efficient calculation of an error estimate. The Legendre-Stieltjes polynomials assist with this task because their zeros *interlace* the zeros of the Legendre polynomials, meaning that between any two zeros of a Legendre polynomial of degree n, there exists a zero of the Legendre-Stieltjes polynomial of degree n+1.

The Legendre-Stieltjes polynomials E_{n+1} are defined by the property that they have *n* vanishing moments against the oscillatory measure P_n, i.e., $\int_{-1}^{1} E_{n+1}(x)P_n(x)\, x^k\, dx = 0$ for $k = 0, 1, ..., n$. The first few are

- $E_1(x) = P_1(x)$
- $E_2(x) = P_2(x) - 2P_0(x)/5$
- $E_3(x) = P_3(x) - 9P_1(x)/14$
- $E_4(x) = P_4(x) - 20P_2(x)/27 + 14P_0(x)/891$
- $E_5(x) = P_5(x) - 35P_3(x)/44 + 135P_1(x)/12584$

where P_i are the Legendre polynomials. The scaling follows Patterson, who expanded the Legendre-Stieltjes polynomials in a Legendre series and took the coefficient of the highest-order Legendre polynomial in the series to be unity.

The Legendre-Stieltjes polynomials do not satisfy three-term recurrence relations or have a particulary simple representation. Hence the constructor call determines what, in fact, the polynomial is. Once the constructor comes back, the polynomial can be evaluated via the Legendre series.

Example usage:

```
// Call to the constructor determines the coefficients in the Legendre expansion
legendre_stieltjes<double> E(12);
// Evaluate the polynomial at a point:
double x = E(0.3);
// Evaluate the derivative at a point:
double x_p = E.prime(0.3);
// Use the norm_sq to change between scalings, if desired:
double norm = std::sqrt(E.norm_sq());
```

Laguerre (and Associated) Polynomials

Synopsis

```
#include <boost/math/special_functions/laguerre.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type laguerre(unsigned n, T x);

template <class T, class Policy>
calculated-result-type laguerre(unsigned n, T x, const Policy&);

template <class T>
calculated-result-type laguerre(unsigned n, unsigned m, T x);

template <class T, class Policy>
calculated-result-type laguerre(unsigned n, unsigned m, T x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type laguerre_next(unsigned n, T1 x, T2 Ln, T3 Lnm1);

template <class T1, class T2, class T3>
calculated-result-type laguerre_next(unsigned n, unsigned m, T1 x, T2 Ln, T3 Lnm1);


}} // namespaces
```

Description

The return type of these functions is computed using the *result type calculation rules*: note than when there is a single template argument the result is the same type as that argument or `double` if the template argument is an integer type.

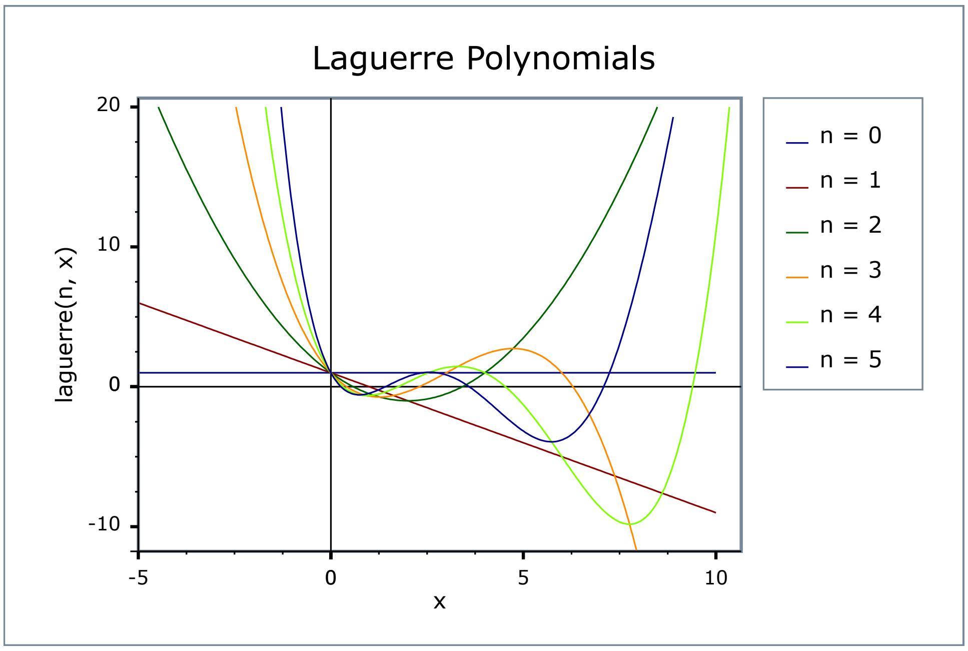
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T>
calculated-result-type laguerre(unsigned n, T x);

template <class T, class Policy>
calculated-result-type laguerre(unsigned n, T x, const Policy&);
```

Returns the value of the Laguerre Polynomial of order *n* at point *x*:

$$\text{laguerre}(n, x) = L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}(x^n e^{-x})$$

The following graph illustrates the behaviour of the first few Laguerre Polynomials:



```
template <class T>
calculated-result-type laguerre(unsigned n, unsigned m, T x);

template <class T, class Policy>
calculated-result-type laguerre(unsigned n, unsigned m, T x, const Policy&);
```

Returns the Associated Laguerre polynomial of degree *n* and order *m* at point *x*:

$$\text{laguerre}\,(n,\ m,\ x) \quad = \quad L_n^m(x) \quad = \quad (-1)^m \frac{d^m}{dx^m} L_{n+m}(x)$$

```
template <class T1, class T2, class T3>
calculated-result-type laguerre_next(unsigned n, T1 x, T2 Ln, T3 Lnm1);
```

Implements the three term recurrence relation for the Laguerre polynomials, this function can be used to create a sequence of values evaluated at the same *x*, and for rising *n*.

$$L_{n+1}(x) \quad = \quad \frac{(2n+1-x)L_n(x) - nL_{n-1}(x)}{(n+1)}$$

For example we could produce a vector of the first 10 polynomial values using:

```
double x = 0.5;  // Abscissa value
vector<double> v;
v.push_back(laguerre(0, x)).push_back(laguerre(1, x));
for(unsigned l = 1; l < 10; ++l)
   v.push_back(laguerre_next(l, x, v[l], v[l-1]));
```

Formally the arguments are:

n	The degree *n* of the last polynomial calculated.
x	The abscissa value

Ln	The value of the polynomial evaluated at degree *n*.
Lnm1	The value of the polynomial evaluated at degree *n-1*.

```
template <class T1, class T2, class T3>
calculated-result-type laguerre_next(unsigned n, unsigned m, T1 x, T2 Ln, T3 Lnm1);
```

Implements the three term recurrence relation for the Associated Laguerre polynomials, this function can be used to create a sequence of values evaluated at the same *x*, and for rising degree *n*.

$$L_{n+1}^{m}(x) \quad = \quad \frac{m+2n+1-x}{n+1} L_{n}^{m}(x) - \frac{m+n}{n+1} L_{n-1}^{m}(x)$$

For example we could produce a vector of the first 10 polynomial values using:

```
double x = 0.5;  // Abscissa value
int m = 10;      // order
vector<double> v;
v.push_back(laguerre(0, m, x)).push_back(laguerre(1, m, x));
for(unsigned l = 1; l < 10; ++l)
   v.push_back(laguerre_next(l, m, x, v[l], v[l-1]));
```

Formally the arguments are:

n	The degree of the last polynomial calculated.
m	The order of the Associated Polynomial.
x	The abscissa value.
Ln	The value of the polynomial evaluated at degree *n*.
Lnm1	The value of the polynomial evaluated at degree *n-1*.

Accuracy

The following table shows peak errors (in units of epsilon) for various domains of input arguments. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error.

Table 58. Error rates for laguerre(n, x)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Laguerre Polynomials	Max = 6.82ε (Mean = 0.408ε) (*GSL 2.1:* Max = 3.1e+03ε (Mean = 185ε))	Max = 1.39e+04ε (Mean = 828ε) (*<cmath>:* Max = 4.2e+03ε (Mean = 251ε))	Max = 1.39e+04ε (Mean = 828ε)	Max = 3.1e+03ε (Mean = 185ε)

Table 59. Error rates for laguerre(n, m, x)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Associated Laguerre Polynomials	Max = 0.84ε (Mean = 0.0358ε) (*GSL 2.1:* Max = 434ε (Mean = 10.7ε))	Max = 167ε (Mean = 6.38ε) (*<cmath>:* Max = 206ε (Mean = 6.86ε))	Max = 167ε (Mean = 6.38ε)	Max = 434ε (Mean = 11.1ε)

Note that the worst errors occur when the degree increases, values greater than ~120 are very unlikely to produce sensible results, especially in the associated polynomial case when the order is also large. Further the relative errors are likely to grow arbitrarily large when the function is very close to a root.

Testing

A mixture of spot tests of values calculated using functions.wolfram.com, and randomly generated test data are used: the test data was computed using NTL::RR at 1000-bit precision.

Implementation

These functions are implemented using the stable three term recurrence relations. These relations guarantee low absolute error but cannot guarantee low relative error near one of the roots of the polynomials.

Hermite Polynomials

Synopsis

```
#include <boost/math/special_functions/hermite.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type hermite(unsigned n, T x);

template <class T, class Policy>
calculated-result-type hermite(unsigned n, T x, const Policy&);

template <class T1, class T2, class T3>
calculated-result-type hermite_next(unsigned n, T1 x, T2 Hn, T3 Hnm1);

}} // namespaces
```

Description

The return type of these functions is computed using the *result type calculation rules*: note than when there is a single template argument the result is the same type as that argument or `double` if the template argument is an integer type.

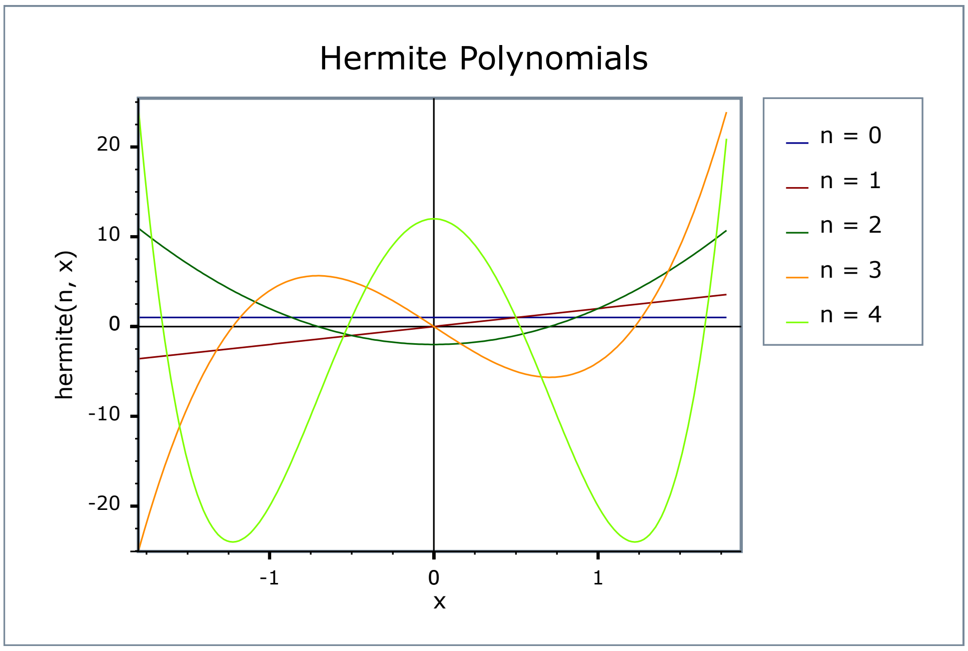
```
template <class T>
calculated-result-type hermite(unsigned n, T x);

template <class T, class Policy>
calculated-result-type hermite(unsigned n, T x, const Policy&);
```

Returns the value of the Hermite Polynomial of order *n* at point *x*:

$$\text{hermite}(x) = H_n(x) = (-1)^n e^{x^2} \frac{d^2}{dx^2} e^{-x^2}$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The following graph illustrates the behaviour of the first few Hermite Polynomials:



```
template <class T1, class T2, class T3>
calculated-result-type hermite_next(unsigned n, T1 x, T2 Hn, T3 Hnm1);
```

Implements the three term recurrence relation for the Hermite polynomials, this function can be used to create a sequence of values evaluated at the same *x*, and for rising *n*.

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x)$$

For example we could produce a vector of the first 10 polynomial values using:

```
double x = 0.5;  // Abscissa value
vector<double> v;
v.push_back(hermite(0, x)).push_back(hermite(1, x));
for(unsigned l = 1; l < 10; ++l)
   v.push_back(hermite_next(l, x, v[l], v[l-1]));
```

Formally the arguments are:

n The degree *n* of the last polynomial calculated.

x	The abscissa value
Hn	The value of the polynomial evaluated at degree *n*.
Hnm1	The value of the polynomial evaluated at degree *n-1*.

Accuracy

The following table shows peak errors (in units of epsilon) for various domains of input arguments. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error.

Table 60. Error rates for hermite

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Hermite Polynomials	Max = 0ε (Mean = 0ε)	Max = 6.24ε (Mean = 2.07ε)	Max = 6.24ε (Mean = 2.07ε)	Max = 4.46ε (Mean = 1.41ε)

Note that the worst errors occur when the degree increases, values greater than ~120 are very unlikely to produce sensible results, especially in the associated polynomial case when the order is also large. Further the relative errors are likely to grow arbitrarily large when the function is very close to a root.

Testing

A mixture of spot tests of values calculated using functions.wolfram.com, and randomly generated test data are used: the test data was computed using NTL::RR at 1000-bit precision.

Implementation

These functions are implemented using the stable three term recurrence relations. These relations guarantee low absolute error but cannot guarantee low relative error near one of the roots of the polynomials.

Chebyshev Polynomials

Synopsis

```
#include <boost/math/special_functions/chebyshev.hpp>
```

```
namespace boost{ namespace math{

template<class Real1, class Real2, class Real3>
calculated-result-type chebyshev_next(Real1 const & x, Real2 const & Tn, Real3 const & Tn_1);

template<class Real>
calculated-result-type chebyshev_t(unsigned n, Real const & x);

template<class Real, class Policy>
calculated-result-type chebyshev_t(unsigned n, Real const & x, const Policy&);

template<class Real>
calculated-result-type chebyshev_u(unsigned n, Real const & x);

template<class Real, class Policy>
calculated-result-type chebyshev_u(unsigned n, Real const & x, const Policy&);

template<class Real>
calculated-result-type chebyshev_t_prime(unsigned n, Real const & x);

template<class Real1, class Real2>
calculated-result-type chebyshev_clenshaw_recur↵
rence(const Real* const c, size_t length, Real2 x);

}} // namespaces
```

“Real analysts cannot do without Fourier, complex analysts cannot do without Laurent, and numerical analysts cannot do without Chebyshev”--Lloyd N. Trefethen

The Chebyshev polynomials of the first kind are defined by the recurrence $T_{n+1}(x) := 2xT_n(x) - T_{n-1}(x)$, $n > 0$, where $T_0(x) := 1$ and $T_1(x) := x$. These can be calculated in Boost using the following simple code

```
double x = 0.5;
double T12 = boost::math::chebyshev_t(12, x);
```

Calculation of derivatives is also straightforward:

```
double T12_prime = boost::math::chebyshev_t_prime(12, x);
```

The complexity of evaluation of the *n*-th Chebyshev polynomial by these functions is linear. So they are unsuitable for use in calculation of (say) a Chebyshev series, as a sum of linear scaling functions scales quadratically. Though there are very sophisticated algorithms for the evaluation of Chebyshev series, a linear time algorithm is presented below:

```
double x = 0.5;
std::vector<double> c{14.2, -13.7, 82.3, 96};
double T0 = 1;
double T1 = x;
double f = c[0]*T0/2;
unsigned l = 1;
while(l < c.size())
{
   f += c[l]*T1;
   std::swap(T0, T1);
   T1 = boost::math::chebyshev_next(x, T0, T1);
   ++l;
}
```

This uses the `chebyshev_next` function to evaluate each term of the Chebyshev series in constant time. However, this naive algorithm has a catastrophic loss of precision as *x* approaches 1. A method to mitigate this way given by Clenshaw, and is implemented in boost as

```
double x = 0.5;
std::vector<double> c{14.2, -13.7, 82.3, 96};
double f = chebyshev_clenshaw_recurrence(c.data(), c.size(), Real x);
```

N.B.: There is factor of *2* difference in our definition of the first coefficient in the Chebyshev series from Clenshaw's original work. This is because two traditions exist in notation for the Chebyshev series expansion,

$f(x) \approx \sum_{n=0}^{N-1} a_n T_n(x)$

and

$f(x) \approx c_0/2 + \sum_{n=1}^{N-1} c_n T_n(x)$

boost math always uses the second convention, with the factor of 1/2 on the first coefficient.

Chebyshev polynomials of the second kind can be evaluated via `chebyshev_u`:

```
double x = -0.23;
double U1 = boost::math::chebyshev_u(1, x);
```

The evaluation of Chebyshev polynomials by a three-term recurrence is known to be mixed forward-backward stable for x [-1, 1]. However, the author does not know of a similar result for *x* outside [-1, 1]. For this reason, evaluation of Chebyshev polynomials outside of [-1, 1] is strongly discouraged. That said, small rounding errors in the course of a computation will often lead to this situation, and termination of the computation due to these small problems is very discouraging. For this reason, `chebyshev_t` and `chebyshev_u` have code paths for $x > 1$ and $x < -1$ which do not use three-term recurrences. These code paths are *much slower*, and should be avoided if at all possible.

Evaluation of a Chebyshev series is relatively simple. The real challenge is *generation* of the Chebyshev series. For this purpose, boost provides a *Chebyshev transform*, a projection operator which projects a function onto a finite-dimensional span of Chebyshev polynomials. But before we discuss the API, let's analyze why we might want to project a function onto a span of Chebyshev polynomials.

- We want a numerically stable way to evaluate the function's derivative
- Our function is expensive to evaluate, and we wish to find a less expensive way to estimate its value. An example are the standard library transcendental functions: These functions are guaranteed to evaluate to within 1 ulp of the exact value, but often this accuracy is not needed. A projection onto the Chebyshev polynomials with a low accuracy requirement can vastly accelerate the computation of these functions.
- We wish to numerically integrate the function

The API is given below.

```
#include <boost/math/special_functions/chebyshev_transform.hpp>
```

```
namespace boost{ namespace math{

template<class Real>
class chebyshev_transform
{
public:
    template<class F>
   chebyshev_transform(const F& f, Real a, Real b, Real tol=500*std::numeric_limits<Real>::ep↵
silon());

    Real operator()(Real x) const

    Real integrate() const

    const std::vector<Real>& coefficients() const

    Real prime(Real x) const
};

}}// end namespaces
```

The Chebyshev transform takes a function f and returns a *near-minimax* approximation to f in terms of Chebyshev polynomials. By *near-minimax*, we mean that the resulting Chebyshev polynomial is "very close" the polynomial p_n which minimizes the uniform norm of f - p_n. The notion of "very close" can be made rigorous; see Trefethen's "Approximation Theory and Approximation Practice" for details.

The Chebyshev transform works by creating a vector of values by evaluating the input function at the Chebyshev points, and then performing a discrete cosine transform on the resulting vector. In order to do this efficiently, we have used FFTW3. So to compile, you must have `FFTW3` installed, and link with `-lfftw3` for double precision, `-lfftw3f` for float precision, `-lfftw3l` for long double precision, and -lfftwq for quad (`__float128`) precision. After the coefficients of the Chebyshev series are known, the routine goes back through them and filters out all the coefficients whose absolute ratio to the largest coefficient are less than the tolerance requested in the constructor.

Spherical Harmonics

Synopsis

```
#include <boost/math/special_functions/spherical_harmonic.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
std::complex<calculated-result-type> spherical_harmonic(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> spherical_harmonic(un↵
signed n, int m, T1 theta, T2 phi, const Policy&);

template <class T1, class T2>
calculated-result-type spherical_harmonic_r(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type spherical_harmonic_r(unsigned n, int m, T1 theta, T2 phi, const Policy&);

template <class T1, class T2>
calculated-result-type spherical_harmonic_i(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type spherical_harmonic_i(unsigned n, int m, T1 theta, T2 phi, const Policy&);

}} // namespaces
```

Description

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2>
std::complex<calculated-result-type> spherical_harmonic(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> spherical_harmonic(un↵
signed n, int m, T1 theta, T2 phi, const Policy&);
```

Returns the value of the Spherical Harmonic Y_n^m(theta, phi):

$$Y_n^m(\theta, \varphi) \quad = \quad \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}}P_n^m(\cos\theta)e^{im\varphi}$$

The spherical harmonics Y_n^m(theta, phi) are the angular portion of the solution to Laplace's equation in spherical coordinates where azimuthal symmetry is not present.

Caution

Care must be taken in correctly identifying the arguments to this function: θ is taken as the polar (colatitudinal) coordinate with θ in [0, π], and ϕ as the azimuthal (longitudinal) coordinate with ϕ in [0,2π). This is the convention used in Physics, and matches the definition used by Mathematica in the function SpericalHarmonicY, but is opposite to the usual mathematical conventions.

Some other sources include an additional Condon-Shortley phase term of $(-1)^m$ in the definition of this function: note however that our definition of the associated Legendre polynomial already includes this term.

This implementation returns zero for m > n

For θ outside [0, π] and ϕ outside [0, 2π] this implementation follows the convention used by Mathematica: the function is periodic with period π in θ and 2π in ϕ. Please note that this is not the behaviour one would get from a casual application of the function's definition. Cautious users should keep θ and ϕ to the range [0, π] and [0, 2π] respectively.

See: Weisstein, Eric W. "Spherical Harmonic." From MathWorld--A Wolfram Web Resource.

```
template <class T1, class T2>
calculated-result-type spherical_harmonic_r(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type spherical_harmonic_r(unsigned n, int m, T1 theta, T2 phi, const Policy&);
```

Returns the real part of Y_n^m(theta, phi):

$$\mathrm{Re}\left(Y_n^m(\theta,\varphi)\right) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}}P_n^m(\cos\theta)\cos(m\varphi)$$

```
template <class T1, class T2>
calculated-result-type spherical_harmonic_i(unsigned n, int m, T1 theta, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type spherical_harmonic_i(unsigned n, int m, T1 theta, T2 phi, const Policy&);
```

Returns the imaginary part of Y_n^m(theta, phi):

$$\mathrm{Im}\left(Y_n^m(\theta,\varphi)\right) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}}P_n^m(\cos\theta)\sin(m\varphi)$$

Accuracy

The following table shows peak errors for various domains of input arguments. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error. Peak errors are the same for both the real and imaginary parts, as the error is dominated by calculation of the associated Legendre polynomials: especially near the roots of the associated Legendre function.

All values are in units of epsilon.

Table 61. Error rates for spherical_harmonic_r

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Spherical Harmonics	Max = 1.58ε (Mean = 0.0707ε)	Max = 2.89e+03ε (Mean = 108ε)	Max = 1.03e+04ε (Mean = 327ε)	Max = 2.27e+04ε (Mean = 725ε)

Table 62. Error rates for spherical_harmonic_i

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Spherical Harmonics	Max = 1.36ε (Mean = 0.0765ε)	Max = 2.89e+03ε (Mean = 108ε)	Max = 1.03e+04ε (Mean = 327ε)	Max = 2.27e+04ε (Mean = 725ε)

Note that the worst errors occur when the degree increases, values greater than ~120 are very unlikely to produce sensible results, especially when the order is also large. Further the relative errors are likely to grow arbitrarily large when the function is very close to a root.

Testing

A mixture of spot tests of values calculated using functions.wolfram.com, and randomly generated test data are used: the test data was computed using NTL::RR at 1000-bit precision.

Implementation

These functions are implemented fairly naively using the formulae given above. Some extra care is taken to prevent roundoff error when converting from polar coordinates (so for example the $1-x^2$ term used by the associated Legendre functions is calculated without roundoff error using $x = cos(theta)$, and $1-x^2 = sin^2(theta)$). The limiting factor in the error rates for these functions is the need to calculate values near the roots of the associated Legendre functions.

Cardinal B-splines

Synopsis

```
#include <boost/math/special_functions/cardinal_b_spline.hpp>
```

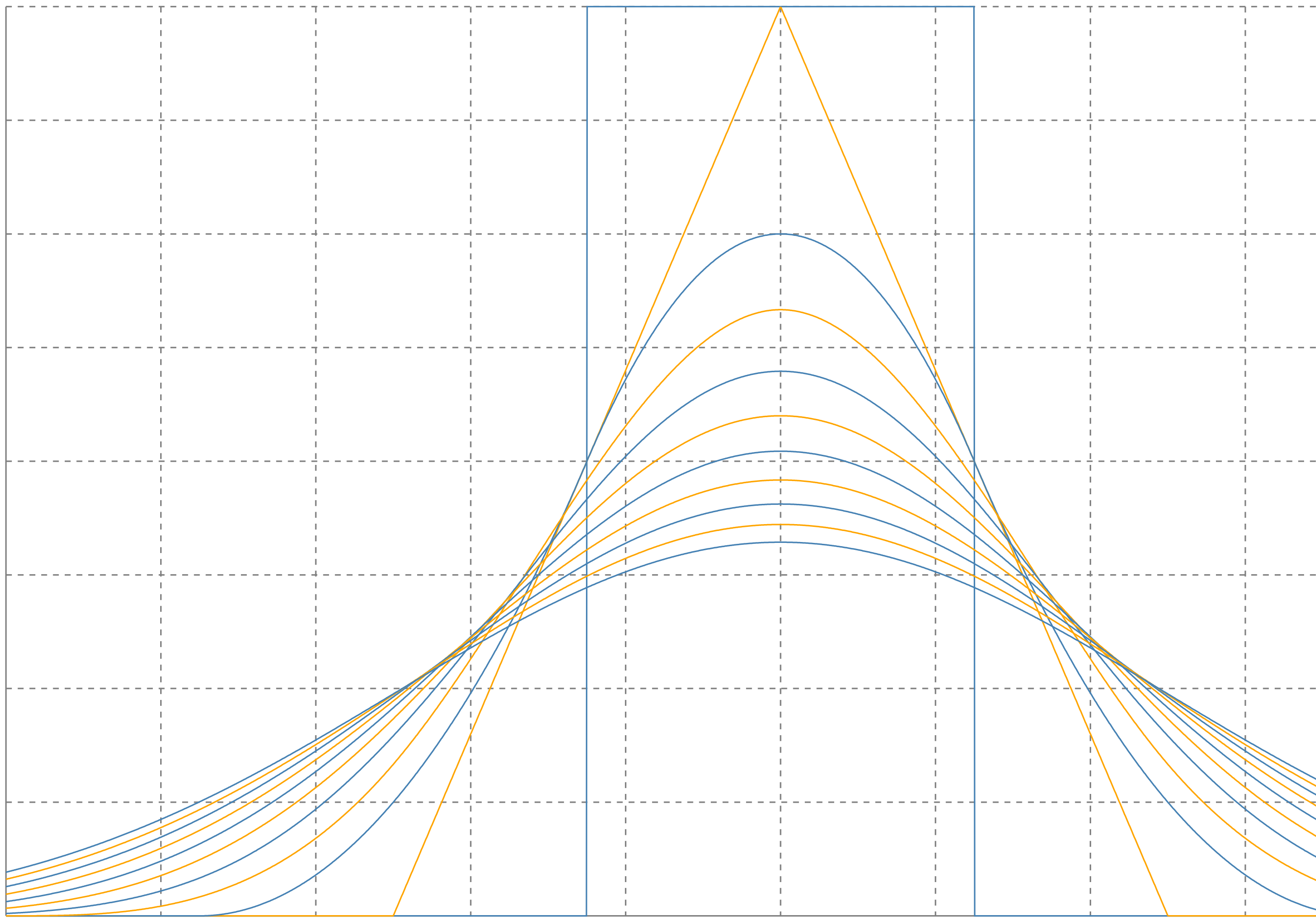
```
namespace boost{ namespace math{

template<unsigned n, typename Real>
Real cardinal_b_spline(Real x);

template<unsigned n, typename Real>
Real forward_cardinal_b_spline(Real x);

}} // namespaces
```

Cardinal *B*-splines are a family of compactly supported functions useful for the smooth interpolation of tables.

The first *B*-spline B_0 is simply a box function: It takes the value one inside the interval [-1/2, 1/2], and is zero elsewhere. *B*-splines of higher smoothness are constructed by iterative convolution, namely, $B_1 := B_0 * B_0$, and $B_{n+1} := B_n * B_0$. For example, $B_1(x) = 1 - |x|$ for x in [-1,1], and zero elsewhere, so it is a hat function.

A basic usage is as follows:

```
using boost::math::cardinal_b_spline;
double x = 0.5;
// B (x), the box function:
double y = cardinal_b_spline<0>(x);
// B (x), the hat function:
y = cardinal_b_spline<1>(x);
```



Caveats

Numerous notational conventions for *B*-splines exist. Whereas Boost.Math (following Kress) zero indexes the *B*-splines, other authors (such as Schoenberg and Massopust) use 1-based indexing. So (for example) Boost.Math's hat function B_1 is what Schoenberg calls M_2. Mathematica, like Boost, uses the zero-indexing convention for its `BSplineCurve`.

Even the support of the splines is not agreed upon. Mathematica starts the support of the splines at zero and rescales the independent variable so that the support of every member is [0, 1]. Massopust as well as Unser puts the support of the *B*-splines at [0, *n*], whereas Kress centers them at zero. Schoenberg distinguishes between the two cases, called the splines starting at zero forward splines, and the ones symmetric about zero *central*.

The *B*-splines of Boost.Math are central, with support support [-(*n*+1)/2, (*n*+1)/2]. If necessary, the forward splines can be evaluated by using `forward_cardinal_b_spline`, whose support is [0, *n*+1].

Implementation

The implementation follows Maurice Cox' 1972 paper 'The Numerical Evaluation of B-splines', and uses the triangular array of Algorithm 6.1 of the reference rather than the rhombohedral array of Algorithm 6.2. Benchmarks revealed that the time to calculate the indexes of the rhombohedral array exceed the time to simply add zeroes together, except for *n* > 18. Since few people use *B* splines of degree 18, the triangular array is used.

Performance

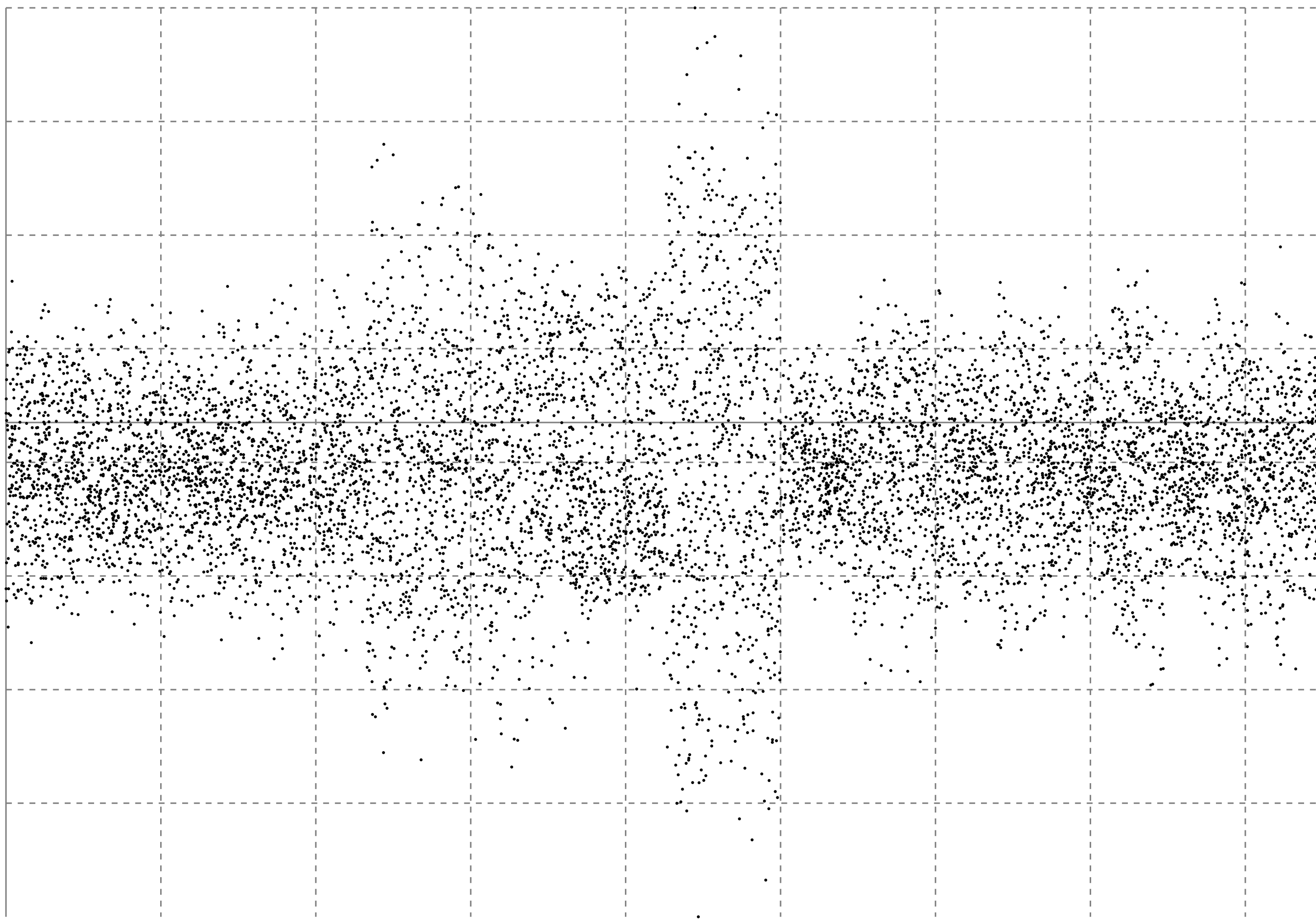
Double precision timing on a consumer x86 laptop is shown below:

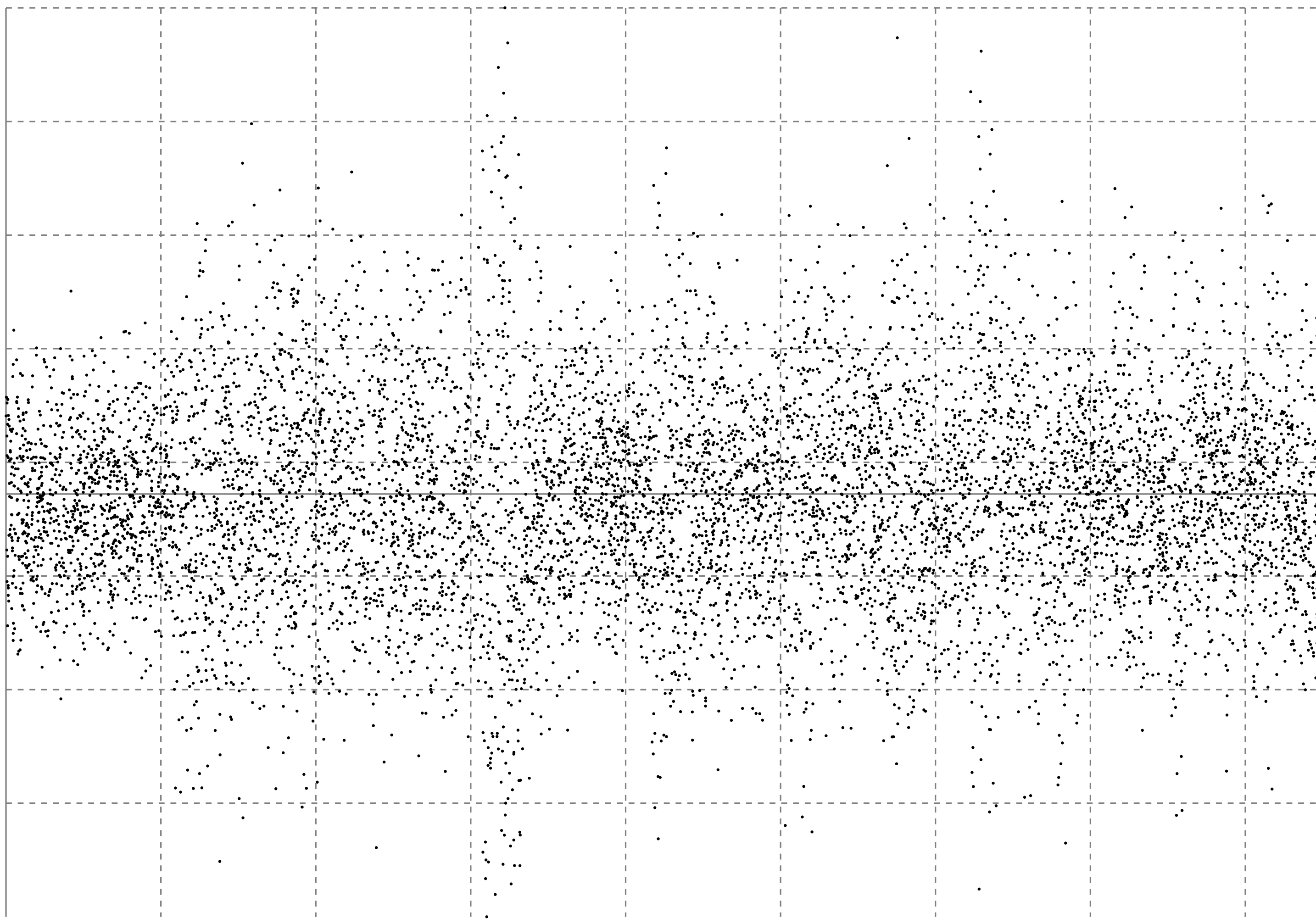
```
Run on (16 X 4300 MHz CPU s)
CPU Caches:
  L1 Data 32K (x8)
  L1 Instruction 32K (x8)
  L2 Unified 1024K (x8)
  L3 Unified 11264K (x1)
Load Average: 0.21, 0.33, 0.29
-----------------------------------------
Benchmark                            Time
-----------------------------------------
CardinalBSpline<1, double>        18.8 ns
CardinalBSpline<2, double>        25.3 ns
CardinalBSpline<3, double>        29.3 ns
CardinalBSpline<4, double>        33.8 ns
CardinalBSpline<5, double>        36.7 ns
CardinalBSpline<6, double>        39.1 ns
CardinalBSpline<7, double>        43.6 ns
CardinalBSpline<8, double>        62.8 ns
CardinalBSpline<9, double>        70.2 ns
CardinalBSpline<10, double>       83.8 ns
CardinalBSpline<11, double>       94.3 ns
CardinalBSpline<12, double>        108 ns
CardinalBSpline<13, double>        122 ns
CardinalBSpline<14, double>        138 ns
CardinalBSpline<15, double>        155 ns
CardinalBSpline<16, double>        170 ns
CardinalBSpline<17, double>        192 ns
CardinalBSpline<18, double>        174 ns
CardinalBSpline<19, double>        180 ns
CardinalBSpline<20, double>        194 ns
UniformReal<double>               11.5 ns
```

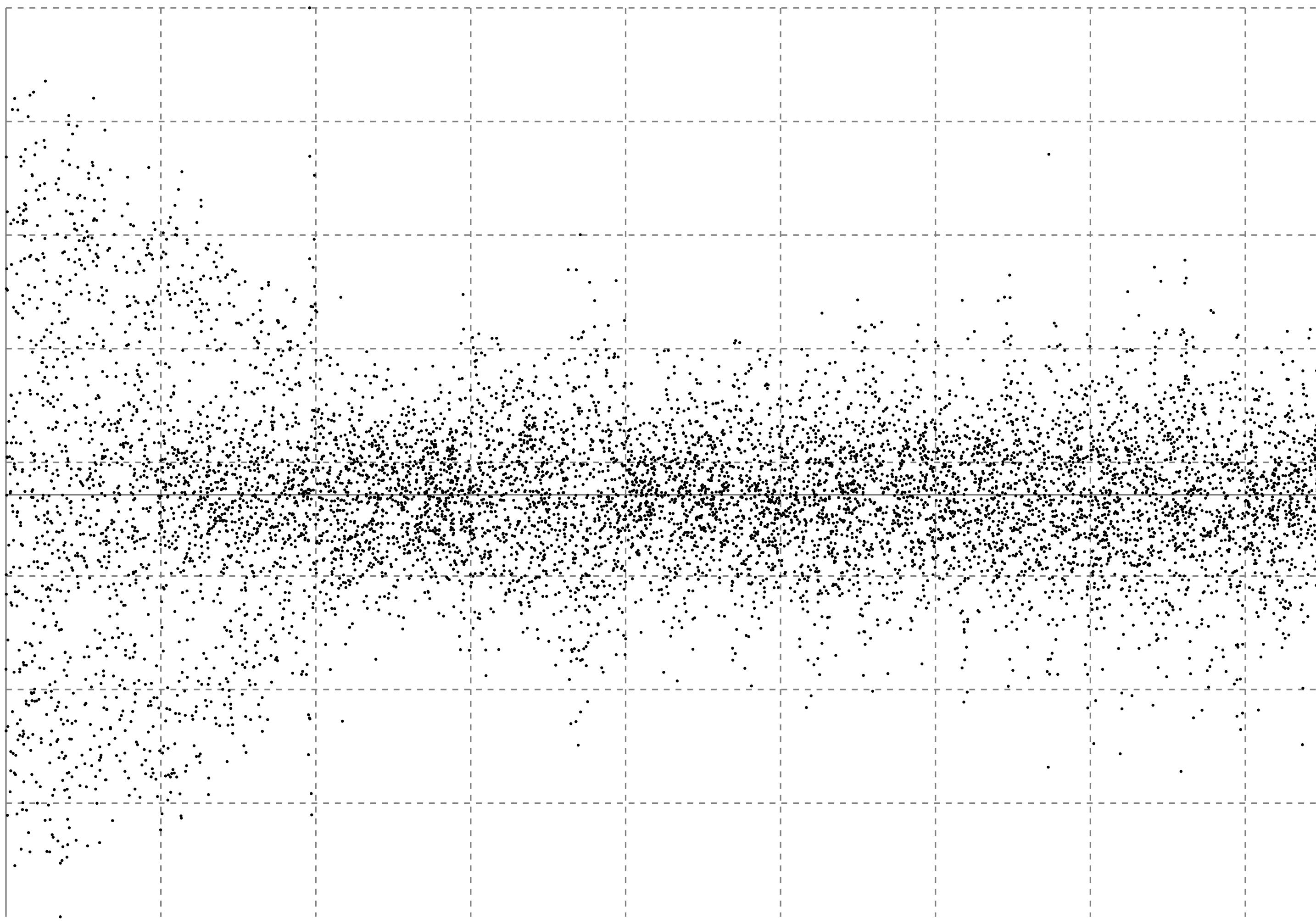
A uniformly distributed random number within the support of the spline is generated for the argument, so subtracting 11.5 ns from these gives a good idea of the performance of the calls.

Accuracy

Some representative ULP plots are shown below. The error grows linearly with *n*, as expected from Cox equation 10.5.







References

- I.J. Schoenberg, *Cardinal Spline Interpolation*, SIAM Volume 12, 1973
- Rainer Kress, *Numerical Analysis*, Springer, 1998
- Peter Massopust, *On Some Generalizations of B-splines*, arxiv preprint, 2019
- Michael Unser and Thierry Blu, *Fractional Splines and Wavelets*, SIAM Review 2000, Volume 42, No. 1
- Cox, Maurice G. *The numerical evaluation of B-splines.*, IMA Journal of Applied Mathematics 10.2 (1972): 134-149.

Bessel Functions

Bessel Function Overview

Ordinary Bessel Functions

Bessel Functions are solutions to Bessel's ordinary differential equation:

$$z^2 \frac{d^2u}{dz^2} + z\frac{du}{dz} + \left(z^2 - \nu^2\right)u = 0$$

where ν is the *order* of the equation, and may be an arbitrary real or complex number, although integer orders are the most common occurrence.

This library supports either integer or real orders.

Since this is a second order differential equation, there must be two linearly independent solutions, the first of these is denoted J_ν and known as a Bessel function of the first kind:

$$J_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

This function is implemented in this library as cyl_bessel_j.

The second solution is denoted either Y_ν or N_ν and is known as either a Bessel Function of the second kind, or as a Neumann function:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

This function is implemented in this library as cyl_neumann.

The Bessel functions satisfy the recurrence relations:

$$J_{\nu+1}(z) = \frac{2\nu}{z}J_\nu(z) - J_{\nu-1}(z)$$

$$Y_{\nu+1}(z) = \frac{2\nu}{z}Y_\nu(z) - Y_{\nu-1}(z)$$

Have the derivatives:

$$J'_\nu(z) = \frac{\nu}{z}J_\nu(z) - J_{\nu+1}(z)$$

$$Y'_\nu(z) = \frac{\nu}{z}Y_\nu(z) - Y_{\nu+1}(z)$$

Have the Wronskian relation:

$$W = J_\nu(z)Y'_\nu(z) - Y_\nu(z)J'_\nu(z) = Y_\nu(z)J_{\nu+1}(z) - J_\nu(z)Y_{\nu+1}(z) = \frac{2}{\pi z}$$

and the reflection formulae:

$$J_{-\nu}(z) = \cos(\nu\pi)J_\nu(z) - \sin(\nu\pi)Y_\nu(z)$$

$$Y_{-v}(z) = \sin(v\pi)J_v(z) + \cos(v\pi)Y_v(z)$$

Modified Bessel Functions

The Bessel functions are valid for complex argument *x*, and an important special case is the situation where *x* is purely imaginary: giving a real valued result. In this case the functions are the two linearly independent solutions to the modified Bessel equation:

$$z^2\frac{d^2u}{dz^2} + z\frac{du}{dz} - (z^2 + v^2)u = 0$$

The solutions are known as the modified Bessel functions of the first and second kind (or occasionally as the hyperbolic Bessel functions of the first and second kind). They are denoted I_v and K_v respectively:

$$I_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(v+k+1)}$$

$$K_v(z) = \frac{\pi}{2} \cdot \frac{I_{-v}(z) - I_v(z)}{\sin(v\pi)}$$

These functions are implemented in this library as cyl_bessel_i and cyl_bessel_k respectively.

The modified Bessel functions satisfy the recurrence relations:

$$I_{v+1}(z) = -\frac{2v}{z}I_v(z) + I_{v-1}(z)$$

$$K_{v+1}(z) = \frac{2v}{z}K_v(z) + K_{v-1}(z)$$

Have the derivatives:

$$I'_v(z) = \frac{v}{z}I_v(z) + I_{v+1}(z)$$

$$K'_v(z) = \frac{v}{z}K_v(z) - K_{v+1}(z)$$

Have the Wronskian relation:

$$W = I_v(z)K'_v(z) - K_v(z)I'_v(z) = -\left[I_v(z)K_{v+1}(z) + K_v(z)I_{v+1}(z)\right] = -\frac{1}{z}$$

and the reflection formulae:

$$I_{-v}(z) = I_v(z) + \frac{2}{\pi}\sin(v\pi)K_v(z)$$

$$K_{-v}(z) = K_v(z)$$

Spherical Bessel Functions

When solving the Helmholtz equation in spherical coordinates by separation of variables, the radial equation has the form:

$$z^2\frac{d^2u}{dz^2} + 2z\frac{du}{dz} + \left[z^2 - n(n+1)\right]u = 0$$

The two linearly independent solutions to this equation are called the spherical Bessel functions j_n and y_n, and are related to the ordinary Bessel functions J_n and Y_n by:

$$\begin{aligned} j_n(z) &= \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z) \\ y_n(z) &= \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z) \end{aligned}$$

The spherical Bessel function of the second kind y_n is also known as the spherical Neumann function n_n.

These functions are implemented in this library as sph_bessel and sph_neumann.

Bessel Functions of the First and Second Kinds

Synopsis

```
#include <boost/math/special_functions/bessel.hpp>
```

```
template <class T1, class T2>
calculated-result-type cyl_bessel_j(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_j(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type cyl_neumann(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_neumann(T1 v, T2 x, const Policy&);
```

Description

The functions cyl_bessel_j and cyl_neumann return the result of the Bessel functions of the first and second kinds respectively:

cyl_bessel_j(v, x) = $J_v(x)$

cyl_neumann(v, x) = $Y_v(x) = N_v(x)$

where:

$$J_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\Gamma(v+k+1)}$$

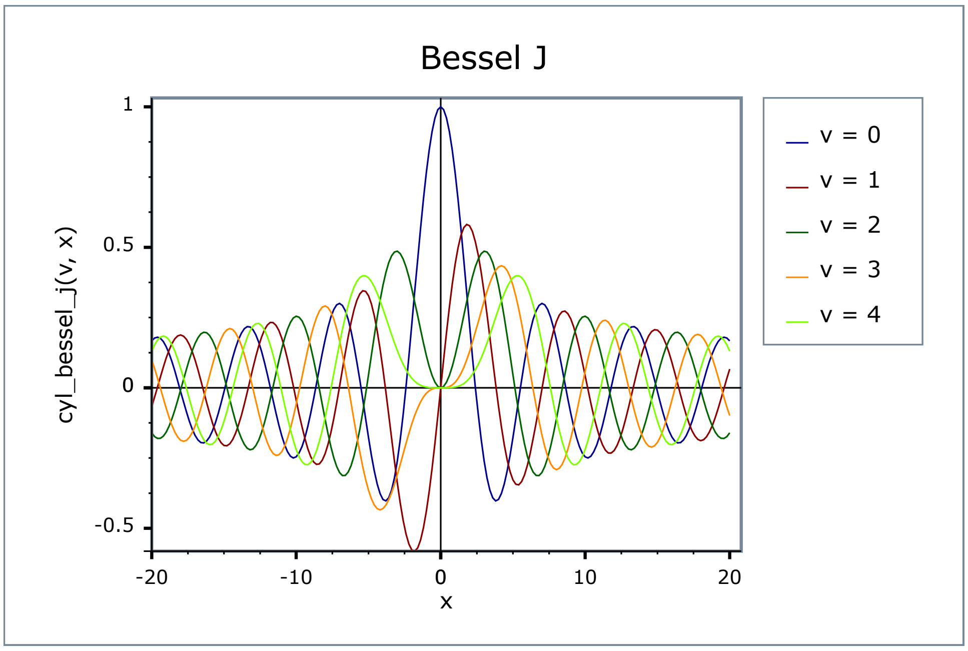
$$Y_v(z) = \frac{J_v(z)\cos(v\pi) - J_{-v}(z)}{\sin(v\pi)}$$

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types. The functions are also optimised for the relatively common case that T1 is an integer.

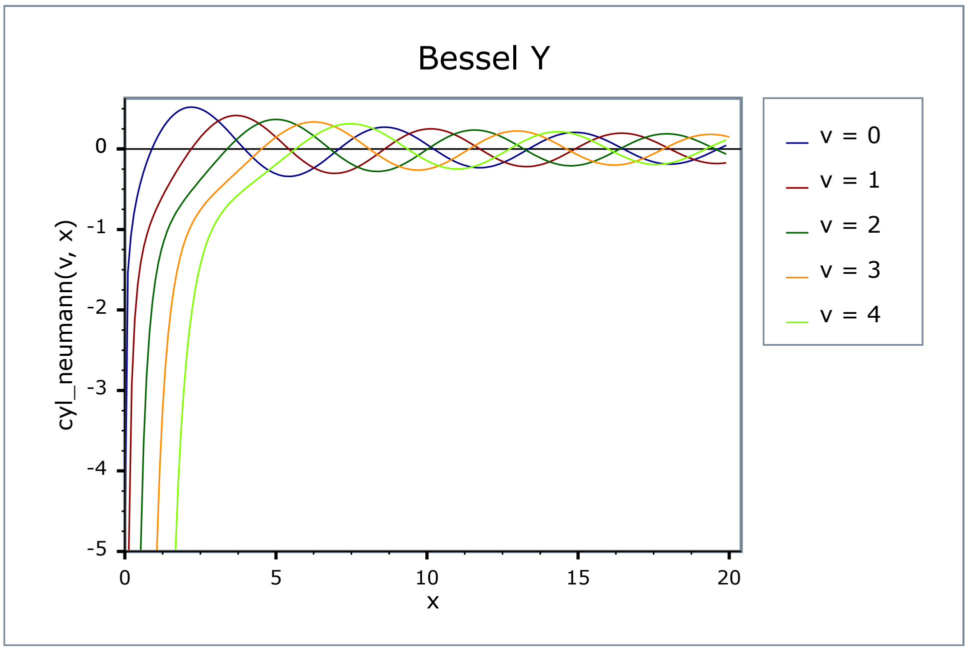
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The functions return the result of domain_error whenever the result is undefined or complex. For cyl_bessel_j this occurs when `x < 0` and v is not an integer, or when `x == 0` and `v != 0`. For cyl_neumann this occurs when `x <= 0`.

The following graph illustrates the cyclic nature of J_v:



The following graph shows the behaviour of Y_v: this is also cyclic for large *x*, but tends to $-\infty$ for small *x*:



Testing

There are two sets of test values: spot values calculated using functions.wolfram.com, and a much larger set of tests computed using a simplified version of this implementation (with all the special case handling removed).

Accuracy

The following tables show how the accuracy of these functions varies on various platforms, along with comparisons to other libraries. Note that the cyclic nature of these functions means that they have an infinite number of irrational roots: in general these functions have arbitrarily large *relative* errors when the arguments are sufficiently close to a root. Of course the absolute error in such cases is always small. Note that only results for the widest floating-point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon. Most of the gross errors exhibited by other libraries occur for very large arguments - you will need to drill down into the actual program output if you need more information on this.

Table 63. Error rates for cyl_bessel_j (integer orders)

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel J0: Mathworld Data (Integer Version)	Max = 6.55ε (Mean = 2.86ε) (*<cmath>:* Max = 5.04ε (Mean = 1.78ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.12ε (Mean = 0.488ε)) (*Rmath 3.2.3:* Max = 0.629ε (Mean = 0.223ε) And other failures.)	Max = 6.55ε (Mean = 2.86ε)	Max = 2.52ε (Mean = 1.2ε) (*<math.h>:* Max = 1.89ε (Mean = 0.988ε))
Bessel J0: Mathworld Data (Tricky cases) (Integer Version)	Max = 1.64e+08ε (Mean = 6.69e+07ε) (*<cmath>:* Max = 4.79e+08ε (Mean = 1.96e+08ε))	Max = 8e+04ε (Mean = 3.27e+04ε) (*GSL 2.1:* Max = 1e+07ε (Mean = 4.11e+06ε)) (*Rmath 3.2.3:* Max = 1.04e+07ε (Mean = 4.29e+06ε))	Max = 1.64e+08ε (Mean = 6.69e+07ε)	Max = 1e+07ε (Mean = 4.09e+06ε) (*<math.h>:* Max = 2.54e+08ε (Mean = 1.04e+08ε))
Bessel J1: Mathworld Data (Integer Version)	Max = 3.59ε (Mean = 1.33ε) (*<cmath>:* Max = 6.1ε (Mean = 2.95ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.89ε (Mean = 0.721ε)) (*Rmath 3.2.3:* Max = 0.946ε (Mean = 0.39ε) And other failures.)	Max = 1.44ε (Mean = 0.637ε)	Max = 1.73ε (Mean = 0.976ε) (*<math.h>:* Max = 11.4ε (Mean = 4.15ε))
Bessel J1: Mathworld Data (tricky cases) (Integer Version)	Max = 2.18e+05ε (Mean = 9.76e+04ε) (*<cmath>:* Max = 2.15e+06ε (Mean = 1.58e+06ε))	Max = 106ε (Mean = 47.5ε) (*GSL 2.1:* Max = 1.26e+06ε (Mean = 6.28e+05ε)) (*Rmath 3.2.3:* Max = 2.93e+06ε (Mean = 1.7e+06ε))	Max = 2.18e+05ε (Mean = 9.76e+04ε)	Max = 3.23e+04ε (Mean = 1.45e+04ε) (*<math.h>:* Max = 1.44e+07ε (Mean = 6.5e+06ε))
Bessel JN: Mathworld Data (Integer Version)	Max = 6.85ε (Mean = 3.35ε) (*<cmath>:* Max = 2.13e+19ε (Mean = 5.16e+18ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.9e+05ε (Mean = 2.53e+05ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 463ε (Mean = 112ε)	Max = 14.7ε (Mean = 5.4ε) (*<math.h>:* Max = +INFε (Mean = +INFε) And other failures.)

Table 64. Error rates for cyl_bessel_j

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel J0: Mathworld Data	Max = 6.55ε (Mean = 2.86ε) (*<cmath>:* Max = 5.04ε (Mean = 1.78ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.629ε (Mean = 0.223ε) And other failures.) (*Rmath 3.2.3:* Max = 0.629ε (Mean = 0.223ε) And other failures.)	Max = 6.55ε (Mean = 2.86ε)	Max = 2.52ε (Mean = 1.2ε)
Bessel J0: Mathworld Data (Tricky cases)	Max = 1.64e+08ε (Mean = 6.69e+07ε) (*<cmath>:* Max = 4.79e+08ε (Mean = 1.96e+08ε))	Max = 8e+04ε (Mean = 3.27e+04ε) (*GSL 2.1:* Max = 6.5e+07ε (Mean = 2.66e+07ε)) (*Rmath 3.2.3:* Max = 1.04e+07ε (Mean = 4.29e+06ε))	Max = 1.64e+08ε (Mean = 6.69e+07ε)	Max = 1e+07ε (Mean = 4.09e+06ε)
Bessel J1: Mathworld Data	Max = 3.59ε (Mean = 1.33ε) (*<cmath>:* Max = 6.1ε (Mean = 2.95ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.62ε (Mean = 2.35ε) And other failures.) (*Rmath 3.2.3:* Max = 0.946ε (Mean = 0.39ε) And other failures.)	Max = 1.44ε (Mean = 0.637ε)	Max = 1.73ε (Mean = 0.976ε)
Bessel J1: Mathworld Data (tricky cases)	Max = 2.18e+05ε (Mean = 9.76e+04ε) (*<cmath>:* Max = 2.15e+06ε (Mean = 1.58e+06ε))	Max = 106ε (Mean = 47.5ε) (*GSL 2.1:* Max = 8.75e+05ε (Mean = 5.32e+05ε)) (*Rmath 3.2.3:* Max = 2.93e+06ε (Mean = 1.7e+06ε))	Max = 2.18e+05ε (Mean = 9.76e+04ε)	Max = 3.23e+04ε (Mean = 1.45e+04ε)
Bessel JN: Mathworld Data	Max = 6.85ε (Mean = 3.35ε) (*<cmath>:* Max = 2.13e+19ε (Mean = 5.16e+18ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.9e+05ε (Mean = 2.15e+05ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 463ε (Mean = 112ε)	Max = 14.7ε (Mean = 5.4ε)

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel J: Mathworld Data	Max = 14.7ε (Mean = 4.11ε) (*<cmath>:* Max = 3.49e+05ε (Mean = 8.09e+04ε) And other failures.)	Max = 10ε (Mean = 2.24ε) (*GSL 2.1:* Max = 2.39e+05ε (Mean = 5.37e+04ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 14.7ε (Mean = 4.22ε)	Max = 14.9ε (Mean = 3.89ε)
Bessel J: Mathworld Data (large values)	Max = 607ε (Mean = 305ε) (*<cmath>:* Max = 34.9ε (Mean = 17.4ε) And other failures.)	Max = 0.536ε (Mean = 0.268ε) (*GSL 2.1:* Max = 4.91e+03ε (Mean = 2.46e+03ε) And other failures.) (*Rmath 3.2.3:* Max = 5.9ε (Mean = 3.76ε))	Max = 607ε (Mean = 305ε)	Max = 9.31ε (Mean = 5.52ε)
Bessel JN: Random Data	Max = 50.8ε (Mean = 3.69ε) (*<cmath>:* Max = 1.12e+03ε (Mean = 88.7ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 75.7ε (Mean = 5.36ε)) (*Rmath 3.2.3:* Max = 3.93ε (Mean = 1.22ε))	Max = 99.6ε (Mean = 22ε)	Max = 17.5ε (Mean = 1.46ε)
Bessel J: Random Data	Max = 11.4ε (Mean = 1.68ε) (*<cmath>:* Max = 501ε (Mean = 52.3ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 15.5ε (Mean = 3.33ε) And other failures.) (*Rmath 3.2.3:* Max = 6.74ε (Mean = 1.3ε))	Max = 260ε (Mean = 34ε)	Max = 9.24ε (Mean = 1.17ε)
Bessel J: Random Data (Tricky large values)	Max = 785ε (Mean = 94.2ε) (*<cmath>:* Max = 5.01e+17ε (Mean = 6.23e+16ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.48e+05ε (Mean = 5.11e+04ε)) (*Rmath 3.2.3:* Max = 71.6ε (Mean = 11.7ε))	Max = 785ε (Mean = 97.4ε)	Max = 59.2ε (Mean = 8.67ε)

Table 65. Error rates for cyl_neumann (integer orders)

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Y0: Mathworld Data (Integer Version)	Max = 5.53ε (Mean = 2.4ε) (*<cmath>:* Max = 2.05e+05ε (Mean = 6.87e+04ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.46ε (Mean = 2.38ε)) (*Rmath 3.2.3:* Max = 167ε (Mean = 56.5ε))	Max = 5.53ε (Mean = 2.4ε)	Max = 4.61ε (Mean = 2.29ε) (*<math.h>:* Max = 5.37e+03ε (Mean = 1.81e+03ε))
Y1: Mathworld Data (Integer Version)	Max = 6.33ε (Mean = 2.25ε) (*<cmath>:* Max = 9.71e+03ε (Mean = 4.08e+03ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.51ε (Mean = 0.839ε)) (*Rmath 3.2.3:* Max = 193ε (Mean = 64.4ε))	Max = 6.33ε (Mean = 2.29ε)	Max = 4.75ε (Mean = 1.72ε) (*<math.h>:* Max = 1.86e+04ε (Mean = 6.2e+03ε))
Yn: Mathworld Data (Integer Version)	Max = 55.2ε (Mean = 17.8ε) (*<cmath>:* Max = 2.2e+20ε (Mean = 6.97e+19ε) And other failures.)	Max = 0.993ε (Mean = 0.314ε) (*GSL 2.1:* Max = 2.41e+05ε (Mean = 7.62e+04ε)) (*Rmath 3.2.3:* Max = 1.24e+04ε (Mean = 4e+03ε))	Max = 55.2ε (Mean = 17.8ε)	Max = 35ε (Mean = 11.9ε) (*<math.h>:* Max = 2.49e+05ε (Mean = 8.14e+04ε))

Table 66. Error rates for cyl_neumann

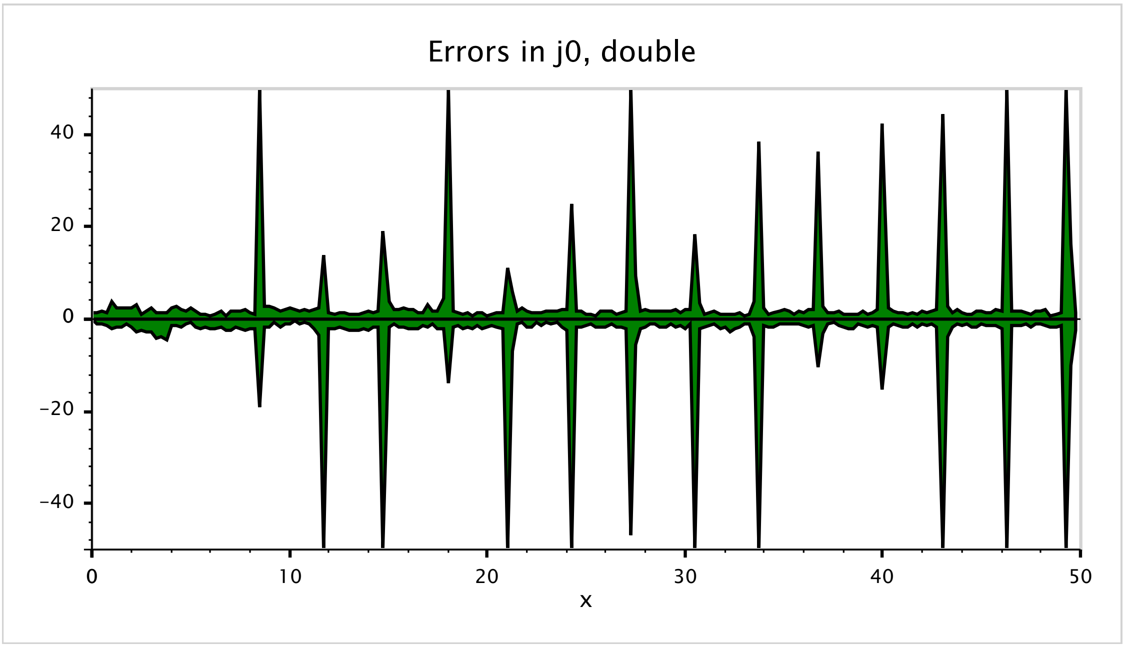
	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Y0: Mathworld Data	Max = 5.53ε (Mean = 2.4ε) (*<cmath>:* Max = 2.05e+05ε (Mean = 6.87e+04ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 60.9ε (Mean = 20.4ε)) (*Rmath 3.2.3:* Max = 167ε (Mean = 56.5ε))	Max = 5.53ε (Mean = 2.4ε)	Max = 4.61ε (Mean = 2.29ε)
Y1: Mathworld Data	Max = 6.33ε (Mean = 2.25ε) (*<cmath>:* Max = 9.71e+03ε (Mean = 4.08e+03ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 23.4ε (Mean = 8.1ε)) (*Rmath 3.2.3:* Max = 193ε (Mean = 64.4ε))	Max = 6.33ε (Mean = 2.29ε)	Max = 4.75ε (Mean = 1.72ε)
Yn: Mathworld Data	Max = 55.2ε (Mean = 17.8ε) (*<cmath>:* Max = 2.2e+20ε (Mean = 6.97e+19ε) And other failures.)	Max = 0.993ε (Mean = 0.314ε) (*GSL 2.1:* Max = 2.41e+05ε (Mean = 7.62e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 1.24e+04ε (Mean = 4e+03ε))	Max = 55.2ε (Mean = 17.8ε)	Max = 35ε (Mean = 11.9ε)
Yv: Mathworld Data	Max = 10.7ε (Mean = 4.93ε) (*<cmath>:* Max = 3.49e+15ε (Mean = 1.05e+15ε) And other failures.)	Max = 10ε (Mean = 3.02ε) (*GSL 2.1:* Max = 1.07e+05ε (Mean = 3.22e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 243ε (Mean = 73.9ε))	Max = 10.7ε (Mean = 5.1ε)	Max = 7.89ε (Mean = 3.27ε)
Yv: Mathworld Data (large values)	Max = 1.7ε (Mean = 1.33ε) (*<cmath>:* Max = 43.2ε (Mean = 16.3ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 60.8ε (Mean = 23ε) And other failures.) (*Rmath 3.2.3:* Max = 0.682ε (Mean = 0.335ε))	Max = 1.7ε (Mean = 1.33ε)	Max = 0.682ε (Mean = 0.423ε)
Y0 and Y1: Random Data	Max = 10.8ε (Mean = 3.04ε) (*<cmath>:* Max = 2.59e+03ε (Mean = 500ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 34.4ε (Mean = 8.9ε)) (*Rmath 3.2.3:* Max = 83ε (Mean = 14.2ε))	Max = 10.8ε (Mean = 3.04ε)	Max = 4.17ε (Mean = 1.24ε)
Yn: Random Data	Max = 338ε (Mean = 27.5ε) (*<cmath>:* Max = 4.01e+03ε (Mean = 348ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 500ε (Mean = 47.8ε)) (*Rmath 3.2.3:* Max = 691ε (Mean = 67.9ε))	Max = 338ε (Mean = 27.5ε)	Max = 117ε (Mean = 10.2ε)

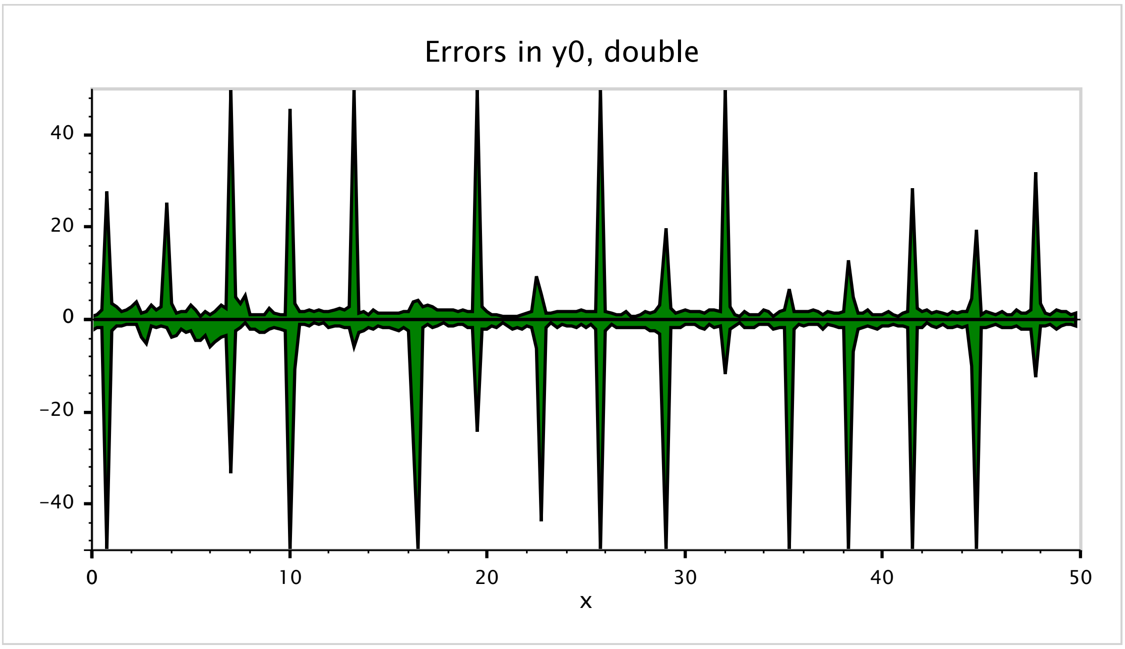
	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Yv: Random Data	Max = 2.08e+03ε (Mean = 149ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 1.53ε (Mean = 0.102ε) (*GSL 2.1:* Max = 1.41e+06ε (Mean = 7.67e+04ε)) (*Rmath 3.2.3:* Max = 1.79e+05ε (Mean = 9.64e+03ε))	Max = 2.08e+03ε (Mean = 149ε)	Max = 1.23e+03ε (Mean = 69.9ε)

Note that for large *x* these functions are largely dependent on the accuracy of the `std::sin` and `std::cos` functions.

Comparison to GSL and Cephes is interesting: both Cephes and this library optimise the integer order case - leading to identical results - simply using the general case is for the most part slightly more accurate though, as noted by the better accuracy of GSL in the integer argument cases. This implementation tends to perform much better when the arguments become large, Cephes in particular produces some remarkably inaccurate results with some of the test data (no significant figures correct), and even GSL performs badly with some inputs to J_v. Note that by way of double-checking these results, the worst performing Cephes and GSL cases were recomputed using functions.wolfram.com, and the result checked against our test data: no errors in the test data were found.

The following error plot are based on an exhaustive search of the functions domain for J0 and Y0, MSVC-15.5 at `double` precision, other compilers and precisions are very similar - the plots simply illustrate the relatively large errors as you approach a zero, and the very low errors elsewhere.





Implementation

The implementation is mostly about filtering off various special cases:

When *x* is negative, then the order *v* must be an integer or the result is a domain error. If the order is an integer then the function is odd for odd orders and even for even orders, so we reflect to $x > 0$.

When the order *v* is negative then the reflection formulae can be used to move to $v > 0$:

$$J_{-v}(z) = \cos(v\pi)J_v(z) - \sin(v\pi)Y_v(z)$$

$$Y_{-v}(z) = \sin(v\pi)J_v(z) + \cos(v\pi)Y_v(z)$$

Note that if the order is an integer, then these formulae reduce to:

$J_{-n} = (-1)^n J_n$

$Y_{-n} = (-1)^n Y_n$

However, in general, a negative order implies that we will need to compute both J and Y.

When *x* is large compared to the order *v* then the asymptotic expansions for large *x* in M. Abramowitz and I.A. Stegun, *Handbook of Mathematical Functions* 9.2.19 are used (these were found to be more reliable than those in A&S 9.2.5).

When the order *v* is an integer the method first relates the result to J_0, J_1, Y_0 and Y_1 using either forwards or backwards recurrence (Miller's algorithm) depending upon which is stable. The values for J_0, J_1, Y_0 and Y_1 are calculated using the rational minimax approximations on root-bracketing intervals for small $|x|$ and Hankel asymptotic expansion for large $|x|$. The coefficients are from:

W.J. Cody, *ALGORITHM 715: SPECFUN - A Portable FORTRAN Package of Special Function Routines and Test Drivers*, ACM Transactions on Mathematical Software, vol 19, 22 (1993).

and

J.F. Hart et al, *Computer Approximations*, John Wiley & Sons, New York, 1968.

These approximations are accurate to around 19 decimal digits: therefore these methods are not used when type T has more than 64 binary digits.

When *x* is smaller than machine epsilon then the following approximations for $Y_0(x)$, $Y_1(x)$, $Y_2(x)$ and $Y_n(x)$ can be used (see: http://functions.wolfram.com/03.03.06.0037.01, http://functions.wolfram.com/03.03.06.0038.01, http://functions.wolfram.com/03.03.06.0039.01 and http://functions.wolfram.com/03.03.06.0040.01):

$$Y_0(z) \quad = \quad \frac{2}{\pi}\left(\log\left(\frac{z}{2}\right)+\gamma\right) \quad ; \quad z<\varepsilon$$

$$Y_1(z) \quad = \quad \frac{z}{\pi}\log\left(\frac{z}{2}\right)-\frac{2}{\pi z}-\frac{z}{2\pi}(1-2\gamma) \quad ; \quad z<\varepsilon$$

$$Y_2 \quad = \quad -\frac{z^2}{8\pi}\left(\frac{3}{2}-2\gamma\right)-\frac{4}{\pi z^2}+\frac{z^2}{4\pi}\log\left(\frac{z}{2}\right) \quad ; \quad z<\varepsilon$$

$$Y_n(z) \quad = \quad -(n-1)\frac{!}{\pi}\left(\frac{z}{2}\right)^{-n} \quad ; z<\varepsilon \quad \wedge \quad n>2$$

When *x* is small compared to *v* and *v* is not an integer, then the following series approximation can be used for $Y_v(x)$, this is also an area where other approximations are often too slow to converge to be used (see http://functions.wolfram.com/03.03.06.0034.01):

$$Y_v(z) \quad = \quad -\frac{\Gamma(v)}{\pi}\sum_{k=0}^{\infty}\frac{(-1)^k\left(\frac{z}{2}\right)^{2k-v}}{(1-v)_k k!} \quad - \quad \frac{\Gamma(-v)\cos(v\pi)}{\pi}\sum_{k=0}^{\infty}\frac{(-1)^k\left(\frac{z}{2}\right)^{2k+v}}{(v+1)_k k!} \quad ; \quad v \notin \mathbb{Z}$$

When *x* is small compared to *v*, $J_v x$ is best computed directly from the series:

$$J_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty}\frac{\left(-\frac{1}{4}z^2\right)^k}{k!\,\Gamma(v+k+1)}$$

In the general case we compute J_v and Y_v simultaneously.

To get the initial values, let $\mu = \nu - \text{floor}(\nu + 1/2)$, then μ is the fractional part of ν such that $|\mu| <= 1/2$ (we need this for convergence later). The idea is to calculate $J_\mu(x)$, $J_{\mu+1}(x)$, $Y_\mu(x)$, $Y_{\mu+1}(x)$ and use them to obtain $J_\nu(x)$, $Y_\nu(x)$.

The algorithm is called Steed's method, which needs two continued fractions as well as the Wronskian:

$$W = J_v(z)Y_v'(z) - Y_v(z)J_v'(z) = Y_v(z)J_{v+1}(z) - J_v(z)Y_{v+1}(z) = \frac{2}{\pi z}$$

$$\text{CF1} \ : f_v = \frac{J_{v+1}}{J_v} = \frac{1}{\frac{2(v+1)}{x} -}\,\frac{1}{\frac{2(v+2)}{x} -}\ \cdot\ \cdot\ \cdot$$

$$\text{CF2} \ : p+iq = \frac{J_v'+iY_v'}{J_v+iY_v} = \left(i-\frac{1}{2x}\right)+\frac{i}{x}\left[\frac{\left(\frac{1}{2}\right)^2-v^2}{2(x+i)+}\,\frac{\left(\frac{3}{2}\right)^2-v^2}{2(x+2i)+}\ \cdot\ \cdot\ \cdot\right]$$

See: F.S. Acton, *Numerical Methods that Work*, The Mathematical Association of America, Washington, 1997.

The continued fractions are computed using the modified Lentz's method (W.J. Lentz, *Generating Bessel functions in Mie scattering calculations using continued fractions*, Applied Optics, vol 15, 668 (1976)). Their convergence rates depend on *x*, therefore we need different strategies for large *x* and small *x*.

x > *v*, CF1 needs O(*x*) iterations to converge, CF2 converges rapidly

x <= *v*, CF1 converges rapidly, CF2 fails to converge when *x* -> 0

When *x* is large (*x* > 2), both continued fractions converge (CF1 may be slow for really large *x*). J_μ, $J_{\mu+1}$, Y_μ, $Y_{\mu+1}$ can be calculated by

$$\begin{aligned} J_\mu &= \pm\left(\frac{W}{q+\gamma(p-f_\mu)}\right)^{\frac{1}{2}} \\ J_{\mu+1} &= J_\mu\left(\frac{\mu}{x}-f_\mu\right) \\ Y_\mu &= \gamma J_\mu \\ Y_{\mu+1} &= Y_\mu\left(\frac{\mu}{x}-p-\frac{q}{\gamma}\right) \end{aligned}$$

where

$$\gamma = \frac{p-f_\mu}{q}$$

J_ν and Y_μ are then calculated using backward (Miller's algorithm) and forward recurrence respectively.

When *x* is small (*x* <= 2), CF2 convergence may fail (but CF1 works very well). The solution here is Temme's series:

$$\begin{aligned} Y_\mu &= -\sum_{k=0}^{\infty} c_k g_k \\ Y_{\mu+1} &= -\frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \end{aligned}$$

where

$$c_k = \frac{1}{k!}\left(-\frac{x^2}{4}\right)^k$$

g_k and h_k are also computed by recursions (involving gamma functions), but the formulas are a little complicated, readers are refered to N.M. Temme, *On the numerical evaluation of the ordinary Bessel function of the second kind*, Journal of Computational Physics, vol 21, 343 (1976). Note Temme's series converge only for |μ| <= 1/2.

As the previous case, Y_ν is calculated from the forward recurrence, so is $Y_{\nu+1}$. With these two values and f_ν, the Wronskian yields $J_\nu(x)$ directly without backward recurrence.

Finding Zeros of Bessel Functions of the First and Second Kinds

Synopsis

```
#include <boost/math/special_functions/bessel.hpp>
```

Functions for obtaining both a single zero or root of the Bessel function, and placing multiple zeros into a container like `std::vector` by providing an output iterator.

The signature of the single value functions are:

```
template <class T>
T cyl_bessel_j_zero(
         T v,             // Floating-point value for Jv.
         int m);          // 1-based index of zero.

template <class T>
T cyl_neumann_zero(
         T v,             // Floating-point value for Jv.
         int m);          // 1-based index of zero.
```

and for multiple zeros:

```
template <class T, class OutputIterator>
OutputIterator cyl_bessel_j_zero(
                     T v,                       // Floating-point value for Jv.
                     int start_index,           // 1-based index of first zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it);    // Destination for zeros.

template <class T, class OutputIterator>
OutputIterator cyl_neumann_zero(
                     T v,                       // Floating-point value for Jv.
                     int start_index,           // 1-based index of zero.
                     unsigned number_of_zeros,  // How many zeros to generate
                     OutputIterator out_it);    // Destination for zeros.
```

There are also versions which allow control of the Policies for error handling and precision.

```
 template <class T>
 T cyl_bessel_j_zero(
          T v,             // Floating-point value for Jv.
          int m,           // 1-based index of zero.
          const Policy&); // Policy to use.

 template <class T>
 T cyl_neumann_zero(
          T v,             // Floating-point value for Jv.
          int m,           // 1-based index of zero.
          const Policy&); // Policy to use.


template <class T, class OutputIterator>
OutputIterator cyl_bessel_j_zero(
                     T v,                       // Floating-point value for Jv.
                     int start_index,           // 1-based index of first zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it,     // Destination for zeros.
                     const Policy& pol);        // Policy to use.

template <class T, class OutputIterator>
OutputIterator cyl_neumann_zero(
                     T v,                       // Floating-point value for Jv.
                     int start_index,           // 1-based index of zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it,     // Destination for zeros.
                     const Policy& pol);        // Policy to use.
```

Description

Every real order ν cylindrical Bessel and Neumann functions have an infinite number of zeros on the positive real axis. The real zeros on the positive real axis can be found by solving for the roots of

$$J_\nu(j_{\nu, m}) = 0$$

$$Y_\nu(y_{\nu, m}) = 0$$

Here, $j_{\nu, m}$ represents the m^{th} root of the cylindrical Bessel function of order ν, and $y_{\nu, m}$ represents the m^{th} root of the cylindrical Neumann function of order ν.

The zeros or roots (values of `x` where the function crosses the horizontal `y = 0` axis) of the Bessel and Neumann functions are computed by two functions, `cyl_bessel_j_zero` and `cyl_neumann_zero`.

In each case the index or rank of the zero returned is 1-based, which is to say:

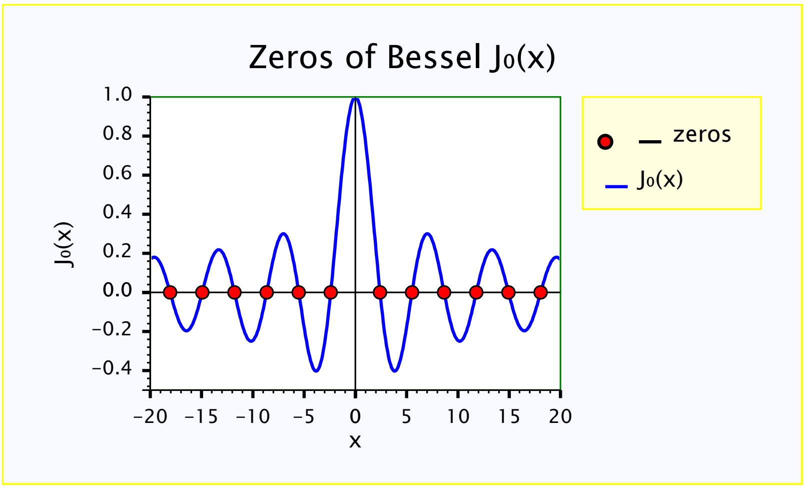
```
cyl_bessel_j_zero(v, 1);
```

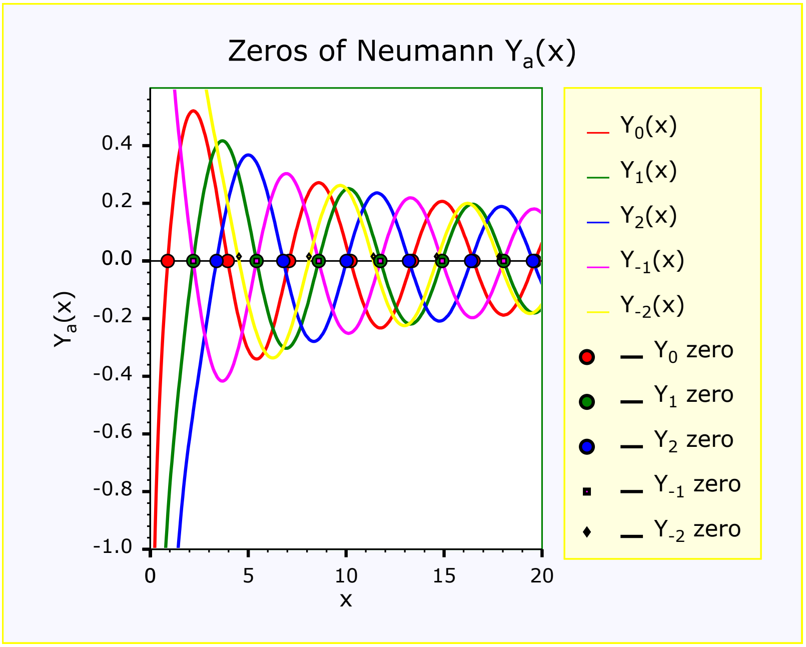
returns the first zero of Bessel J.

Passing an `start_index <= 0` results in a `std::domain_error` being raised.

For certain parameters, however, the zero'th root is defined and it has a value of zero. For example, the zero'th root of `J[sub v](x)` is defined and it has a value of zero for all values of `v > 0` and for negative integer values of `v = -n`. Similar cases are described in the implementation details below.

The order `v` of `J` can be positive, negative and zero for the `cyl_bessel_j` and `cyl_neumann` functions, but not infinite nor NaN.





Examples of finding Bessel and Neumann zeros

This example demonstrates calculating zeros of the Bessel and Neumann functions. It also shows how Boost.Math and Boost.Multiprecision can be combined to provide a many decimal digit precision. For 50 decimal digit precision we need to include

```
#include <boost/multiprecision/cpp_dec_float.hpp>
```

and a `typedef` for `float_type` may be convenient (allowing a quick switch to re-compute at built-in `double` or other precision)

```
typedef boost::multiprecision::cpp_dec_float_50 float_type;
```

To use the functions for finding zeros of the functions we need

```
#include <boost/math/special_functions/bessel.hpp>
```

This file includes the forward declaration signatures for the zero-finding functions:

```
//  #include <boost/math/special_functions/math_fwd.hpp>
```

but more details are in the full documentation, for example at Boost.Math Bessel functions.

This example shows obtaining both a single zero of the Bessel function, and then placing multiple zeros into a container like `std::vector` by providing an iterator.

Tip

It is always wise to place code using Boost.Math inside try'n'catch blocks; this will ensure that helpful error messages are shown when exceptional conditions arise.

First, evaluate a single Bessel zero.

The precision is controlled by the float-point type of template parameter `T` of `v` so this example has `double` precision, at least 15 but up to 17 decimal digits (for the common 64-bit double).

```
//    double root = boost::math::cyl_bessel_j_zero(0.0, 1);
//    // Displaying with default precision of 6 decimal digits:
//    std::cout << "boost::math::cyl_bessel_j_zero(0.0, 1) " << root << std::endl; // 2.40483
//    // And with all the guaranteed (15) digits:
//    std::cout.precision(std::numeric_limits<double>::digits10);
//    std::cout << "boost::math::cyl_bessel_j_zero(0.0, 1) " << root << std::endl; // ↵
2.40482555769577
```

But note that because the parameter `v` controls the precision of the result, `v` **must be a floating-point type**. So if you provide an integer type, say 0, rather than 0.0, then it will fail to compile thus:

```
root = boost::math::cyl_bessel_j_zero(0, 1);
```

with this error message

```
error C2338: Order must be a floating-point type.
```

Optionally, we can use a policy to ignore errors, C-style, returning some value, perhaps infinity or NaN, or the best that can be done. (See user error handling).

To create a (possibly unwise!) policy `ignore_all_policy` that ignores all errors:

```
typedef boost::math::policies::policy<
  boost::math::policies::domain_error<boost::math::policies::ignore_error>,
  boost::math::policies::overflow_error<boost::math::policies::ignore_error>,
  boost::math::policies::underflow_error<boost::math::policies::ignore_error>,
  boost::math::policies::denorm_error<boost::math::policies::ignore_error>,
  boost::math::policies::pole_error<boost::math::policies::ignore_error>,
  boost::math::policies::evaluation_error<boost::math::policies::ignore_error>
            > ignore_all_policy;
```

Examples of use of this `ignore_all_policy` are

```
double inf = std::numeric_limits<double>::infinity();
double nan = std::numeric_limits<double>::quiet_NaN();

double dodgy_root = boost::math::cyl_bessel_j_zero(-1.0, 1, ignore_all_policy());
std::cout << "boost::math::cyl_bessel_j_zero(-1.0, 1) " << dodgy_root << std::endl; // 1.#QNAN
double inf_root = boost::math::cyl_bessel_j_zero(inf, 1, ignore_all_policy());
std::cout << "boost::math::cyl_bessel_j_zero(inf, 1) " << inf_root << std::endl; // 1.#QNAN
double nan_root = boost::math::cyl_bessel_j_zero(nan, 1, ignore_all_policy());
std::cout << "boost::math::cyl_bessel_j_zero(nan, 1) " << nan_root << std::endl; // 1.#QNAN
```

Another version of `cyl_bessel_j_zero` allows calculation of multiple zeros with one call, placing the results in a container, often `std::vector`. For example, generate and display the first five `double` roots of J_v for integral order 2, as column $J_2(x)$ in table 1 of Wolfram Bessel Function Zeros.

```
unsigned int n_roots = 5U;
std::vector<double> roots;
boost::math::cyl_bessel_j_zero(2.0, 1, n_roots, std::back_inserter(roots));
std::copy(roots.begin(),
          roots.end(),
          std::ostream_iterator<double>(std::cout, "\n"));
```

Or we can use Boost.Multiprecision to generate 50 decimal digit roots of J_v for non-integral order `v= 71/19 == 3.736842`, expressed as an exact-integer fraction to generate the most accurate value possible for all floating-point types.

We set the precision of the output stream, and show trailing zeros to display a fixed 50 decimal digits.

```
std::cout.precision(std::numeric_limits<float_type>::digits10); // 50 decimal digits.
std::cout << std::showpoint << std::endl; // Show trailing zeros.

float_type x = float_type(71) / 19;
float_type r = boost::math::cyl_bessel_j_zero(x, 1); // 1st root.
std::cout << "x = " << x << ", r = " << r << std::endl;

r = boost::math::cyl_bessel_j_zero(x, 20U); // 20th root.
std::cout << "x = " << x << ", r = " << r << std::endl;

std::vector<float_type> zeros;
boost::math::cyl_bessel_j_zero(x, 1, 3, std::back_inserter(zeros));

std::cout << "cyl_bessel_j_zeros" << std::endl;
// Print the roots to the output stream.
std::copy(zeros.begin(), zeros.end(),
          std::ostream_iterator<float_type>(std::cout, "\n"));
```

Using Output Iterator to sum zeros of Bessel Functions

This example demonstrates summing zeros of the Bessel functions. To use the functions for finding zeros of the functions we need

```
#include <boost/math/special_functions/bessel.hpp>
```

We use the `cyl_bessel_j_zero` output iterator parameter `out_it` to create a sum of *1/zeros*^{2} by defining a custom output iterator:

```
template <class T>
struct output_summation_iterator
{
   output_summation_iterator(T* p) : p_sum(p)
   {}
   output_summation_iterator& operator*()
   { return *this; }
    output_summation_iterator& operator++()
   { return *this; }
   output_summation_iterator& operator++(int)
   { return *this; }
   output_summation_iterator& operator = (T const& val)
   {
     *p_sum += 1./ (val * val); // Summing 1/zero^2.
     return *this;
   }
private:
   T* p_sum;
};
```

The sum is calculated for many values, converging on the analytical exact value of `1/8`.

```
using boost::math::cyl_bessel_j_zero;
double nu = 1.;
double sum = 0;
output_summation_iterator<double> it(&sum);  // sum of 1/zeros^2
cyl_bessel_j_zero(nu, 1, 10000, it);

double s = 1/(4 * (nu + 1)); // 0.125 = 1/8 is exact analytical solution.
std::cout << std::setprecision(6) << "nu = " << nu << ", sum = " << sum
  << ", exact = " << s << std::endl;
// nu = 1.00000, sum = 0.124990, exact = 0.125000
```

Calculating zeros of the Neumann function.

This example also shows how Boost.Math and Boost.Multiprecision can be combined to provide a many decimal digit precision. For 50 decimal digit precision we need to include

```
#include <boost/multiprecision/cpp_dec_float.hpp>
```

and a `typedef` for `float_type` may be convenient (allowing a quick switch to re-compute at built-in `double` or other precision)

```
typedef boost::multiprecision::cpp_dec_float_50 float_type;
```

To use the functions for finding zeros of the `cyl_neumann` function we need:

```
#include <boost/math/special_functions/bessel.hpp>
```

The Neumann (Bessel Y) function zeros are evaluated very similarly:

```
using boost::math::cyl_neumann_zero;
double zn = cyl_neumann_zero(2., 1);
std::cout << "cyl_neumann_zero(2., 1) = " << zn << std::endl;

std::vector<float> nzeros(3); // Space for 3 zeros.
cyl_neumann_zero<float>(2.F, 1, nzeros.size(), nzeros.begin());

std::cout << "cyl_neumann_zero<float>(2.F, 1, ";
// Print the zeros to the output stream.
std::copy(nzeros.begin(), nzeros.end(),
          std::ostream_iterator<float>(std::cout, ", "));

std::cout << "\n""cyl_neumann_zero(static_cast<float_type>(220)/100, 1) = "
  << cyl_neumann_zero(static_cast<float_type>(220)/100, 1) << std::endl;
// 3.6154383428745996706772556069431792744372398748422
```

Error messages from 'bad' input

Another example demonstrates calculating zeros of the Bessel functions showing the error messages from 'bad' input is handled by throwing exceptions.

To use the functions for finding zeros of the functions we need:

```
#include <boost/math/special_functions/bessel.hpp>
#include <boost/math/special_functions/airy.hpp>
```

Tip

It is always wise to place all code using Boost.Math inside try'n'catch blocks; this will ensure that helpful error messages can be shown when exceptional conditions arise.

Examples below show messages from several 'bad' arguments that throw a `domain_error` exception.

```
try
{ // Try a zero order v.
  float dodgy_root = boost::math::cyl_bessel_j_zero(0.F, 0);
  std::cout << "boost::math::cyl_bessel_j_zero(0.F, 0) " << dodgy_root << std::endl;
  // Thrown exception Error in function boost::math::cyl_bessel_j_zero<double>(double, int):
  // Requested the 0'th zero of J0, but the rank must be > 0 !
}
catch (std::exception& ex)
{
  std::cout << "Thrown exception " << ex.what() << std::endl;
}
```

Note

The type shown in the error message is the type **after promotion**, using precision policy and internal promotion policy, from `float` to `double` in this case.

In this example the promotion goes:

1. Arguments are `float` and `int`.
2. Treat `int` "as if" it were a `double`, so arguments are `float` and `double`.

3. Common type is `double` - so that's the precision we want (and the type that will be returned).

4. Evaluate internally as `double` for full `float` precision.

See full code for other examples that promote from `double` to `long double`.

Other examples of 'bad' inputs like infinity and NaN are below. Some compiler warnings indicate that 'bad' values are detected at compile time.

```
try
{ // order v = inf
   std::cout << "boost::math::cyl_bessel_j_zero(inf, 1) " << std::endl;
   double inf = std::numeric_limits<double>::infinity();
   double inf_root = boost::math::cyl_bessel_j_zero(inf, 1);
   std::cout << "boost::math::cyl_bessel_j_zero(inf, 1) " << inf_root << std::endl;
   // Throw exception Error in function boost::math::cyl_bessel_j_zero<long double>(long ↵
double, unsigned):
   // Order argument is 1.#INF, but must be finite >= 0 !
}
catch (std::exception& ex)
{
  std::cout << "Thrown exception " << ex.what() << std::endl;
}

try
{ // order v = NaN, rank m = 1
   std::cout << "boost::math::cyl_bessel_j_zero(nan, 1) " << std::endl;
   double nan = std::numeric_limits<double>::quiet_NaN();
   double nan_root = boost::math::cyl_bessel_j_zero(nan, 1);
   std::cout << "boost::math::cyl_bessel_j_zero(nan, 1) " << nan_root << std::endl;
   // Throw exception Error in function boost::math::cyl_bessel_j_zero<long double>(long ↵
double, unsigned):
   // Order argument is 1.#QNAN, but must be finite >= 0 !
}
catch (std::exception& ex)
{
  std::cout << "Thrown exception " << ex.what() << std::endl;
}
```

The output from other examples are shown appended to the full code listing.

The full code (and output) for these examples is at Bessel zeros, Bessel zeros iterator, Neumann zeros, Bessel error messages.

Implementation

Various methods are used to compute initial estimates for $j_{\nu, m}$ and $y_{\nu, m}$; these are described in detail below.

After finding the initial estimate of a given root, its precision is subsequently refined to the desired level using Newton-Raphson iteration from Boost.Math's root-finding with derivatives utilities combined with the functions cyl_bessel_j and cyl_neumann.

Newton iteration requires both $J_\nu(x)$ or $Y_\nu(x)$ as well as its derivative. The derivatives of $J_\nu(x)$ and $Y_\nu(x)$ with respect to x are given by M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions, NBS (1964). In particular,

$$d/_{dx}\, J_\nu(x) = J_{\nu-1}(x) - \nu J_\nu(x) / x$$

$$d/_{dx}\, Y_\nu(x) = Y_{\nu-1}(x) - \nu Y_\nu(x) / x$$

Enumeration of the rank of a root (in other words the index of a root) begins with one and counts up, in other words $m,=1,2,3,\ldots$ The value of the first root is always greater than zero.

For certain special parameters, cylindrical Bessel functions and cylindrical Neumann functions have a root at the origin. For example, $J_{\nu}(x)$ has a root at the origin for every positive order $\nu > 0$, and for every negative integer order $\nu = -n$ with $n \in {}^{+}$ and $n \neq 0$.

In addition, $Y_{\nu}(x)$ has a root at the origin for every negative half-integer order $\nu = -n/2$, with $n \in {}^{+}$ and and $n \neq 0$.

For these special parameter values, the origin with a value of $x = 0$ is provided as the 0^{th} root generated by `cyl_bessel_j_zero()` and `cyl_neumann_zero()`.

When calculating initial estimates for the roots of Bessel functions, a distinction is made between positive order and negative order, and different methods are used for these. In addition, different algorithms are used for the first root $m = 1$ and for subsequent roots with higher rank $m \geq 2$. Furthermore, estimates of the roots for Bessel functions with order above and below a cutoff at $\nu = 2.2$ are calculated with different methods.

Calculations of the estimates of $j_{\nu,1}$ and $y_{\nu,1}$ with $0 \leq \nu < 2.2$ use empirically tabulated values. The coefficients for these have been generated by a computer algebra system.

Calculations of the estimates of $j_{\nu,1}$ and $y_{\nu,1}$ with $\nu \geq 2.2$ use Eqs.9.5.14 and 9.5.15 in M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions, NBS (1964).

In particular,

$$j_{\nu,1} \cong \nu + 1.85575\ \nu\ + 1.033150\ \nu^{-}\ - 0.00397\ \nu^{-1} - 0.0908\ \nu^{-5/3} + 0.043\ \nu^{-7/3} + \ldots$$

and

$$y_{\nu,1} \cong \nu + 0.93157\ \nu\ + 0.26035\ \nu^{-}\ + 0.01198\ \nu^{-1} - 0.0060\ \nu^{-5/3} - 0.001\ \nu^{-7/3} + \ldots$$

Calculations of the estimates of $j_{\nu, m}$ and $y_{\nu, m}$ with rank $m > 2$ and $0 \leq \nu < 2.2$ use McMahon's approximation, as described in M. Abramowitz and I. A. Stegan, Section 9.5 and 9.5.12. In particular,

$$j_{\nu,m}, y_{\nu,m} \cong \beta - (\mu-1) / 8\beta$$

$$- 4(\mu-1)(7\mu - 31) / 3(8\beta)^{3}$$

$$-32(\mu-1)(83\mu^2 - 982\mu + 3779) / 15(8\beta)^{5}$$

$$-64(\mu-1)(6949\mu^{3} - 153855\mu^2 + 1585743\mu - 6277237) / 105(8a)^{7}$$

$$- \ldots \qquad (5)$$

where $\mu = 4\nu^{2}$ and $\beta = (m + ½\nu - ¼)\pi$ for $j_{\nu,m}$ and $\beta = (m + ½\nu - ¾)\pi$ for $y_{\nu,m}$.

Calculations of the estimates of $j_{\nu, m}$ and $y_{\nu, m}$ with $\nu \geq 2.2$ use one term in the asymptotic expansion given in Eq.9.5.22 and top line of Eq.9.5.26 combined with Eq. 9.3.39, all in M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions, NBS (1964) explicit and easy-to-understand treatment for asymptotic expansion of zeros. The latter two equations are expressed for argument (x) greater than one. (Olver also gives the series form of the equations in §10.21(vi) McMahon's Asymptotic Expansions for Large Zeros - using slightly different variable names).

In summary,

$$j_{\nu, m} \sim \nu x(-\zeta) + f_{1}(-\zeta/\nu)$$

where $-\zeta = \nu^{-2/3} a_{m}$ and a_{m} is the absolute value of the m^{th} root of $Ai(x)$ on the negative real axis.

Here $x = x(-\zeta)$ is the inverse of the function

$$(-\zeta)^{3/2} = \sqrt{(x^2 - 1)} - \cos {}^{1}(1/x) \qquad (7)$$

Furthermore,

$$f_{1}(-\zeta) = ½x(-\zeta)\ \{h(-\zeta)\}^2 \cdot b_{0}(-\zeta)$$

where

$h(-\zeta) = \{4(-\zeta) / (x^2 - 1)\}^4$

and

$b_0(-\zeta) = -5/(48\zeta^2) + 1/(-\zeta)^{½} \cdot \{ 5/(24(x^2-1)^{3/2}) + 1/(8(x^2-1)^{½)}\}$

When solving for $x(-\zeta)$ in Eq. 7 above, the right-hand-side is expanded to order 2 in a Taylor series for large x. This results in

$(-\zeta)^{3/2} \approx x + 1/2x - \pi/2$

The positive root of the resulting quadratic equation is used to find an initial estimate $x(-\zeta)$. This initial estimate is subsequently refined with several steps of Newton-Raphson iteration in Eq. 7.

Estimates of the roots of cylindrical Bessel functions of negative order on the positive real axis are found using interlacing relations. For example, the m^{th} root of the cylindrical Bessel function $j_{-v,m}$ is bracketed by the m^{th} root and the $(m+1)^{th}$ root of the Bessel function of corresponding positive integer order. In other words,

$j_{nv,m} < j_{-v,m} < j_{nv,m+1}$

where $m > 1$ and n_v represents the integral floor of the absolute value of $|-v|$.

Similar bracketing relations are used to find estimates of the roots of Neumann functions of negative order, whereby a discontinuity at every negative half-integer order needs to be handled.

Bracketing relations do not hold for the first root of cylindrical Bessel functions and cylindrical Neumann functions with negative order. Therefore, iterative algorithms combined with root-finding via bisection are used to localize $j_{-v,1}$ and $y_{-v,1}$.

Testing

The precision of evaluation of zeros was tested at 50 decimal digits using `cpp_dec_float_50` and found identical with spot values computed by Wolfram Alpha.

Modified Bessel Functions of the First and Second Kinds

Synopsis

`#include <boost/math/special_functions/bessel.hpp>`

```
template <class T1, class T2>
calculated-result-type cyl_bessel_i(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_i(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type cyl_bessel_k(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_k(T1 v, T2 x, const Policy&);
```

Description

The functions cyl_bessel_i and cyl_bessel_k return the result of the modified Bessel functions of the first and second kind respectively:

cyl_bessel_i(v, x) = $I_v(x)$

cyl_bessel_k(v, x) = $K_v(x)$

where:

$$I_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

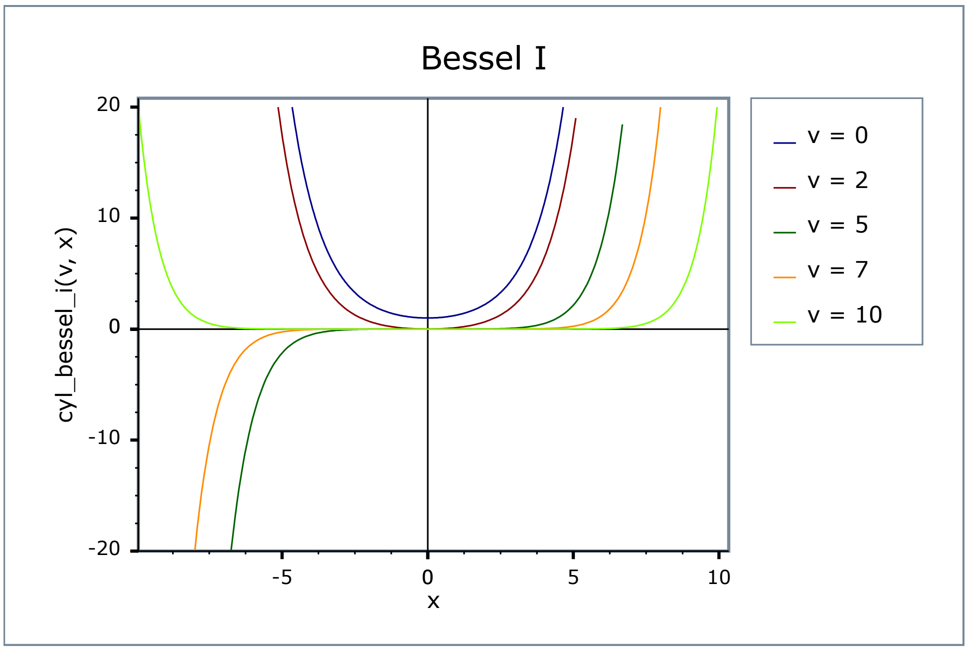
$$K_\nu(z) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types. The functions are also optimised for the relatively common case that T1 is an integer.

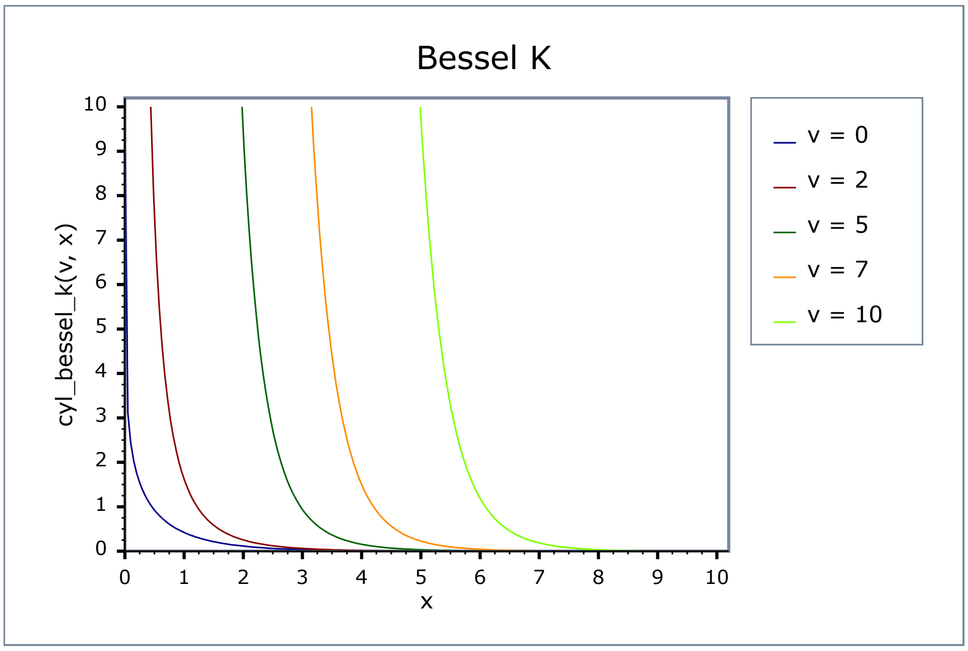
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The functions return the result of domain_error whenever the result is undefined or complex. For cyl_bessel_j this occurs when `x < 0` and v is not an integer, or when `x == 0` and `v != 0`. For cyl_neumann this occurs when `x <= 0`.

The following graph illustrates the exponential behaviour of I_v.



The following graph illustrates the exponential decay of K_v.



Testing

There are two sets of test values: spot values calculated using functions.wolfram.com, and a much larger set of tests computed using a simplified version of this implementation (with all the special case handling removed).

Accuracy

The following tables show how the accuracy of these functions varies on various platforms, along with comparison to other libraries. Note that only results for the widest floating-point type on the system are given, as narrower types have effectively zero error. All values are relative errors in units of epsilon. Note that our test suite includes some fairly extreme inputs which results in most of the worst problem cases in other libraries:

Table 67. Error rates for cyl_bessel_i (integer orders)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel I0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.79ε (Mean = 0.482ε)) (*Rmath 3.2.3:* Max = 1.52ε (Mean = 0.622ε) And other failures.)	Max = 1.95ε (Mean = 0.738ε) (*<cmath>:* Max = 8.49ε (Mean = 3.46ε) And other failures.)	Max = 1.95ε (Mean = 0.661ε)	Max = 0.762ε (Mean = 0.329ε)
Bessel I1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.82ε (Mean = 0.456ε)) (*Rmath 3.2.3:* Max = 1.53ε (Mean = 0.483ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε) (*<cmath>:* Max = 5ε (Mean = 2.15ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε)	Max = 0.767ε (Mean = 0.398ε)
Bessel In: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.15ε (Mean = 2.13ε) And other failures.) (*Rmath 3.2.3:* Max = 1.73ε (Mean = 0.601ε) And other failures.)	Max = 1.8ε (Mean = 1.33ε) (*<cmath>:* Max = 430ε (Mean = 163ε) And other failures.)	Max = 463ε (Mean = 140ε)	Max = 3.46ε (Mean = 1.32ε)

Table 68. Error rates for cyl_bessel_i

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel I0: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 270ε (Mean = 91.6ε) And other failures.) (*Rmath 3.2.3:* Max = 1.52ε (Mean = 0.622ε) And other failures.)	Max = 1.95ε (Mean = 0.738ε) (*<cmath>:* Max = 8.49ε (Mean = 3.46ε) And other failures.)	Max = 1.95ε (Mean = 0.661ε)	Max = 0.762ε (Mean = 0.329ε)
Bessel I1: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 128ε (Mean = 41ε) And other failures.) (*Rmath 3.2.3:* Max = 1.53ε (Mean = 0.483ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε) (*<cmath>:* Max = 5ε (Mean = 2.15ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε)	Max = 0.767ε (Mean = 0.398ε)
Bessel In: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.31ε (Mean = 0.838ε) And other failures.) (*Rmath 3.2.3:* Max = 1.73ε (Mean = 0.601ε) And other failures.)	Max = 1.8ε (Mean = 1.33ε) (*<cmath>:* Max = 430ε (Mean = 163ε) And other failures.)	Max = 463ε (Mean = 140ε)	Max = 3.46ε (Mean = 1.32ε)
Bessel Iv: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.95ε (Mean = 2.08ε) And other failures.) (*Rmath 3.2.3:* Max = 3.53ε (Mean = 1.39ε))	Max = 4.12ε (Mean = 1.85ε) (*<cmath>:* Max = 616ε (Mean = 221ε) And other failures.)	Max = 4.12ε (Mean = 1.95ε)	Max = 2.97ε (Mean = 1.24ε)
Bessel In: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 261ε (Mean = 53.2ε) And other failures.) (*Rmath 3.2.3:* Max = 7.37ε (Mean = 2.4ε))	Max = 4.62ε (Mean = 1.06ε) (*<cmath>:* Max = 645ε (Mean = 132ε))	Max = 176ε (Mean = 39.1ε)	Max = 9.67ε (Mean = 1.88ε)
Bessel Iv: Random Data	Max = 0.661ε (Mean = 0.0441ε) (*GSL 2.1:* Max = 6.18e+03ε (Mean = 1.55e+03ε) And other failures.) (*Rmath 3.2.3:* Max = 4.28e+08ε (Mean = 2.85e+07ε))	Max = 8.35ε (Mean = 1.62ε) (*<cmath>:* Max = 1.05e+03ε (Mean = 224ε) And other failures.)	Max = 283ε (Mean = 88.4ε)	Max = 7.46ε (Mean = 1.71ε)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel Iv: Mathworld Data (large values)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 37ε (Mean = 18ε) And other failures.) (*Rmath 3.2.3:* Max = 3.77e+168ε (Mean = 2.39e+168ε) And other failures.)	Max = 14.7ε (Mean = 6.66ε) (*<cmath>:* Max = 118ε (Mean = 57.2ε) And other failures.)	Max = 14.7ε (Mean = 6.59ε)	Max = 3.67ε (Mean = 1.64ε)

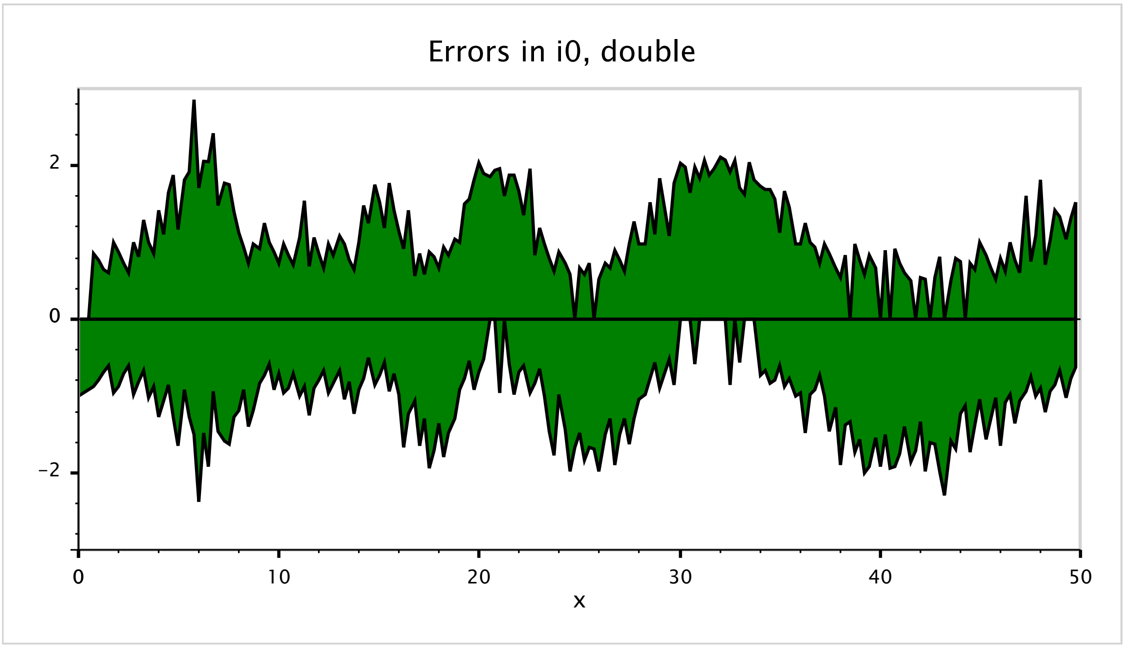
Table 69. Error rates for cyl_bessel_k (integer orders)

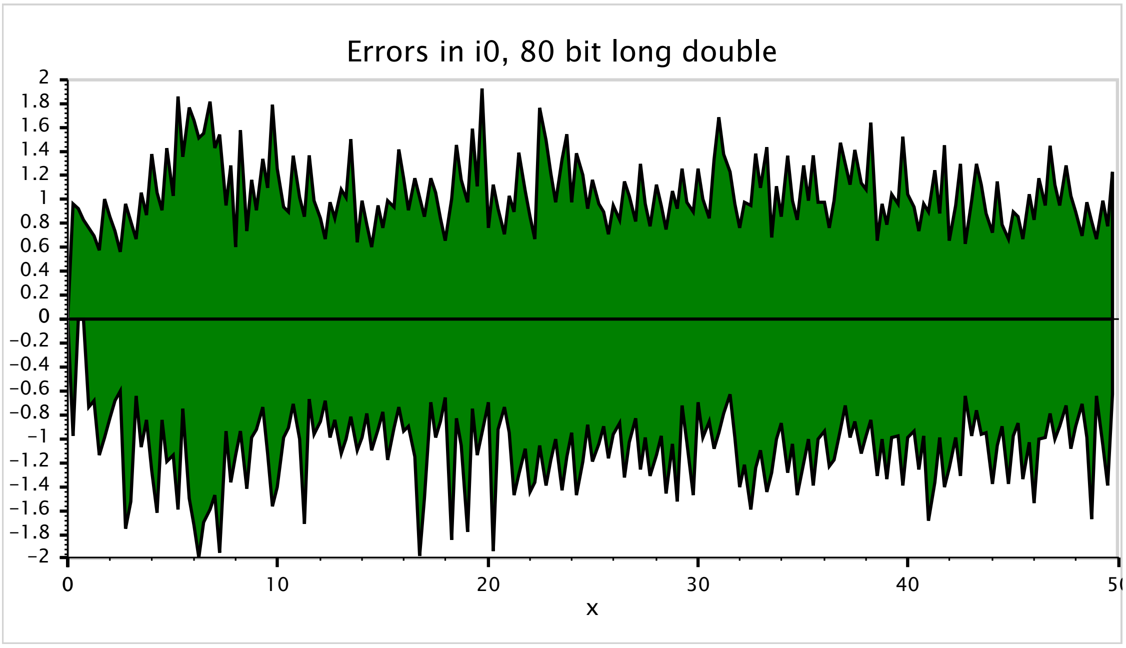
	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel K0: Mathworld Data (Integer Version)	Max = 0.833ε (Mean = 0.436ε) (*<cmath>:* Max = 9.33ε (Mean = 3.25ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.2ε (Mean = 0.733ε)) (*Rmath 3.2.3:* Max = 0.833ε (Mean = 0.601ε))	Max = 0.833ε (Mean = 0.436ε)	Max = 0.833ε (Mean = 0.552ε)
Bessel K1: Mathworld Data (Integer Version)	Max = 0.786ε (Mean = 0.329ε) (*<cmath>:* Max = 8.94ε (Mean = 3.19ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.626ε (Mean = 0.333ε)) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.516ε))	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel Kn: Mathworld Data (Integer Version)	Max = 2.6ε (Mean = 1.21ε) (*<cmath>:* Max = 12.9ε (Mean = 4.91ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 168ε (Mean = 59.5ε)) (*Rmath 3.2.3:* Max = 8.48ε (Mean = 2.98ε))	Max = 2.6ε (Mean = 1.21ε)	Max = 3.63ε (Mean = 1.46ε)

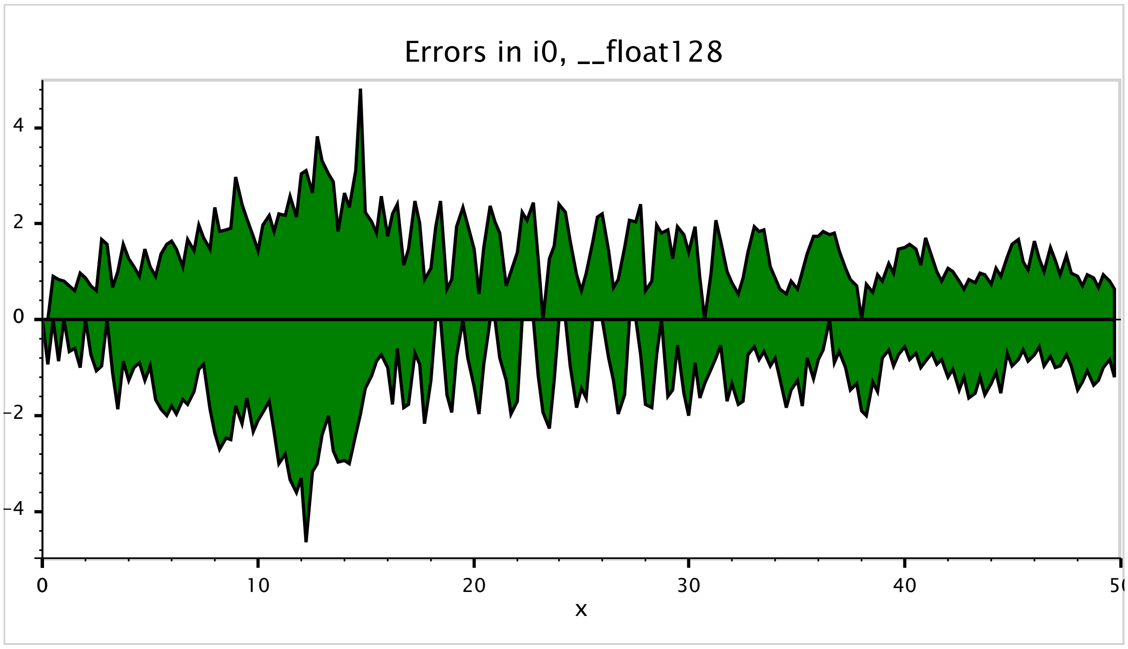
Table 70. Error rates for cyl_bessel_k

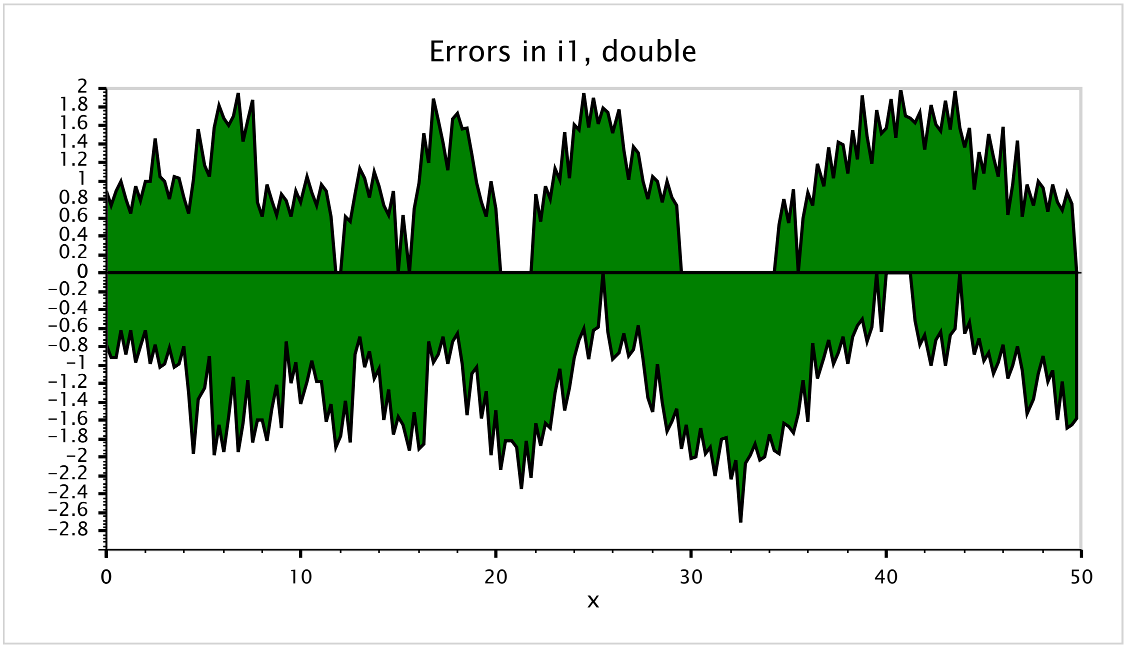
	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel K0: Mathworld Data	Max = 0.833ε (Mean = 0.436ε) (*<cmath>:* Max = 9.33ε (Mean = 3.25ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.04ε (Mean = 2.16ε)) (*Rmath 3.2.3:* Max = 0.833ε (Mean = 0.601ε))	Max = 0.833ε (Mean = 0.436ε)	Max = 0.833ε (Mean = 0.552ε)
Bessel K1: Mathworld Data	Max = 0.786ε (Mean = 0.329ε) (*<cmath>:* Max = 8.94ε (Mean = 3.19ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.26ε (Mean = 2.21ε)) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.516ε))	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel Kn: Mathworld Data	Max = 2.6ε (Mean = 1.21ε) (*<cmath>:* Max = 12.9ε (Mean = 4.91ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.36ε (Mean = 1.43ε) And other failures.) (*Rmath 3.2.3:* Max = 8.48ε (Mean = 2.98ε))	Max = 2.6ε (Mean = 1.21ε)	Max = 3.63ε (Mean = 1.46ε)
Bessel Kv: Mathworld Data	Max = 3.58ε (Mean = 2.39ε) (*<cmath>:* Max = 13ε (Mean = 4.81ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.47ε (Mean = 2.04ε) And other failures.) (*Rmath 3.2.3:* Max = 3.15ε (Mean = 1.35ε))	Max = 5.21ε (Mean = 2.53ε)	Max = 4.78ε (Mean = 2.19ε)
Bessel Kv: Mathworld Data (large values)	Max = 42.3ε (Mean = 21ε) (*<cmath>:* Max = 42.3ε (Mean = 19.8ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 308ε (Mean = 142ε) And other failures.) (*Rmath 3.2.3:* Max = 84.6ε (Mean = 37.8ε))	Max = 42.3ε (Mean = 21ε)	Max = 59.8ε (Mean = 26.9ε)
Bessel Kn: Random Data	Max = 4.55ε (Mean = 1.12ε) (*<cmath>:* Max = 13.9ε (Mean = 2.91ε))	Max = 0.764ε (Mean = 0.0348ε) (*GSL 2.1:* Max = 8.71ε (Mean = 1.76ε) And other failures.) (*Rmath 3.2.3:* Max = 7.47ε (Mean = 1.34ε))	Max = 4.55ε (Mean = 1.12ε)	Max = 9.34ε (Mean = 1.7ε)
Bessel Kv: Random Data	Max = 7.88ε (Mean = 1.48ε) (*<cmath>:* Max = 13.6ε (Mean = 2.68ε) And other failures.)	Max = 0.507ε (Mean = 0.0313ε) (*GSL 2.1:* Max = 9.71ε (Mean = 1.47ε) And other failures.) (*Rmath 3.2.3:* Max = 7.37ε (Mean = 1.49ε))	Max = 7.88ε (Mean = 1.47ε)	Max = 8.33ε (Mean = 1.62ε)

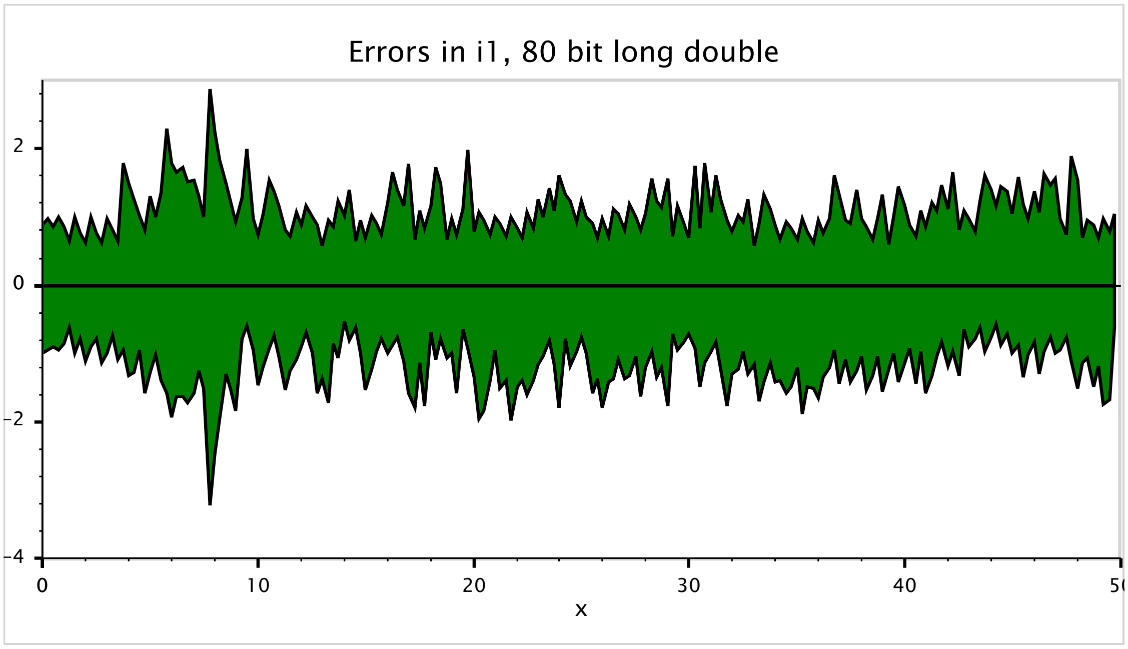
The following error plot are based on an exhaustive search of the functions domain for I0, I1, K0, and K1, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.

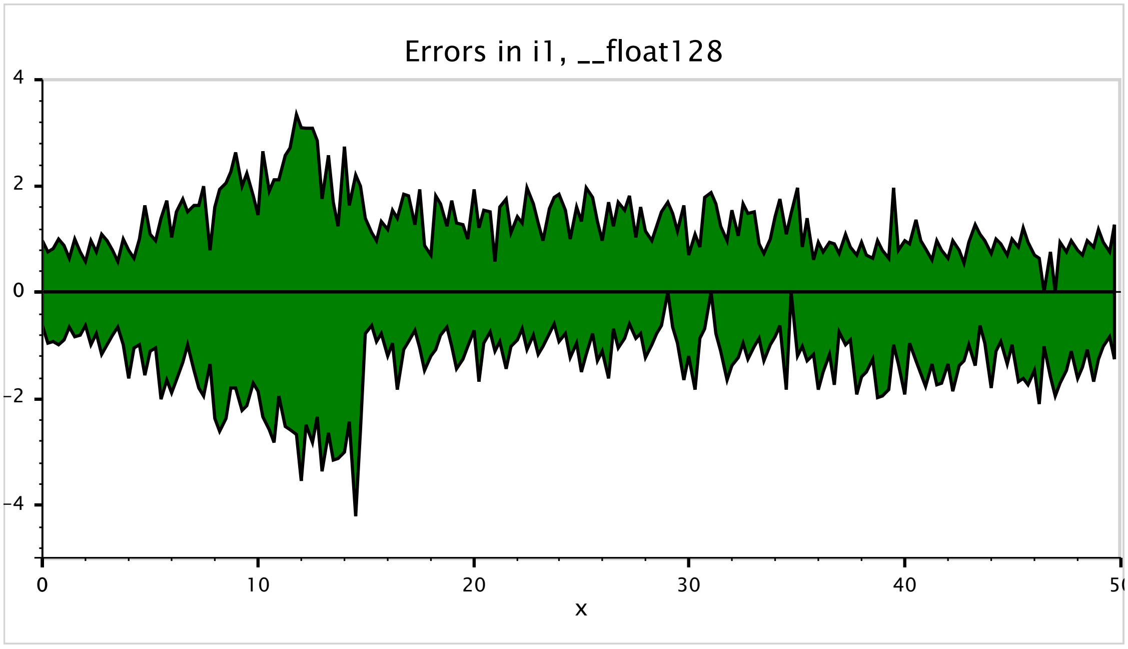


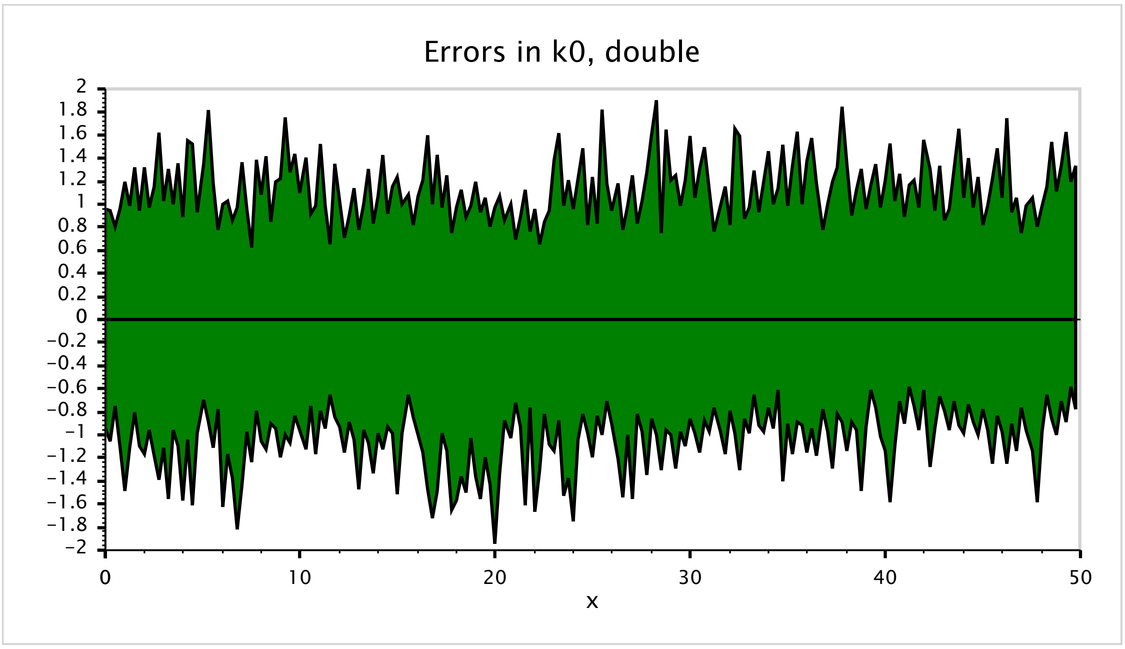


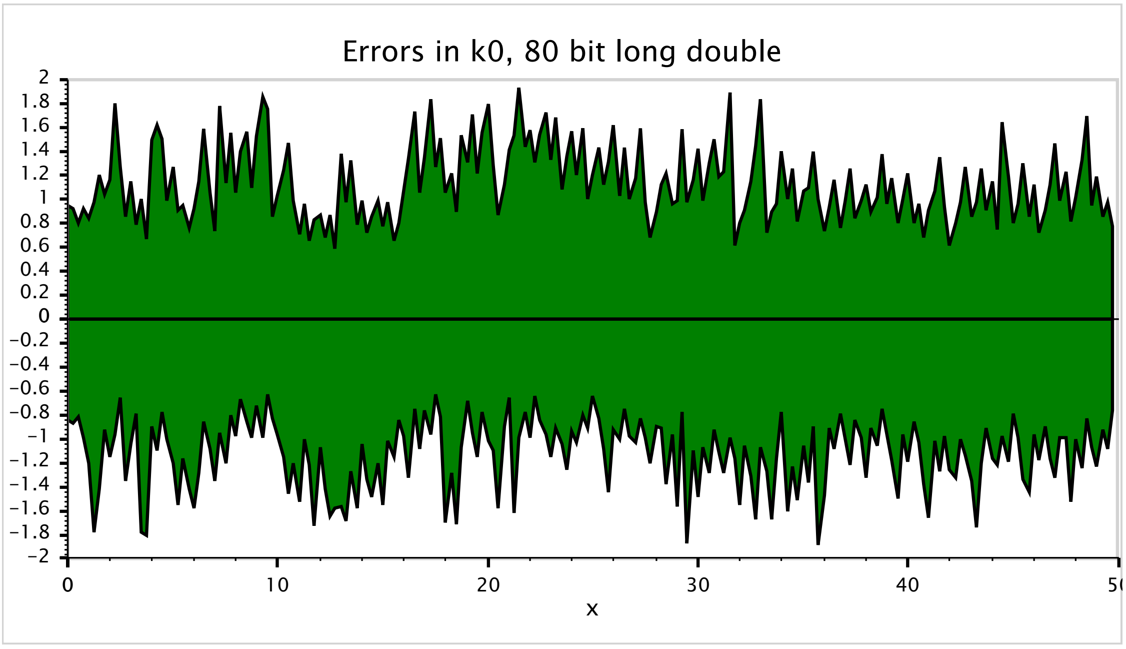

Errors in i0, __float128
4
2
0
-2
-4
0
10
20
30
40
50
x

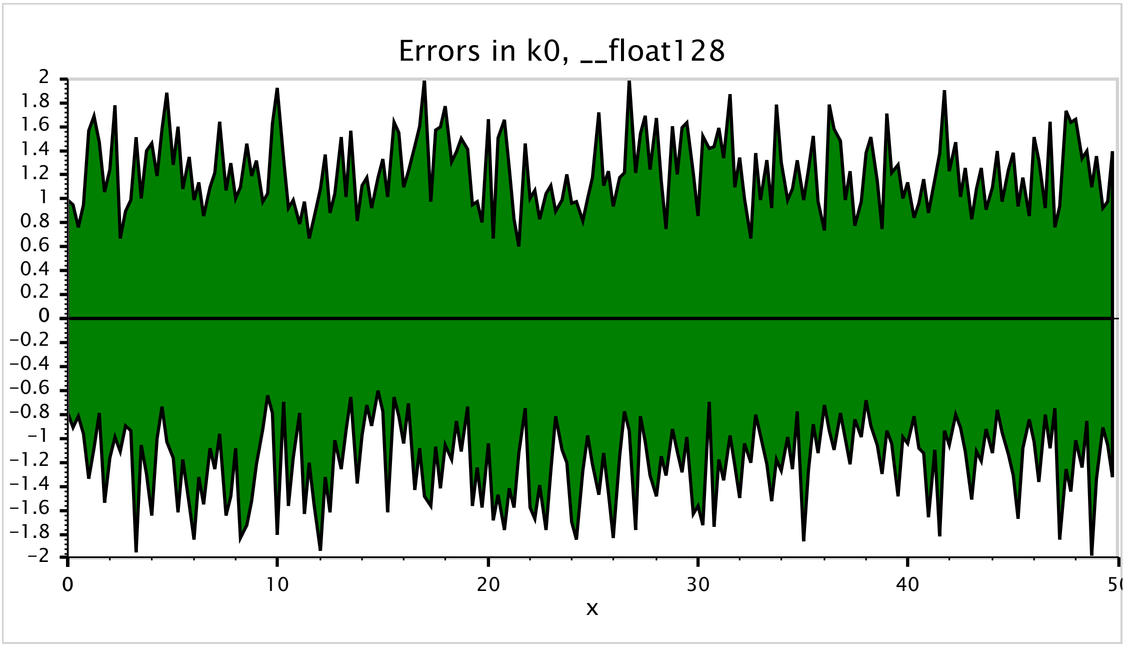

Errors in i1, double
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1.2
−1.4
−1.6
−1.8
−2
−2.2
−2.4
−2.6
−2.8
0
10
20
30
40
50
x

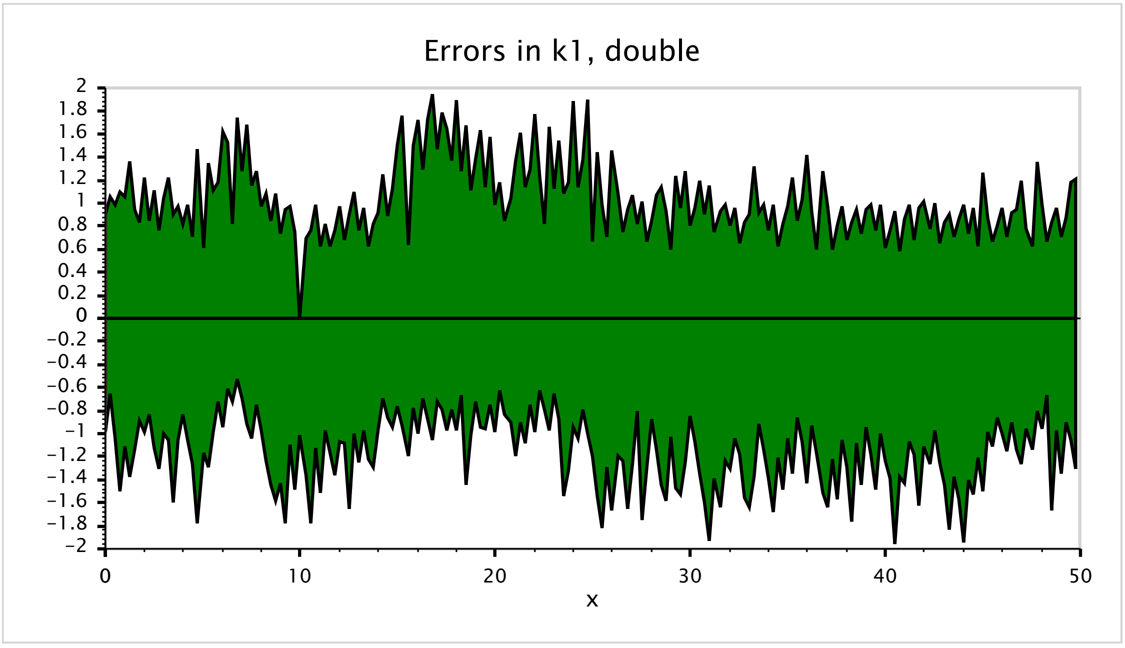

Errors in i1, 80 bit long double
2
0
-2
-4
0
10
20
30
40
50
x

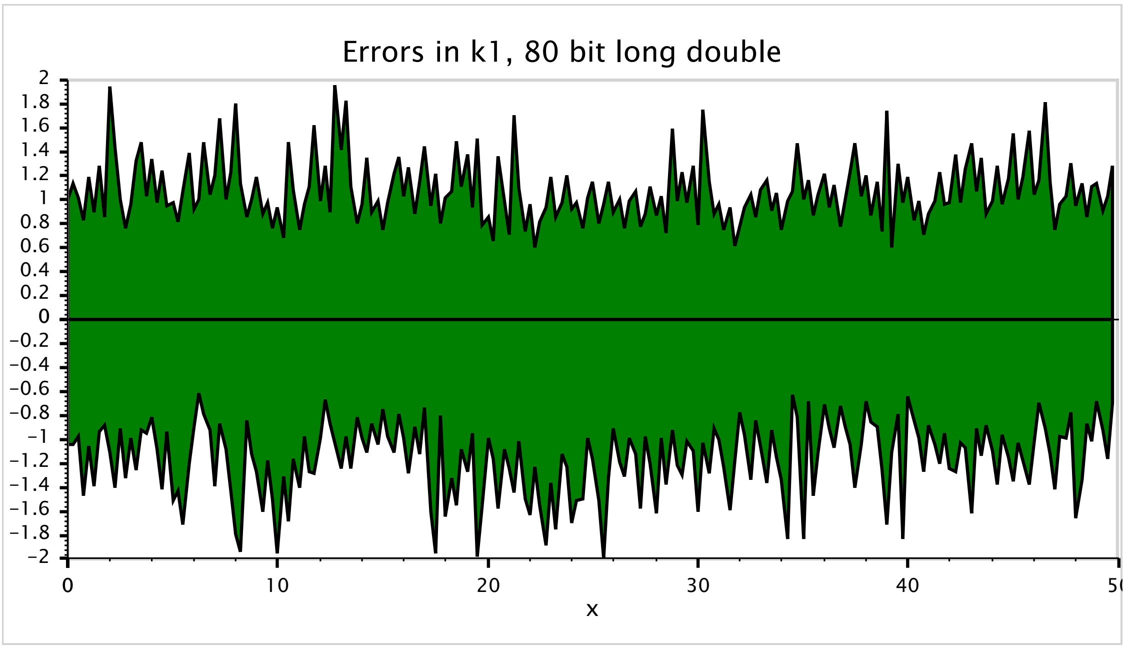

Errors in i1, __float128
4
2
0
-2
-4
0
10
20
30
40
50
x

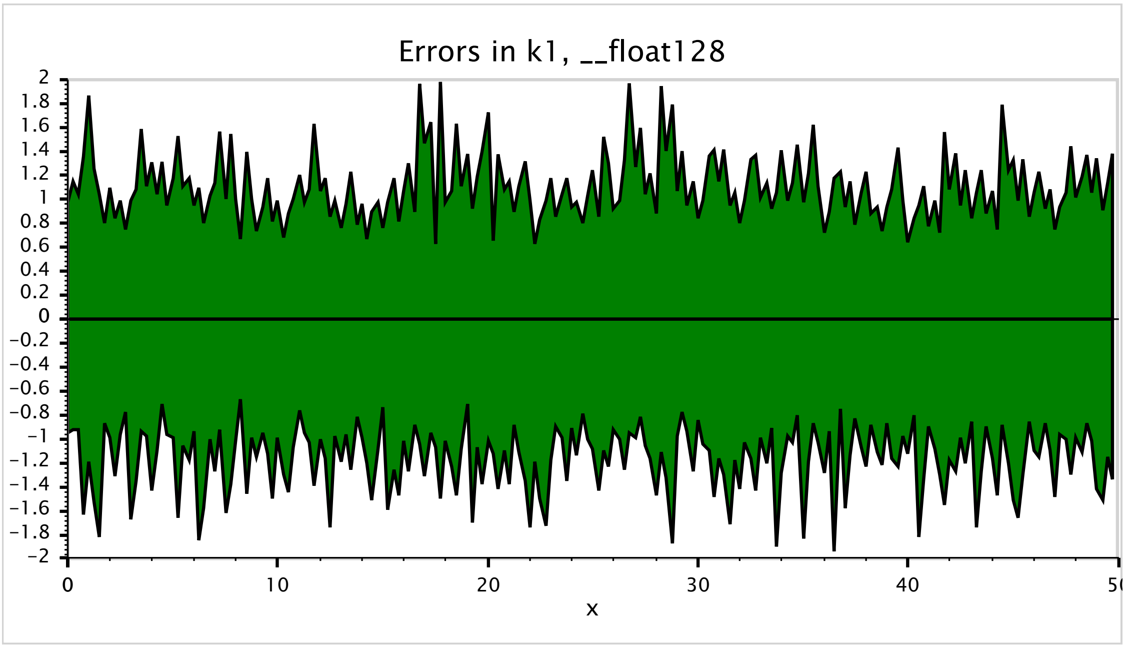

Errors in k0, double
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1.2
−1.4
−1.6
−1.8
−2
0
10
20
30
40
50
x


Errors in k0, 80 bit long double
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1.2
−1.4
−1.6
−1.8
−2
0
10
20
30
40
x


Errors in k0, __float128
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1.2
−1.4
−1.6
−1.8
−2
0
10
20
30
40
50
x


Errors in k1, double
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1.2
−1.4
−1.6
−1.8
−2
0
10
20
30
40
50
x





Implementation

The following are handled as special cases first:

When computing I_v for *x* < *0*, then ν must be an integer or a domain error occurs. If ν is an integer, then the function is odd if ν is odd and even if ν is even, and we can reflect to *x* > *0*.

For I_v with v equal to 0, 1 or 0.5 are handled as special cases.

The 0 and 1 cases use polynomial approximations on finite and infinite intervals. The approximating forms are based on "Rational Approximations for the Modified Bessel Function of the First Kind - $I_0(x)$ for Computations with Double Precision" by Pavel Holoborodko, extended by us to deal with up to 128-bit precision (with different approximations for each target precision).

$$I_0(x) \approx \left[\frac{x}{2}\right]^2 P\left(\left[\frac{x}{2}\right]^2\right) + 1 \quad ; \quad x < 7.75$$

$$I_0(x) \approx \frac{e^x}{\sqrt{x}} P\left(\frac{1}{x}\right) \quad ; \quad x > 7.75$$

$$I_1(x) \approx \frac{x}{2}\left(1 + \frac{1}{2}\left[\frac{x}{2}\right]^2 + \left[\frac{x}{2}\right]^4 P\left(\left[\frac{x}{2}\right]^2\right)\right) \quad ; \quad x < 7.75$$

$$I_1(x) \approx \frac{e^x}{\sqrt{x}} P\left(\frac{1}{x}\right) \quad ; \quad x > 7.75$$

Similarly we have:

$$K_0(x) + \ln(x) I_0(x) = R\left(x^2\right) \quad ; \quad I_0(x) = \left[\frac{x}{2}\right]^2 R\left(\left[\frac{x}{2}\right]^2\right) + 1 \quad ; \quad x < 1$$

$$K_0(x) = R\left(\frac{1}{x}\right)\frac{e^{-x}}{\sqrt{x}} \quad ; \quad x > 1$$

$$K_1(x) - \ln(x) I_1(x) - 1 = xR\left(x^2\right) \quad ; \quad I_1(x) = \frac{x}{2}\left(1 + \frac{1}{2}\left[\frac{x}{2}\right]^2 + \left[\frac{x}{2}\right]^4 R\left(\left[\frac{x}{2}\right]^2\right)\right) \quad ; \quad x$$

$$K_1(x) = R\left(\frac{1}{x}\right)\frac{e^{-x}}{\sqrt{x}} \quad ; \quad x > 1$$

The 0.5 case is a simple trigonometric function:

$I_{0.5}(x) = \mathrm{sqrt}(2 / \pi x) * \sinh(x)$

For K_v with *v* an integer, the result is calculated using the recurrence relation:

$$K_{v+1}(z) = \frac{2v}{z} K_v(z) + K_{v-1}(z)$$

starting from K_0 and K_1 which are calculated using rational the approximations above. These rational approximations are accurate to around 19 digits, and are therefore only used when T has no more than 64 binary digits of precision.

When *x* is small compared to *v*, $I_v x$ is best computed directly from the series:

$$I_v(x) \quad = \quad \sum_{k=0}^{\infty} \frac{1}{\Gamma(k+v+1)k!}\left(\frac{x}{2}\right)^{2k+v}$$

In the general case, we first normalize ν to `[0, [inf])` with the help of the reflection formulae:

$$I_{-v}(z) = I_v(z) + \frac{2}{\pi}\sin(v\pi) K_v(z)$$

$$K_{-v}(z) = K_v(z)$$

Let μ = ν - floor(ν + 1/2), then μ is the fractional part of ν such that |μ| <= 1/2 (we need this for convergence later). The idea is to calculate $K_\mu(x)$ and $K_{\mu+1}(x)$, and use them to obtain $I_\nu(x)$ and $K_\nu(x)$.

The algorithm is proposed by Temme in N.M. Temme, *On the numerical evaluation of the modified bessel function of the third kind*, Journal of Computational Physics, vol 19, 324 (1975), which needs two continued fractions as well as the Wronskian:

$$\mathrm{CF1}\ :\ f_v = \frac{I_{v+1}}{I_v} = \frac{1}{\frac{2(v+1)}{x}+}\frac{1}{\frac{2(v+2)}{x}+}\cdots$$

$$\mathrm{CF2}\ :\ \frac{z_1}{z_0} = \frac{1}{2(x+1)+}\frac{v^2-\left(\frac{3}{2}\right)^2}{2(x+2)+}\frac{v^2-\left(\frac{5}{2}\right)^2}{2(x+3)+}\cdots$$

$$W = I_v(z)K_v^{'}(z) - K_v(z)I_v^{'}(z) = -\left[I_v(z)K_{v+1}(z) + K_v(z)I_{v+1}(z)\right] = -\frac{1}{z}$$

The continued fractions are computed using the modified Lentz's method (W.J. Lentz, *Generating Bessel functions in Mie scattering calculations using continued fractions*, Applied Optics, vol 15, 668 (1976)). Their convergence rates depend on *x*, therefore we need different strategies for large *x* and small *x*.

$x > v$, CF1 needs O(*x*) iterations to converge, CF2 converges rapidly.

$x <= v$, CF1 converges rapidly, CF2 fails to converge when x -> 0.

When *x* is large ($x > 2$), both continued fractions converge (CF1 may be slow for really large *x*). K_μ and $K_{\mu+1}$ can be calculated by

$$\begin{aligned} K_\mu &= \sqrt{\pi}(2x)^\mu e^{-x} z_0 \\ K_{\mu+1} &= \frac{K_\mu}{x}\left(\frac{1}{2} + \mu + x + \left(\mu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right) \end{aligned}$$

where

$$z_0 = \frac{1}{1+S}\left(\frac{1}{2x}\right)^{\mu+\frac{1}{2}}$$

S is also a series that is summed along with CF2, see I.J. Thompson and A.R. Barnett, *Modified Bessel functions I_v and K_v of real order and complex argument to selected accuracy*, Computer Physics Communications, vol 47, 245 (1987).

When *x* is small ($x <= 2$), CF2 convergence may fail (but CF1 works very well). The solution here is Temme's series:

$$\begin{aligned} K_\mu &= \sum_{k=0}^{\infty} c_k f_k \\ K_{\mu+1} &= \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k \end{aligned}$$

where

$$c_k = \frac{1}{k!}\left(\frac{x^2}{4}\right)^k$$

f_k and h_k are also computed by recursions (involving gamma functions), but the formulas are a little complicated, readers are referred to N.M. Temme, *On the numerical evaluation of the modified Bessel function of the third kind*, Journal of Computational Physics, vol 19, 324 (1975). Note: Temme's series converge only for $|\mu| <= 1/2$.

$K_v(x)$ is then calculated from the forward recurrence, as is $K_{v+1}(x)$. With these two values and f_v, the Wronskian yields $I_v(x)$ directly.

Spherical Bessel Functions of the First and Second Kinds

Synopsis

```
#include <boost/math/special_functions/bessel.hpp>
```

```
template <class T1, class T2>
calculated-result-type sph_bessel(unsigned v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type sph_bessel(unsigned v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type sph_neumann(unsigned v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type sph_neumann(unsigned v, T2 x, const Policy&);
```

Description

The functions sph_bessel and sph_neumann return the result of the Spherical Bessel functions of the first and second kinds respectively:

sph_bessel(v, x) = $j_v(x)$

sph_neumann(v, x) = $y_v(x) = n_v(x)$

where:

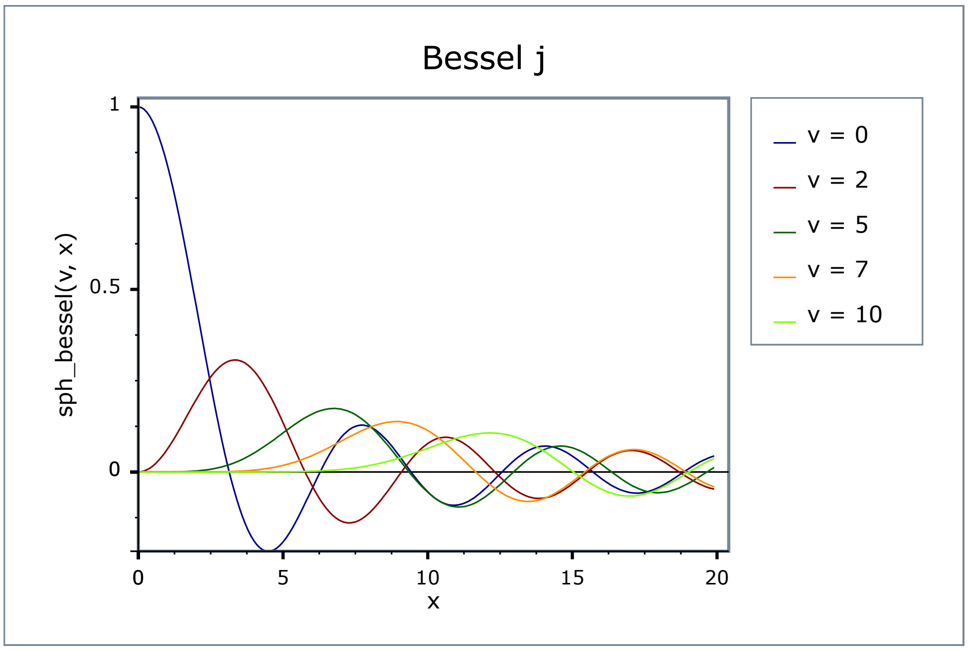
$$\begin{aligned} j_n(z) &= \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z) \\ y_n(z) &= \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z) \end{aligned}$$

The return type of these functions is computed using the *result type calculation rules* for the single argument type T.

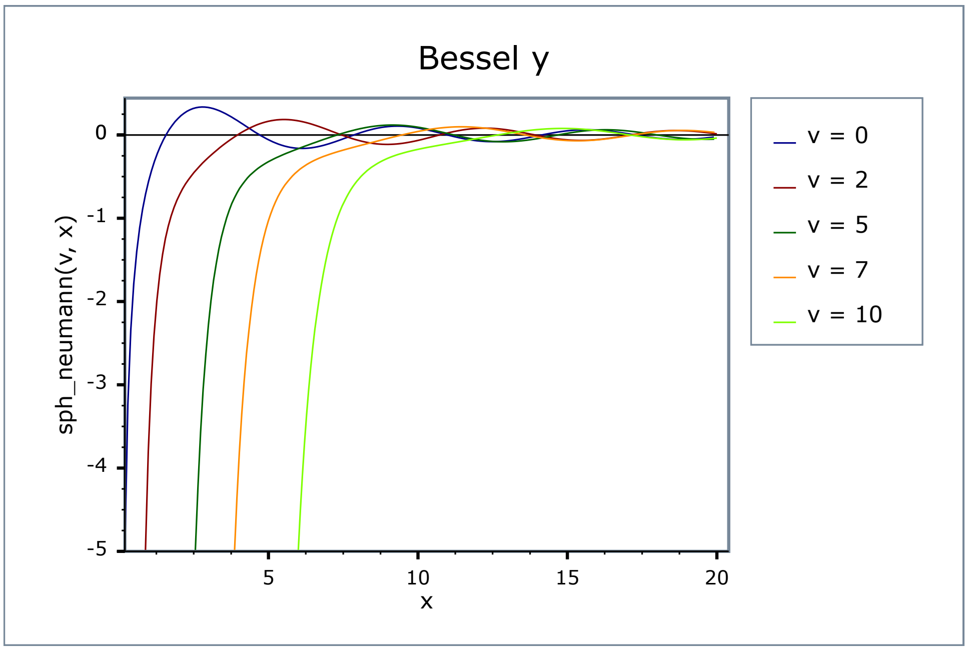
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The functions return the result of domain_error whenever the result is undefined or complex: this occurs when `x < 0`.

The j_v function is cyclic like J_v but differs in its behaviour at the origin:



Likewise y_v is also cyclic for large x, but tends to -∞ for small *x*:



Testing

There are two sets of test values: spot values calculated using functions.wolfram.com, and a much larger set of tests computed using a simplified version of this implementation (with all the special case handling removed).

Accuracy

Table 71. Error rates for sph_bessel

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel j: Random Data	Max = 243ε (Mean = 13.3ε) (*<cmath>:* Max = 1.91e+06ε (Mean = 1.09e+05ε))	Max = 0.978ε (Mean = 0.0445ε) (*GSL 2.1:* Max = 1.79e+03ε (Mean = 107ε))	Max = 243ε (Mean = 33.7ε)	Max = 245ε (Mean = 16.3ε)

Table 72. Error rates for sph_neumann

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
y: Random Data	Max = 234ε (Mean = 19.5ε) (*<cmath>:* Max = 1.6e+06ε (Mean = 1.4e+05ε))	Max = 0.995ε (Mean = 0.0665ε) (*GSL 2.1:* Max = 8.5e+04ε (Mean = 5.33e+03ε))	Max = 234ε (Mean = 19.8ε)	Max = 281ε (Mean = 31.1ε)

Implementation

Other than error handling and a couple of special cases these functions are implemented directly in terms of their definitions:

$$\begin{aligned} j_n(z) &= \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z) \\ y_n(z) &= \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z) \end{aligned}$$

The special cases occur for:

j_0 = sinc_pi(x) = sin(x) / x

and for small *x < 1*, we can use the series:

$$j_v(z) = \sqrt{\frac{\pi}{4}} \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{2k+n}}{k!\,\Gamma\left(n+k+1+\frac{1}{2}\right)}$$

which neatly avoids the problem of calculating 0/0 that can occur with the main definition as x → 0.

Derivatives of the Bessel Functions

Synopsis

```
#include <boost/math/special_functions/bessel_prime.hpp>
```

```
template <class T1, class T2>
calculated-result-type cyl_bessel_j_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_j_prime(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type cyl_neumann_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_neumann_prime(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type cyl_bessel_i_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_i_prime(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type cyl_bessel_k_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type cyl_bessel_k_prime(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type sph_bessel_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type sph_bessel_prime(T1 v, T2 x, const Policy&);

template <class T1, class T2>
calculated-result-type sph_neumann_prime(T1 v, T2 x);

template <class T1, class T2, class Policy>
calculated-result-type sph_neumann_prime(T1 v, T2 x, const Policy&);
```

Description

These functions return the first derivative with respect to *x* of the corresponding Bessel function.

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types. The functions are also optimised for the relatively common case that T1 is an integer.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The functions return the result of domain_error whenever the result is undefined or complex.

Testing

There are two sets of test values: spot values calculated using wolframalpha.com, and a much larger set of tests computed using a relation to the underlying Bessel functions that the implementation does not use.

Accuracy

The accuracy of these functions is broadly similar to the underlying Bessel functions.

Table 73. Error rates for cyl_bessel_i_prime (integer orders)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel I'0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.354ε)
Bessel I'1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.36ε (Mean = 0.782ε)
Bessel I'n: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 2.31ε (Mean = 1.41ε)	Max = 701ε (Mean = 212ε)	Max = 3.61ε (Mean = 1.22ε)

Table 74. Error rates for cyl_bessel_i_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel I'0: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.354ε)
Bessel I'1: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.36ε (Mean = 0.782ε)
Bessel I'n: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 2.31ε (Mean = 1.41ε)	Max = 701ε (Mean = 212ε)	Max = 3.61ε (Mean = 1.22ε)
Bessel I'v: Mathworld Data	Max = 1.62ε (Mean = 0.512ε)	Max = 2.89e+03ε (Mean = 914ε)	Max = 2.89e+03ε (Mean = 914ε)	Max = 3.76e+03ε (Mean = 1.19e+03ε)
Bessel I'n: Random Data	Max = 0ε (Mean = 0ε)	Max = 3.95ε (Mean = 1.06ε)	Max = 195ε (Mean = 37.1ε)	Max = 9.85ε (Mean = 1.82ε)
Bessel I'v: Random Data	Max = 0ε (Mean = 0ε)	Max = 14.1ε (Mean = 2.93ε)	Max = 336ε (Mean = 68.5ε)	Max = 14ε (Mean = 2.5ε)
Bessel I'v: Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 42.6ε (Mean = 20.2ε)	Max = 42.6ε (Mean = 20.2ε)	Max = 59.5ε (Mean = 26.6ε)

Table 75. Error rates for cyl_bessel_j_prime (integer orders)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel J0': Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 18.9ε (Mean = 6.82ε)	Max = 18.9ε (Mean = 6.72ε)	Max = 6.62ε (Mean = 2.55ε)
Bessel J0': Mathworld Data (Tricky cases) (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 7.44ε (Mean = 3.34ε)	Max = 7.44ε (Mean = 3.31ε)	Max = 3.67ε (Mean = 1.74ε)
Bessel J1': Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 0.999ε (Mean = 0.627ε)
Bessel J1': Mathworld Data (tricky cases) (Integer Version)	Max = 287ε (Mean = 129ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 288ε (Mean = 129ε)
Bessel JN': Mathworld Data (Integer Version)	Max = 0.527ε (Mean = 0.128ε)	Max = 1.29e+03ε (Mean = 312ε)	Max = 1.29e+03ε (Mean = 355ε)	Max = 14ε (Mean = 6.13ε)

Table 76. Error rates for cyl_bessel_j_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel J0': Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 18.9ε (Mean = 6.82ε)	Max = 18.9ε (Mean = 6.72ε)	Max = 6.62ε (Mean = 2.55ε)
Bessel J0': Mathworld Data (Tricky cases)	Max = 0ε (Mean = 0ε)	Max = 7.44ε (Mean = 3.34ε)	Max = 7.44ε (Mean = 3.31ε)	Max = 3.67ε (Mean = 1.74ε)
Bessel J1': Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 0.999ε (Mean = 0.627ε)
Bessel J1': Mathworld Data (tricky cases)	Max = 287ε (Mean = 129ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 288ε (Mean = 129ε)
Bessel JN': Mathworld Data	Max = 0.527ε (Mean = 0.128ε)	Max = 1.29e+03ε (Mean = 312ε)	Max = 1.29e+03ε (Mean = 355ε)	Max = 14ε (Mean = 6.13ε)
Bessel J': Mathworld Data	Max = 21.5ε (Mean = 4.7ε)	Max = 42.5ε (Mean = 9.31ε)	Max = 42.5ε (Mean = 9.32ε)	Max = 23.7ε (Mean = 8ε)
Bessel J': Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 989ε (Mean = 495ε)	Max = 989ε (Mean = 495ε)	Max = 2.9ε (Mean = 1.61ε)
Bessel JN': Random Data	Max = 0.593ε (Mean = 0.0396ε)	Max = 11.3ε (Mean = 1.85ε)	Max = 79.4ε (Mean = 16.2ε)	Max = 6.34ε (Mean = 0.999ε)
Bessel J': Random Data	Max = 0.885ε (Mean = 0.033ε)	Max = 139ε (Mean = 6.47ε)	Max = 279ε (Mean = 27.2ε)	Max = 176ε (Mean = 9.75ε)
Bessel J': Random Data (Tricky large values)	Max = 0ε (Mean = 0ε)	Max = 474ε (Mean = 62.2ε)	Max = 474ε (Mean = 64.5ε)	Max = 379ε (Mean = 45.4ε)

Table 77. Error rates for cyl_bessel_k_prime (integer orders)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K'0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel K'1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.761ε (Mean = 0.444ε)
Bessel K'n: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 4.17ε (Mean = 1.75ε)

Table 78. Error rates for cyl_bessel_k_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K'0: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel K'1: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.761ε (Mean = 0.444ε)
Bessel K'n: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 4.17ε (Mean = 1.75ε)
Bessel K'v: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 3.94ε (Mean = 2.44ε)	Max = 3.94ε (Mean = 2.34ε)	Max = 3.94ε (Mean = 1.47ε)
Bessel K'v: Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 59.2ε (Mean = 42.9ε)	Max = 58.7ε (Mean = 42.6ε)	Max = 18.6ε (Mean = 11.8ε)
Bessel K'n: Random Data	Max = 0ε (Mean = 0ε)	Max = 4.45ε (Mean = 1.19ε)	Max = 4.45ε (Mean = 1.19ε)	Max = 9.67ε (Mean = 1.73ε)
Bessel K'v: Random Data	Max = 0ε (Mean = 0ε)	Max = 7.95ε (Mean = 1.53ε)	Max = 7.95ε (Mean = 1.52ε)	Max = 8.32ε (Mean = 1.65ε)

Table 79. Error rates for sph_bessel_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel j': Random Data	Max = 0.753ε (Mean = 0.0343ε)	Max = 167ε (Mean = 12ε)	Max = 167ε (Mean = 33.2ε)	Max = 307ε (Mean = 25.2ε)

Table 80. Error rates for sph_neumann_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
y': Random Data	Max = 0.988ε (Mean = 0.0869ε)	Max = 158ε (Mean = 18.8ε)	Max = 158ε (Mean = 20.2ε)	Max = 296ε (Mean = 25.6ε)

Implementation

In the general case, the derivatives are calculated using the relations:

$$\begin{aligned}
J'_v(x) &= \frac{J_{v-1}(x) - J_{v+1}(x)}{2} \\
Y'_v(x) &= \frac{Y_{v-1}(x) - Y_{v+1}(x)}{2} \\
I'_v(x) &= \frac{I_{v-1}(x) + I_{v+1}(x)}{2} \\
K'_v(x) &= \frac{K_{v-1}(x) + K_{v+1}(x)}{-2} \\
j'_n &= \left(\frac{n}{x}\right) j_n(x) - j_{n+1}(x) \\
y'_n &= \left(\frac{n}{x}\right) y_n(x) - y_{n+1}(x)
\end{aligned}$$

There are also a number of special cases, for large x we have:

$$\begin{aligned}
J'_v(x) &= N_v \cos\phi_v \\
Y'_v(x) &= N_v \sin\phi_v \\
N_v^2 &\approx \frac{2}{\pi x}\left\{1 - \frac{1}{2}\frac{\mu - 3}{(2x)^2} - \frac{1}{8}\frac{(\mu - 1)(\mu - 45)}{(2x)^4}\right\} \\
\phi_v &\approx x - \left(\frac{1}{2}v - \frac{1}{4}\right)\pi + \frac{\mu + 3}{2(4x)} + \frac{\mu^2 + 46\mu - 63}{6(4x)^3} + \frac{\mu^3 + 185\mu^2 - 2053\mu + 1899}{5(4x)^5} \\
\mu &= 4v^2
\end{aligned}$$

And for small x:

$$\begin{aligned}
J'_v(z) &= \frac{z^{v-1}}{2^v}\sum_{k=0}^{\infty}\frac{\left(-\frac{z^2}{4}\right)^k (v + 2k)}{k!\,\Gamma(1 + k + v)} \\
Y'_v(x) &= -\frac{\Gamma(-v)\cos(v\pi)}{\pi} \times \frac{z^{v-1}}{2^v} \times \sum_{k=0}^{\infty}\frac{\left(-\frac{z^2}{4}\right)^k (v + 2k)}{(v + 1)_k k!} - \frac{\Gamma(v)}{\pi} \times \frac{2^v}{z^{v+1}} \times \sum_{k=0}^{\infty}\frac{\left(-\frac{z^2}{4}\right)^k (2k - v)}{(1 - v)_k k!}
\end{aligned}$$

Hankel Functions

Cyclic Hankel Functions

Synopsis

```
template <class T1, class T2>
std::complex<calculated-result-type> cyl_hankel_1(T1 v, T2 x);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> cyl_hankel_1(T1 v, T2 x, const Policy&);

template <class T1, class T2>
std::complex<calculated-result-type> cyl_hankel_2(T1 v, T2 x);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> cyl_hankel_2(T1 v, T2 x, const Policy&);
```

Description

The functions cyl_hankel_1 and cyl_hankel_2 return the result of the Hankel functions of the first and second kind respectively:

$$cyl\_hankel\_1(v, x) = H_v^{(1)}(x) = J_v(x) + i\,Y_v(x)$$

$$cyl\_hankel\_2(v, x) = H_v^{(2)}(x) = J_v(x) - i\,Y_v(x)$$

where:

$J_v(x)$ is the Bessel function of the first kind, and $Y_v(x)$ is the Bessel function of the second kind.

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types. The functions are also optimised for the relatively common case that T1 is an integer.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that while the arguments to these functions are real values, the results are complex. That means that the functions can only be instantiated on types `float`, `double` and `long double`. The functions have also been extended to operate over the whole range of *v* and *x* (unlike cyl_bessel_j and cyl_neumann).

Performance

These functions are generally more efficient than two separate calls to the underlying Bessel functions as internally Bessel J and Y can be computed simultaneously.

Testing

There are just a few spot tests to exercise all the special case handling - the bulk of the testing is done on the Bessel functions upon which these are based.

Accuracy

Refer to cyl_bessel_j and cyl_neumann.

Implementation

For $x < 0$ the following reflection formulae are used:

$$
\begin{aligned}
J_v(-z) &= (-z)^v z^{-v} J_v(z) \\
Y_v(-z) &= z^v Y_v(z)(-z)^{-v} + ((-z)^v z^{-v} - (-z)^{-v} z^v) J_v(z)\cot(\pi v) \quad ; \quad v \notin \mathbb{Z} \\
Y_v(-z) &= (-1)^v \left(Y_v(z) - \frac{2}{\pi}(\log(z) - \log(-z)) J_v(z) \right) \quad ; \quad v \in \mathbb{Z}
\end{aligned}
$$

Otherwise the implementation is trivially in terms of the Bessel J and Y functions.

Note however, that the Hankel functions compute the Bessel J and Y functions simultaneously, and therefore a single Hankel function call is more efficient than two Bessel function calls. The one exception is when *v* is a small positive integer, in which case the usual Bessel function routines for integer order are used.

Spherical Hankel Functions

Synopsis

```
template <class T1, class T2>
std::complex<calculated-result-type> sph_hankel_1(T1 v, T2 x);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> sph_hankel_1(T1 v, T2 x, const Policy&);

template <class T1, class T2>
std::complex<calculated-result-type> sph_hankel_2(T1 v, T2 x);

template <class T1, class T2, class Policy>
std::complex<calculated-result-type> sph_hankel_2(T1 v, T2 x, const Policy&);
```

Description

The functions sph_hankel_1 and sph_hankel_2 return the result of the spherical Hankel functions of the first and second kind respectively:

$$
\begin{aligned}
h_v^{(1)}(x) &= \sqrt{\frac{\pi}{2}} \frac{1}{\sqrt{x}} H_{v+\frac{1}{2}}^{(1)}(x) \\
h_v^{(2)}(x) &= \sqrt{\frac{\pi}{2}} \frac{1}{\sqrt{x}} H_{v+\frac{1}{2}}^{(2)}(x)
\end{aligned}
$$

The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types. The functions are also optimised for the relatively common case that T1 is an integer.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that while the arguments to these functions are real values, the results are complex. That means that the functions can only be instantiated on types `float`, `double` and `long double`. The functions have also been extended to operate over the whole range of *v* and *x* (unlike cyl_bessel_j and cyl_neumann).

Testing

There are just a few spot tests to exercise all the special case handling - the bulk of the testing is done on the Bessel functions upon which these are based.

Accuracy

Refer to cyl_bessel_j and cyl_neumann.

Implementation

These functions are trivially implemented in terms of cyl_hankel_1 and cyl_hankel_2.

Airy Functions

Airy Ai Function

Synopsis

```
#include <boost/math/special_functions/airy.hpp>
```

```
namespace boost { namespace math {

 template <class T>
 calculated-result-type airy_ai(T x);

 template <class T, class Policy>
 calculated-result-type airy_ai(T x, const Policy&);

}} // namespaces
```

Description

The function airy_ai calculates the Airy function Ai which is the first solution to the differential equation:

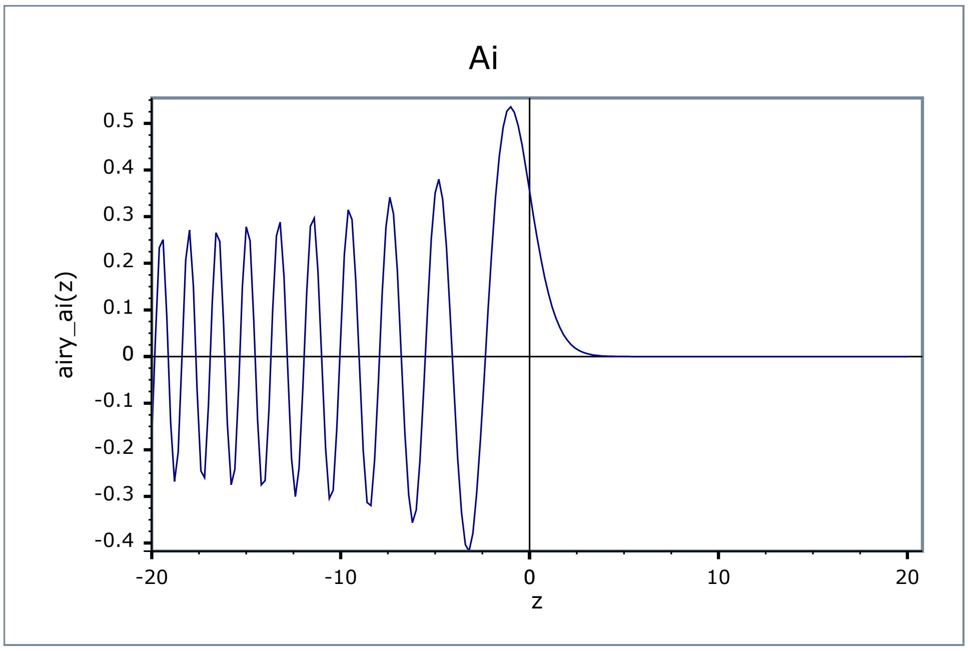
$$\frac{d^2w}{dz^2} = zw \quad ; \quad w = Ai(z),\ Bi(z),\ Ai\left(ze^{\pm\frac{2\pi i}{3}}\right)$$

See Weisstein, Eric W. "Airy Functions." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/AiryFunctions.html

and Airy Zeta function.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

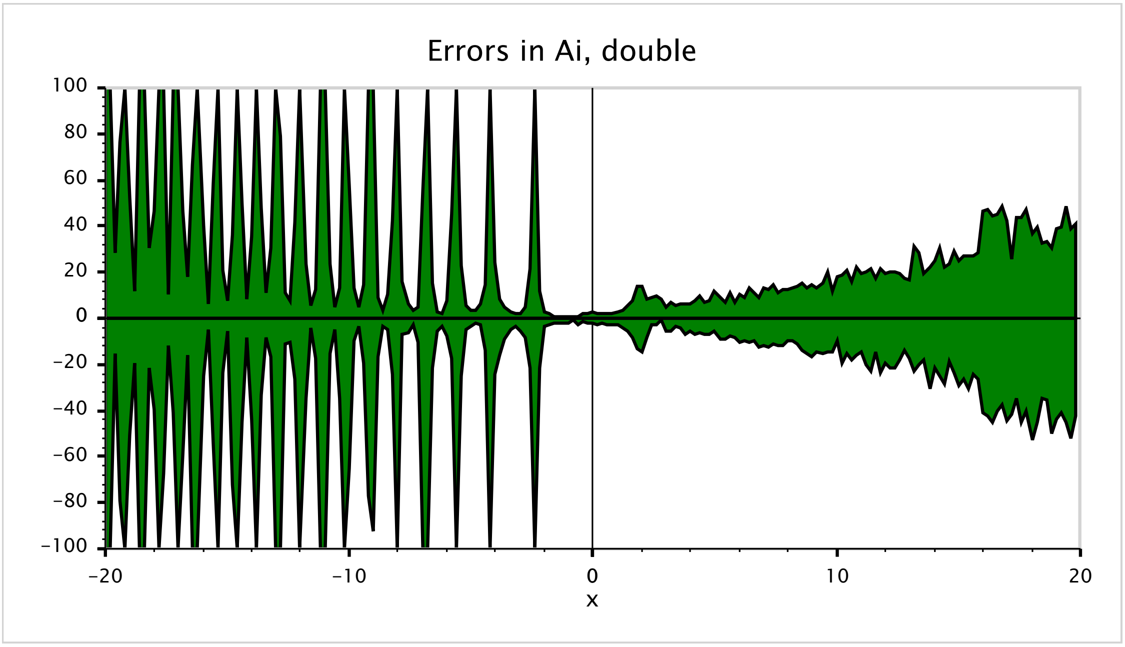
The following graph illustrates how this function changes as *x* changes: for negative *x* the function is cyclic, while for positive *x* the value tends to zero:



Accuracy

This function is implemented entirely in terms of the Bessel functions cyl_bessel_j and cyl_bessel_k - refer to those functions for detailed accuracy information.

In general though, the relative error is low (less than 100 ε) for $x > 0$ while only the absolute error is low for $x < 0$ as the following error plot illustrates:



Testing

Since this function is implemented in terms of other special functions, there are only a few basic sanity checks, using test values from Wolfram Airy Functions.

Implementation

This function is implemented in terms of the Bessel functions using the relations:

$$
\begin{aligned}
Ai(0) &= \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)} \\
Ai(z) &= \pi^{-1}\sqrt{\tfrac{z}{3}}K_{\pm\frac{1}{3}}(\zeta) \qquad ; \quad \zeta = \tfrac{2}{3}z^{\frac{2}{3}} \\
Ai(-z) &= \left(\tfrac{\sqrt{z}}{3}\right)\left(J_{\frac{1}{3}}(\zeta) + J_{-\frac{1}{3}}(\zeta)\right)
\end{aligned}
$$

Airy Bi Function

Synopsis

```
#include <boost/math/special_functions/airy.hpp>
```

```
namespace boost { namespace math {

 template <class T>
 calculated-result-type airy_bi(T x);

 template <class T, class Policy>
 calculated-result-type airy_bi(T x, const Policy&);

}} // namespaces
```

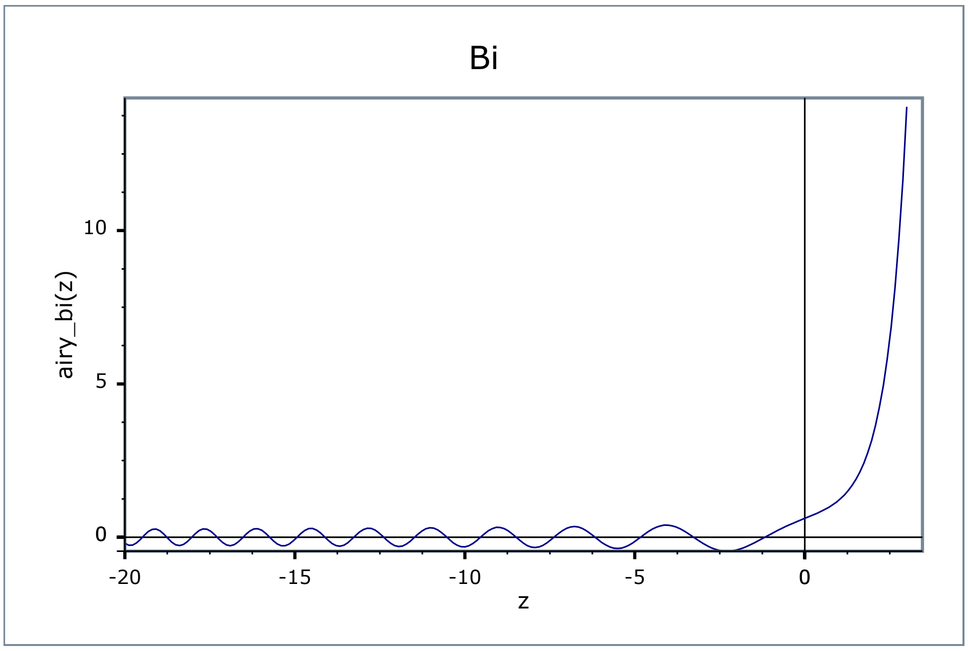
Description

The function airy_bi calculates the Airy function Bi which is the second solution to the differential equation:

$$\frac{d^2w}{dz^2} = zw \quad ; \quad w = Ai(z),\ Bi(z),\ Ai\left(ze^{\pm\frac{2\pi i}{3}}\right)$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

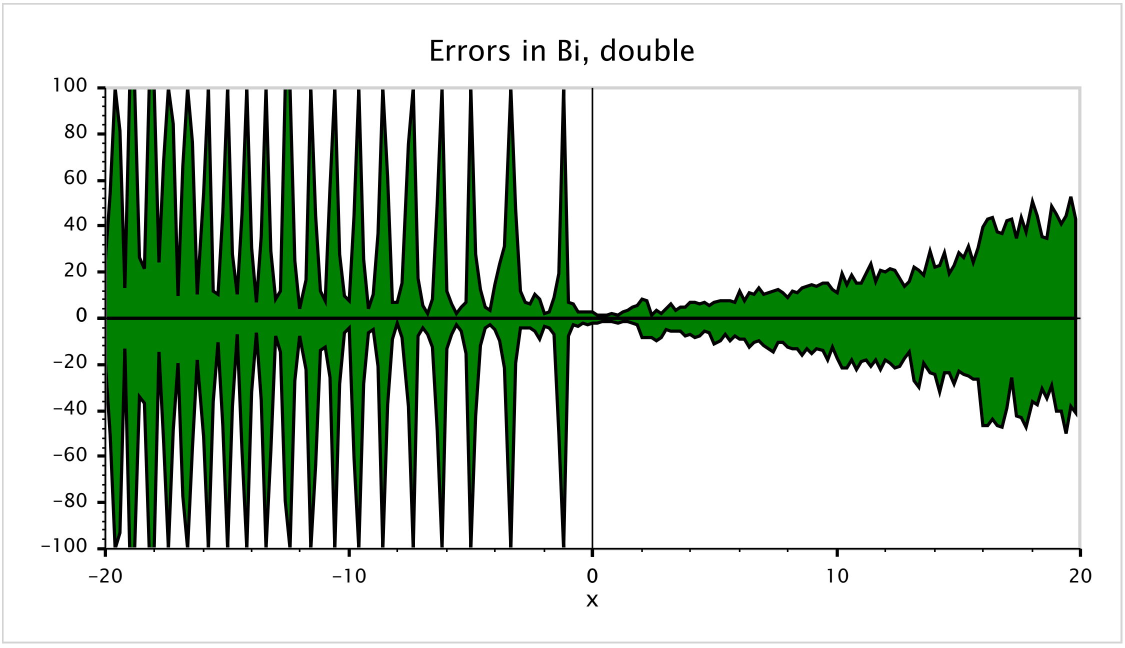
The following graph illustrates how this function changes as *x* changes: for negative *x* the function is cyclic, while for positive *x* the value tends to infinity:



Accuracy

This function is implemented entirely in terms of the Bessel functions cyl_bessel_i and cyl_bessel_j - refer to those functions for detailed accuracy information.

In general though, the relative error is low (less than 100 ε) for *x > 0* while only the absolute error is low for *x < 0* as the following error plot illustrate:



Testing

Since this function is implemented in terms of other special functions, there are only a few basic sanity checks, using test values from functions.wolfram.com.

Implementation

This function is implemented in terms of the Bessel functions using the relations:

$$
\begin{aligned}
Bi(0) &= \frac{1}{3^{\frac{1}{6}}\Gamma\left(\frac{2}{3}\right)} \\
Bi(z) &= \sqrt{\frac{z}{3}}\left(I_{\frac{1}{3}}(\zeta) + I_{-\frac{1}{3}}(\zeta)\right) \quad ; \quad \zeta = \frac{2}{3}z^{\frac{2}{3}} \\
Bi(-z) &= \sqrt{\frac{z}{3}}\left(J_{-\frac{1}{3}}(\zeta) - J_{\frac{1}{3}}(\zeta)\right)
\end{aligned}
$$

Airy Ai' Function

Synopsis

```
#include <boost/math/special_functions/airy.hpp>
```

```
namespace boost { namespace math {

 template <class T>
 calculated-result-type airy_ai_prime(T x);

 template <class T, class Policy>
 calculated-result-type airy_ai_prime(T x, const Policy&);

}} // namespaces
```

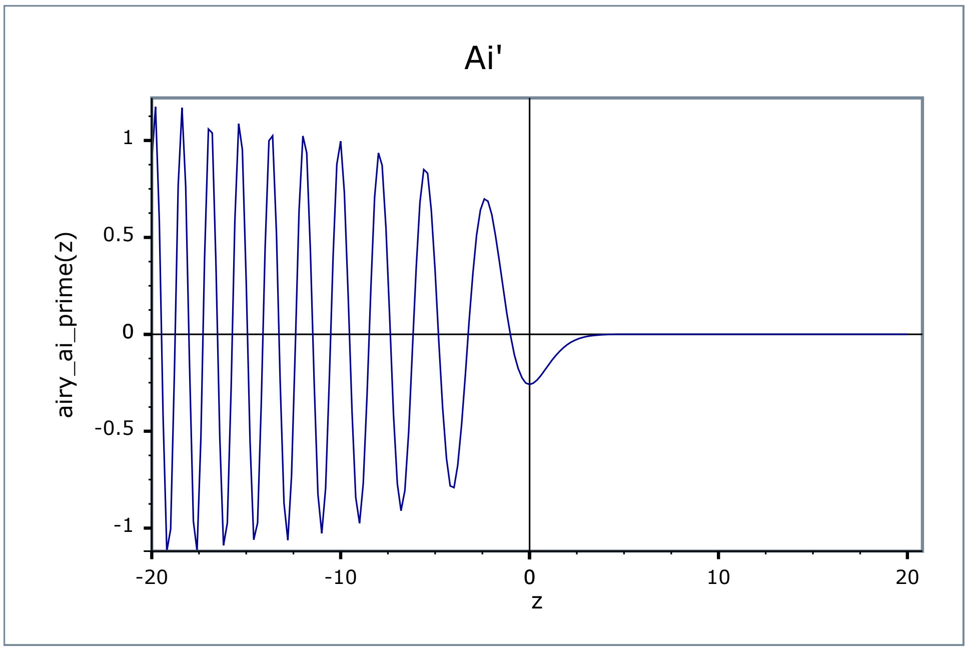
Description

The function airy_ai_prime calculates the Airy function Ai' which is the derivative of the first solution to the differential equation:

$$
\frac{d^2w}{dz^2} = zw \quad ; \quad w = Ai(z),\ Bi(z),\ Ai\left(ze^{\pm\frac{2\pi i}{3}}\right)
$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

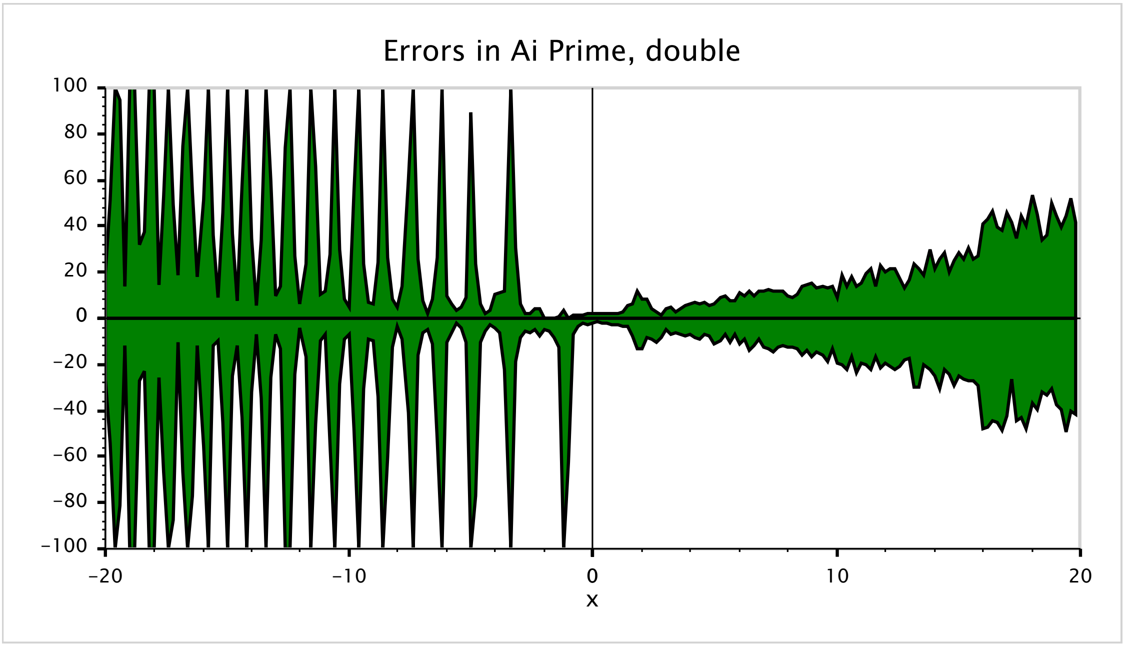
The following graph illustrates how this function changes as *x* changes: for negative *x* the function is cyclic, while for positive *x* the value tends to zero:



Accuracy

This function is implemented entirely in terms of the Bessel functions cyl_bessel_j and cyl_bessel_k - refer to those functions for detailed accuracy information.

In general though, the relative error is low (less than 100 ε) for *x > 0* while only the absolute error is low for *x < 0* as the following error plot illustrates:



Testing

Since this function is implemented in terms of other special functions, there are only a few basic sanity checks, using test values from functions.wolfram.com.

Implementation

This function is implemented in terms of the Bessel functions using the relations:

$$
\begin{aligned}
Ai'(0) &= -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)} \\
Ai'(z) &= \frac{z}{\pi\sqrt{3}}K_{\pm\frac{2}{3}}(\zeta) \qquad ; \quad \zeta = \frac{2}{3}z^{\frac{2}{3}} \\
Ai'(-z) &= \frac{z}{3}\left(J_{\frac{2}{3}}(\zeta) - J_{-\frac{2}{3}}(\zeta)\right)
\end{aligned}
$$

Airy Bi' Function

Synopsis

```
#include <boost/math/special_functions/airy.hpp>
```

```
namespace boost { namespace math {

 template <class T>
 calculated-result-type airy_bi_prime(T x);

 template <class T, class Policy>
 calculated-result-type airy_bi_prime(T x, const Policy&);

}} // namespaces
```

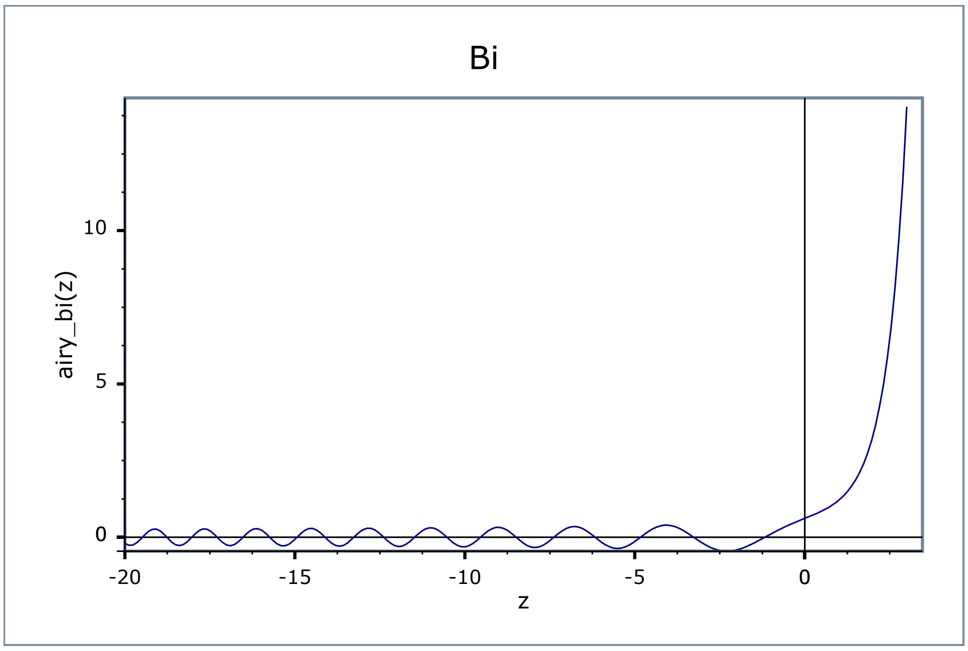
Description

The function airy_bi_prime calculates the Airy function Bi' which is the derivative of the second solution to the differential equation:

$$\frac{d^2w}{dz^2} = zw \quad ; \quad w = Ai(z),\ Bi(z),\ Ai\left(ze^{\pm\frac{2\pi i}{3}}\right)$$

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

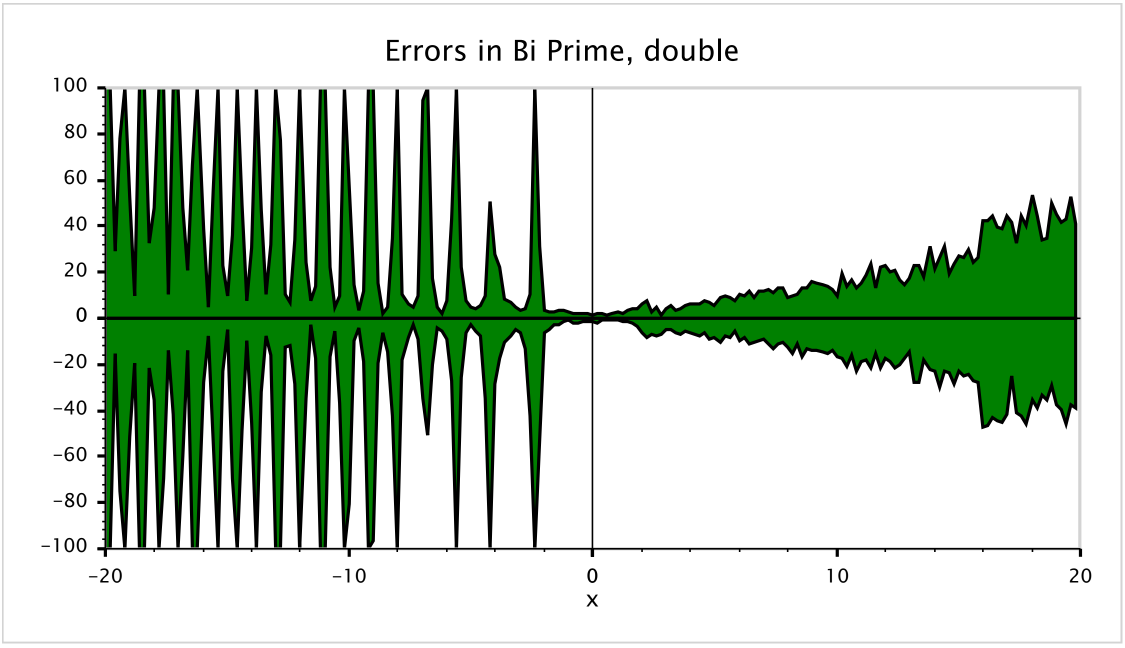
The following graph illustrates how this function changes as *x* changes: for negative *x* the function is cyclic, while for positive *x* the value tends to infinity:



Accuracy

This function is implemented entirely in terms of the Bessel functions cyl_bessel_i and cyl_bessel_j - refer to those functions for detailed accuracy information.

In general though, the relative error is low (less than 100 ε) for *x > 0* while only the absolute error is low for *x < 0* as the following error plot illustrates:



Testing

Since this function is implemented in terms of other special functions, there are only a few basic sanity checks, using test values from functions.wolfram.com.

Implementation

This function is implemented in terms of the Bessel functions using the relations:

$$Bi'(0) = \frac{3^{\frac{1}{6}}}{\Gamma\left(\frac{1}{3}\right)}$$

$$Bi'(z) = \frac{z}{\sqrt{3}}\left(I_{\frac{2}{3}}(\zeta) + I_{-\frac{2}{3}}(\zeta)\right) \quad ; \quad \zeta = \frac{2}{3}z^{\frac{2}{3}}$$

$$Bi'(-z) = \frac{z}{\sqrt{3}}\left(J_{-\frac{2}{3}}(\zeta) + J_{\frac{2}{3}}(\zeta)\right)$$

Finding Zeros of Airy Functions

Synopsis

```
#include <boost/math/special_functions/airy.hpp>
```

Functions for obtaining both a single zero or root of the Airy functions, and placing multiple zeros into a container like `std::vector` by providing an output iterator.

The signature of the single value functions are:

```
template <class T>
T airy_ai_zero(
         int m);         // 1-based index of zero.

template <class T>
T airy_bi_zero(
         int m);         // 1-based index of zero.
```

and for multiple zeros:

```
template <class T, class OutputIterator>
OutputIterator airy_ai_zero(
                     int start_index,           // 1-based index of first zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it);    // Destination for zeros.

template <class T, class OutputIterator>
OutputIterator airy_bi_zero(
                     int start_index,           // 1-based index of zero.
                     unsigned number_of_zeros,  // How many zeros to generate
                     OutputIterator out_it);    // Destination for zeros.
```

There are also versions which allow control of the Policies for error handling and precision.

```
 template <class T>
 T airy_ai_zero(
          int m,           // 1-based index of zero.
          const Policy&); // Policy to use.

 template <class T>
 T airy_bi_zero(
          int m,           // 1-based index of zero.
          const Policy&); // Policy to use.


template <class T, class OutputIterator>
OutputIterator airy_ai_zero(
                     int start_index,           // 1-based index of first zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it,     // Destination for zeros.
                     const Policy& pol);        // Policy to use.

template <class T, class OutputIterator>
OutputIterator airy_bi_zero(
                     int start_index,           // 1-based index of zero.
                     unsigned number_of_zeros,  // How many zeros to generate.
                     OutputIterator out_it,     // Destination for zeros.
                     const Policy& pol);        // Policy to use.
```

Description

The Airy Ai and Bi functions have an infinite number of zeros on the negative real axis. The real zeros on the negative real axis can be found by solving for the roots of

$$Ai(x_m) = 0$$

$$Bi(y_m) = 0$$

Here, x_m represents the m^{th} root of the Airy Ai function, and y_m represents the m^{th} root of the Airy Bi function.

The zeros or roots (values of `x` where the function crosses the horizontal `y = 0` axis) of the Airy Ai and Bi functions are computed by two functions, `airy_ai_zero` and `airy_bi_zero`.

In each case the index or rank of the zero returned is 1-based, which is to say:

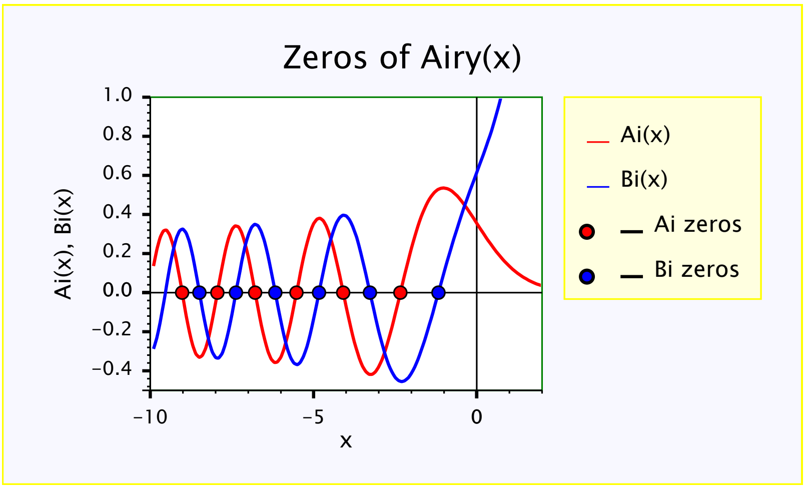
```
airy_ai_zero(1);
```

returns the first zero of Ai.

Passing an `start_index <= 0` results in a domain_error being raised.

The first few zeros returned by these functions have approximate values as follows:

m	Ai	Bi
1	-2.33811...	-1.17371...
2	-4.08795...	-3.27109...
3	-5.52056...	-4.83074...
4	-6.78671...	-6.16985...
5	-7.94413...	-7.37676...
6	-9.02265...	-8.49195...



Examples of finding Airy Zeros

This example demonstrates calculating zeros of the Airy functions. It also shows how Boost.Math and Boost.Multiprecision can be combined to provide a many decimal digit precision. For 50 decimal digit precision we need to include

```
#include <boost/multiprecision/cpp_dec_float.hpp>
```

and a `typedef` for `float_type` may be convenient (allowing a quick switch to re-compute at built-in `double` or other precision)

```
typedef boost::multiprecision::cpp_dec_float_50 float_type;
```

To use the functions for finding zeros of the functions we need

```
#include <boost/math/special_functions/airy.hpp>
```

This example shows obtaining both a single zero of the Airy functions, and then placing multiple zeros into a container like `std::vector` by providing an iterator. The signature of the single-value Airy Ai function is:

```
template <class T>
T airy_ai_zero(unsigned m); // 1-based index of the zero.
```

The signature of multiple zeros Airy Ai function is:

```
template <class T, class OutputIterator>
OutputIterator airy_ai_zero(
                         unsigned start_index, // 1-based index of the zero.
                         unsigned number_of_zeros, // How many zeros to generate.
                         OutputIterator out_it); // Destination for zeros.
```

There are also versions which allows control of the Policies for error handling and precision.

```
template <class T, class OutputIterator, class Policy>
OutputIterator airy_ai_zero(
                         unsigned start_index, // 1-based index of the zero.
                         unsigned number_of_zeros, // How many zeros to generate.
                         OutputIterator out_it, // Destination for zeros.
                         const Policy& pol);  // Policy to use.
```

Tip

It is always wise to place code using Boost.Math inside `try'n'catch` blocks; this will ensure that helpful error messages are shown when exceptional conditions arise.

First, evaluate a single Airy zero.

The precision is controlled by the template parameter `T`, so this example has `double` precision, at least 15 but up to 17 decimal digits (for the common 64-bit double).

```
double aiz1 = boost::math::airy_ai_zero<double>(1);
std::cout << "boost::math::airy_ai_zero<double>(1) = " << aiz1 << std::endl;
double aiz2 = boost::math::airy_ai_zero<double>(2);
std::cout << "boost::math::airy_ai_zero<double>(2) = " << aiz2 << std::endl;
double biz3 = boost::math::airy_bi_zero<double>(3);
std::cout << "boost::math::airy_bi_zero<double>(3) = " << biz3 << std::endl;
```

Other versions of `airy_ai_zero` and `airy_bi_zero` allow calculation of multiple zeros with one call, placing the results in a container, often `std::vector`. For example, generate and display the first five `double` roots Wolfram Airy Functions Zeros.

```
unsigned int n_roots = 5U;
std::vector<double> roots;
boost::math::airy_ai_zero<double>(1U, n_roots, std::back_inserter(roots));
std::cout << "airy_ai_zeros:" << std::endl;
std::copy(roots.begin(),
          roots.end(),
          std::ostream_iterator<double>(std::cout, "\n"));
```

The first few real roots of Ai(x) are approximately -2.33811, -4.08795, -5.52056, -6.7867144, -7.94413, -9.02265 ...

Or we can use Boost.Multiprecision to generate 50 decimal digit roots.

We set the precision of the output stream, and show trailing zeros to display a fixed 50 decimal digits.

```
std::cout.precision(std::numeric_limits<float_type>::digits10); // float_type has 50 decimal ↲
digits.
std::cout << std::showpoint << std::endl; // Show trailing zeros too.

unsigned int m = 1U;
float_type r = boost::math::airy_ai_zero<float_type>(1U); // 1st root.
std::cout << "boost::math::airy_bi_zero<float_type>(" << m << ")  = " << r << std::endl;
m = 2;
r = boost::math::airy_ai_zero<float_type>(2U); // 2nd root.
std::cout << "boost::math::airy_bi_zero<float_type>(" << m << ")  = " << r << std::endl;
m = 7U;
r = boost::math::airy_bi_zero<float_type>(7U); // 7th root.
std::cout << "boost::math::airy_bi_zero<float_type>(" << m << ")  = " << r << std::endl;

std::vector<float_type> zeros;
boost::math::airy_ai_zero<float_type>(1U, 3, std::back_inserter(zeros));
std::cout << "airy_ai_zeros:" << std::endl;
// Print the roots to the output stream.
std::copy(zeros.begin(), zeros.end(),
          std::ostream_iterator<float_type>(std::cout, "\n"));
```

Produces the program output:

```
boost::math::airy_ai_zero<double>(1) = -2.33811
boost::math::airy_ai_zero<double>(2) = -4.08795
boost::math::airy_bi_zero<double>(3) = -4.83074
airy_ai_zeros:
-2.33811
-4.08795
-5.52056
-6.78671
-7.94413

boost::math::airy_bi_zero<float_type>(1)  = -2.3381074104597670384891972524467354406385401456711
boost::math::airy_bi_zero<float_type>(2)  = -4.0879494441309706166369887014573910602247646991085
boost::math::airy_bi_zero<float_type>(7)  = -9.5381943793462388866329885451560196208390720763825
airy_ai_zeros:
-2.3381074104597670384891972524467354406385401456711
-4.0879494441309706166369887014573910602247646991085
-5.5205598280955510591298555129312935737972142806175
```

The full code (and output) for this example is at airy_zeros_example.cpp,

Implementation

Given the following function (A&S 10.4.105):

$$f(z) = z^{\frac{2}{3}}\left(1 + \frac{5}{48z^2} - \frac{5}{36z^4} + \frac{77125}{82944z^6} - \frac{108056875}{6967296z^8} + \frac{162375596875}{334430208z^{10}}\right)$$

Then an initial estimate for the nth zero a$_n$ of Ai is given by (A&S 10.4.94):

$$a_n = -f\left(3\pi\frac{(4n-1)}{8}\right)$$

and an initial estimate for the nth zero b$_n$ of Bi is given by (A&S 10.4.98):

$$b_n = -f\left(3\pi\frac{(4n-3)}{8}\right)$$

Thereafter the roots are refined using Newton iteration.

Testing

The precision of evaluation of zeros was tested at 50 decimal digits using `cpp_dec_float_50` and found identical with spot values computed by Wolfram Alpha.

Elliptic Integrals

Elliptic Integral Overview

The main reference for the elliptic integrals is:

> M. Abramowitz and I. A. Stegun (Eds.) (1964) Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, National Bureau of Standards Applied Mathematics Series, U.S. Government Printing Office, Washington, D.C.

and its recently revised version NIST Digital Library of Mathematical Functions (DMLF), in particular Elliptic Integrals, B. C. Carlson

Mathworld also contain a lot of useful background information:

> Weisstein, Eric W. "Elliptic Integral." From MathWorld--A Wolfram Web Resource.

As does Wikipedia Elliptic integral.

Notation

All variables are real numbers unless otherwise noted.

Definition

$$\int R(t, s)dt$$

is called elliptic integral if *R(t, s)* is a rational function of *t* and *s*, and s^2 is a cubic or quartic polynomial in *t*.

Elliptic integrals generally cannot be expressed in terms of elementary functions. However, Legendre showed that all elliptic integrals can be reduced to the following three canonical forms:

Elliptic Integral of the First Kind (Legendre form)

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

Elliptic Integral of the Second Kind (Legendre form)

$$E(\varphi, k) = \int_0^{\varphi} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$$

Elliptic Integral of the Third Kind (Legendre form)

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

where

$$k = \sin\alpha, \quad |k| \le 1$$

Note

ϕ is called the amplitude.

k is called the elliptic modulus or eccentricity.

α is called the modular angle.

n is called the characteristic.

Caution

Perhaps more than any other special functions the elliptic integrals are expressed in a variety of different ways. In particular, the final parameter *k* (the modulus) may be expressed using a modular angle α, or a parameter *m*. These are related by:

$k = \sin \alpha$

$m = k^2 = \sin^2\alpha$

So that the integral of the third kind (for example) may be expressed as either:

$\Pi(n, \phi, k)$

$\Pi(n, \phi \setminus \alpha)$

$\Pi(n, \phi \mid m)$

To further complicate matters, some texts refer to the *complement of the parameter m*, or 1 - m, where:

$1 - m = 1 - k^2 = \cos^2\alpha$

This implementation uses *k* throughout: this matches the requirements of the Technical Report on C++ Library Extensions.

So you should be extra careful when using these functions!

Warning

Boost.Math order of arguments differs from other implementations: *k* is always the **first** argument.

A simple example comparing use of Wolfram Alpha with Boost.Math (including much higher precision using Boost.Multiprecision) is jacobi_zeta_example.cpp.

When $\phi = \pi / 2$, the elliptic integrals are called *complete*.

Complete Elliptic Integral of the First Kind (Legendre form)

$$K(k) = F\left(\frac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}$$

Complete Elliptic Integral of the Second Kind (Legendre form)

$$E(k) = E\left(\frac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$$

Complete Elliptic Integral of the Third Kind (Legendre form)

$$\prod(n, k) = \prod\left(n, \tfrac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - k^2\sin^2\theta}}$$

Legendre also defined a fourth integral D(ϕ,k) which is a combination of the other three:

$$\begin{aligned} D(\varphi, k) &= \int_0^{\varphi} \frac{\sin^2\theta}{\sqrt{(1 - k^2\sin^2\theta)}} d\theta \\ &= \int_0^{\sin\varphi} \frac{t^2}{\sqrt{1 - t^2}\sqrt{1 - k^2t^2}} dt \\ &= \frac{(F(\varphi, k) - E(\varphi, k))}{k^2} \\ &= \tfrac{1}{3} R_D\left(c - 1, c - k^2, c\right) \quad ; \quad c = \frac{1}{\sin^2\varphi} \end{aligned}$$

Like the other Legendre integrals this comes in both complete and incomplete forms.

Carlson Elliptic Integrals

Carlson [Carlson77] [Carlson78] gives an alternative definition of elliptic integral's canonical forms:

Carlson's Elliptic Integral of the First Kind

$$R_F(x, y, z) = \frac{1}{2}\int_0^{\infty} [(t + x)(t + y)(t + z)]^{-\frac{1}{2}} dt$$

where *x*, *y*, *z* are nonnegative and at most one of them may be zero.

Carlson's Elliptic Integral of the Second Kind

$$R_D(x, y, z) = \frac{3}{2}\int_0^{\infty} [(t + x)(t + y)]^{-\frac{1}{2}} (t + z)^{-\frac{3}{2}} dt$$

where *x*, *y* are nonnegative, at most one of them may be zero, and *z* must be positive.

Carlson's Elliptic Integral of the Third Kind

$$R_J(x, y, z, p) = \frac{3}{2}\int_0^{\infty} (t + p)^{-1} [(t + x)(t + y)(t + z)]^{-\frac{1}{2}} dt$$

where *x*, *y*, *z* are nonnegative, at most one of them may be zero, and *p* must be nonzero.

Carlson's Degenerate Elliptic Integral

$$R_C(x, y) = \frac{1}{2}\int_0^{\infty} (t + x)^{-\frac{1}{2}} (t + y)^{-1} dt$$

where *x* is nonnegative and *y* is nonzero.

Note

$R_C(x, y) = R_F(x, y, y)$

$R_D(x, y, z) = R_J(x, y, z, z)$

Carlson's Symmetric Integral

$$R_G(x,\ y,\ z) = \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi}\sqrt{\left(x\sin^2\theta\cos^2\varphi + y\sin^2\theta\sin^2\varphi + z\cos^2\theta\right)}\,\sin\theta d\,\theta d\,\varphi$$

Duplication Theorem

Carlson proved in [Carlson78] that

$$\begin{aligned}
R_F(x,\ y,\ z) &= 2R_F(x+\lambda,\ y+\lambda,\ z+\lambda) \\
&= R_F\left(\tfrac{x+\lambda}{4},\ \tfrac{y+\lambda}{4},\ \tfrac{z+\lambda}{4}\right) \\
\lambda &= \sqrt{xy} + \sqrt{yz} + \sqrt{zx}
\end{aligned}$$

Carlson's Formulas

The Legendre form and Carlson form of elliptic integrals are related by equations:

$$\begin{aligned}
F(\varphi,\ k) &= \sin\varphi R_F\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1\right) \\
E(\varphi,\ k) &= \sin\varphi R_F\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1\right) - \tfrac{1}{3}k^2\sin^3\varphi R_D\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1\right) \\
\Pi(n,\ \varphi,\ k) &= \sin\varphi R_F\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1\right) + \tfrac{1}{3}n\sin^3\varphi R_J\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1,\ 1-n\sin^2\varphi\right)
\end{aligned}$$

In particular,

$$\begin{aligned}
K(k) &= R_F\left(0,\ 1-k^2,\ 1\right) \\
E(k) &= R_F\left(0,\ 1-k^2,\ 1\right) - \tfrac{1}{3}k^2R_D\left(0,\ 1-k^2,\ 1\right) \\
\Pi(n,\ k) &= R_F\left(0,\ 1-k^2,\ 1\right) + \tfrac{1}{3}nR_J\left(0,\ 1-k^2,\ 1,\ 1-n\right)
\end{aligned}$$

Miscellaneous Elliptic Integrals

There are two functions related to the elliptic integrals which otherwise defy categorisation, these are the Jacobi Zeta function:

$$\begin{aligned}
Z(\varphi,\ k) &= E(\varphi,\ k) - \frac{E(k)F(\varphi,\ k)}{K(k)} \\
&= \frac{k^2}{3K(k)}\sin\varphi\cos\varphi\sqrt{\left(1-k^2\sin^2\varphi\right)}R_J\left(0,1-k^2,\ 1,1-k^2\sin^2\varphi\right)
\end{aligned}$$

and the Heuman Lambda function:

$$\begin{aligned}
\Lambda_0(\varphi,\ k) &= \frac{F\left(\varphi,\ \sqrt{1-k^2}\right)}{K\left(\sqrt{1-k^2}\right)} + \tfrac{2}{\pi}K(k)Z\left(\varphi,\ \sqrt{1-k^2}\right) \\
&= \frac{2}{\pi}\frac{\left(1-k^2\right)\sin\varphi\cos\varphi}{\Delta}\left(R_F\left(0,1-k^2,\ 1\right) + \frac{k^2}{3\Delta^2}R_J\left(0,1-k^2,\ 1,1-\frac{k^2}{\Delta^2}\right)\right) \quad ;\Delta^2 =
\end{aligned}$$

Both of these functions are easily implemented in terms of Carlson's integrals, and are provided in this library as jacobi_zeta and heuman_lambda.

Numerical Algorithms

The conventional methods for computing elliptic integrals are Gauss and Landen transformations, which converge quadratically and work well for elliptic integrals of the first and second kinds. Unfortunately they suffer from loss of significant digits for the third kind. Carlson's algorithm [Carlson79] [Carlson78], by contrast, provides a unified method for all three kinds of elliptic integrals with satisfactory precisions.

References

Special mention goes to:

A. M. Legendre, *Traité des Fonctions Elliptiques et des Integrales Euleriennes*, Vol. 1. Paris (1825).

However the main references are:

1. M. Abramowitz and I. A. Stegun (Eds.) (1964) Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, National Bureau of Standards Applied Mathematics Series, U.S. Government Printing Office, Washington, D.C.
2. NIST Digital Library of Mathematical Functions, Elliptic Integrals, B. C. Carlson
3. B.C. Carlson, *Computing elliptic integrals by duplication*, Numerische Mathematik, vol 33, 1 (1979).
4. B.C. Carlson, *Elliptic Integrals of the First Kind*, SIAM Journal on Mathematical Analysis, vol 8, 231 (1977).
5. B.C. Carlson, *Short Proofs of Three Theorems on Elliptic Integrals*, SIAM Journal on Mathematical Analysis, vol 9, 524 (1978).
6. B.C. Carlson and E.M. Notis, *ALGORITHM 577: Algorithms for Incomplete Elliptic Integrals*, ACM Transactions on Mathematmal Software, vol 7, 398 (1981).
7. B. C. Carlson, *On computing elliptic integrals and functions*. J. Math. and Phys., 44 (1965), pp. 36-51.
8. B. C. Carlson, *A table of elliptic integrals of the second kind*. Math. Comp., 49 (1987), pp. 595-606. (Supplement, ibid., pp. S13-S17.)
9. B. C. Carlson, *A table of elliptic integrals of the third kind*. Math. Comp., 51 (1988), pp. 267-280. (Supplement, ibid., pp. S1-S5.)
10. B. C. Carlson, *A table of elliptic integrals: cubic cases*. Math. Comp., 53 (1989), pp. 327-333.
11. B. C. Carlson, *A table of elliptic integrals: one quadratic factor*. Math. Comp., 56 (1991), pp. 267-280.
12. B. C. Carlson, *A table of elliptic integrals: two quadratic factors*. Math. Comp., 59 (1992), pp. 165-180.
13. B. C. Carlson, *Numerical computation of real or complex elliptic integrals*. Numerical Algorithms, Volume 10, Number 1 / March, 1995, p13-26.
14. B. C. Carlson and John L. Gustafson, *Asymptotic Approximations for Symmetric Elliptic Integrals*, SIAM Journal on Mathematical Analysis, Volume 25, Issue 2 (March 1994), 288-303.

The following references, while not directly relevent to our implementation, may also be of interest:

1. R. Burlisch, *Numerical Compuation of Elliptic Integrals and Elliptic Functions.* Numerical Mathematik 7, 78-90.
2. R. Burlisch, *An extension of the Bartky Transformation to Incomplete Elliptic Integrals of the Third Kind*. Numerical Mathematik 13, 266-284.
3. R. Burlisch, *Numerical Compuation of Elliptic Integrals and Elliptic Functions. III*. Numerical Mathematik 13, 305-315.

4. T. Fukushima and H. Ishizaki, *Numerical Computation of Incomplete Elliptic Integrals of a General Form.* Celestial Mechanics and Dynamical Astronomy, Volume 59, Number 3 / July, 1994, 237-251.

Elliptic Integrals - Carlson Form

Synopsis

```
#include <boost/math/special_functions/ellint_rf.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2, class T3>
calculated-result-type ellint_rf(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rf(T1 x, T2 y, T3 z, const Policy&)

}} // namespaces
```

```
#include <boost/math/special_functions/ellint_rd.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2, class T3>
calculated-result-type ellint_rd(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rd(T1 x, T2 y, T3 z, const Policy&)

}} // namespaces
```

```
#include <boost/math/special_functions/ellint_rj.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2, class T3, class T4>
calculated-result-type ellint_rj(T1 x, T2 y, T3 z, T4 p)

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ellint_rj(T1 x, T2 y, T3 z, T4 p, const Policy&)

}} // namespaces
```

```
#include <boost/math/special_functions/ellint_rc.hpp>
```

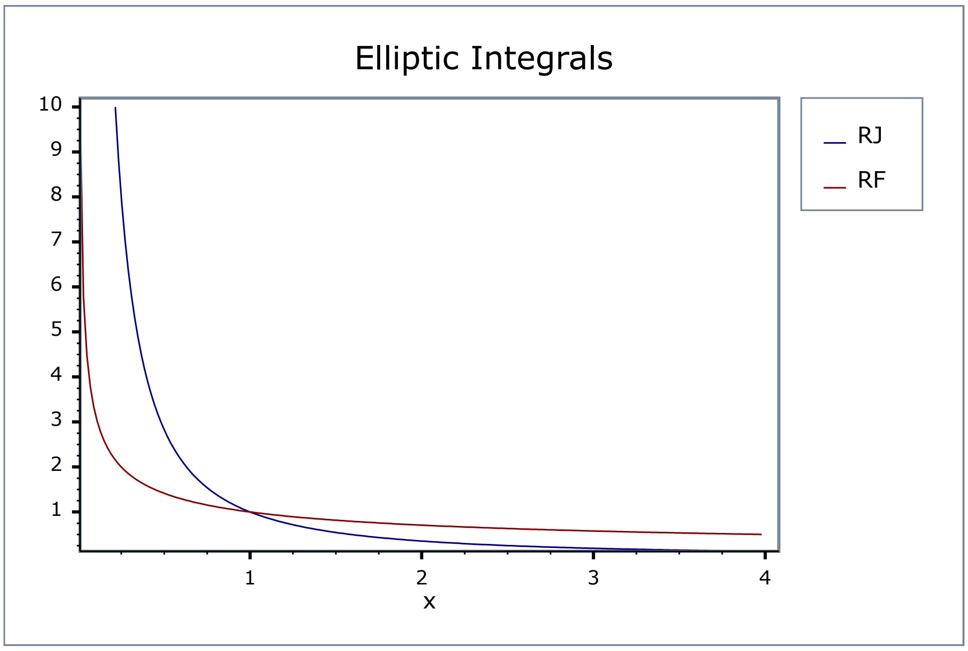
```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type ellint_rc(T1 x, T2 y)

template <class T1, class T2, class Policy>
calculated-result-type ellint_rc(T1 x, T2 y, const Policy&)

}} // namespaces
```

```
#include <boost/math/special_functions/ellint_rg.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2, class T3>
calculated-result-type ellint_rg(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rg(T1 x, T2 y, T3 z, const Policy&)

}} // namespaces
```

Description

These functions return Carlson's symmetrical elliptic integrals, the functions have complicated behavior over all their possible domains, but the following graph gives an idea of their behavior:



The return type of these functions is computed using the *result type calculation rules* when the arguments are of different types: otherwise the return is the same type as the arguments.

```
template <class T1, class T2, class T3>
calculated-result-type ellint_rf(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rf(T1 x, T2 y, T3 z, const Policy&)
```

Returns Carlson's Elliptic Integral R_F:

$$R_F(x, y, z) = \frac{1}{2}\int_0^\infty [(t+x)(t+y)(t+z)]^{-\frac{1}{2}}dt$$

Requires that all of the arguments are non-negative, and at most one may be zero. Otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3>
calculated-result-type ellint_rd(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rd(T1 x, T2 y, T3 z, const Policy&)
```

Returns Carlson's elliptic integral R_D:

$$R_D(x,\ y,\ z) = \frac{3}{2}\int_0^{\infty}[(t+x)(t+y)]^{-\frac{1}{2}}(t+z)^{-\frac{3}{2}}dt$$

Requires that x and y are non-negative, with at most one of them zero, and that z >= 0. Otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2, class T3, class T4>
calculated-result-type ellint_rj(T1 x, T2 y, T3 z, T4 p)

template <class T1, class T2, class T3, class T4, class Policy>
calculated-result-type ellint_rj(T1 x, T2 y, T3 z, T4 p, const Policy&)
```

Returns Carlson's elliptic integral R_J:

$$R_J(x,\ y,\ z,\ p) = \frac{3}{2}\int_0^{\infty}(t+p)^{-1}[(t+x)(t+y)(t+z)]^{-\frac{1}{2}}dt$$

Requires that x, y and z are non-negative, with at most one of them zero, and that *p != 0*. Otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

When *p < 0* the function returns the Cauchy principal value using the relation:

$$\begin{aligned}(y+q)R_J(x,\ y,\ z,\ -q) \quad &= \quad (p-y)R_J(x,\ y,\ z,\ p) - 3R_F(x,\ y,\ z) \\ &\qquad +3\left(\frac{xyz}{xz+pq}\right)^{\frac{1}{2}}R_C(xz+pq,\ pq) \\ \text{with: } q > 0 \qquad &\qquad \text{and: } p = y + \frac{(z-y)(y-x)}{(y+q)}\end{aligned}$$

```
template <class T1, class T2>
calculated-result-type ellint_rc(T1 x, T2 y)

template <class T1, class T2, class Policy>
calculated-result-type ellint_rc(T1 x, T2 y, const Policy&)
```

Returns Carlson's elliptic integral R_C:

$$R_C(x,\ y) = \frac{1}{2}\int_0^{\infty}(t+x)^{-\frac{1}{2}}(t+y)^{-1}dt$$

Requires that *x > 0* and that *y != 0*. Otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

When *y < 0* the function returns the Cauchy principal value using the relation:

$$R_C(x, -y) = \left(\frac{x}{x+y}\right)^{\frac{1}{2}} R_C(x+y, y)$$

```
template <class T1, class T2, class T3>
calculated-result-type ellint_rg(T1 x, T2 y, T3 z)

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_rg(T1 x, T2 y, T3 z, const Policy&)
```

Returns Carlson's elliptic integral R_G:

$$R_G(x, y, z) = \frac{1}{4\pi}\int_0^{2\pi}\int_0^{\pi}\sqrt{\left(x\sin^2\theta\cos^2\varphi + y\sin^2\theta\sin^2\varphi + z\cos^2\theta\right)}\sin\theta d\theta d\varphi$$

Requires that x and y are non-negative, otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Testing

There are two sets of tests.

Spot tests compare selected values with test data given in:

B. C. Carlson, *Numerical computation of real or complex elliptic integrals*. Numerical Algorithms, Volume 10, Number 1 / March, 1995, pp 13-26.

Random test data generated using NTL::RR at 1000-bit precision and our implementation checks for rounding-errors and/or regressions.

There are also sanity checks that use the inter-relations between the integrals to verify their correctness: see the above Carlson paper for details.

Accuracy

These functions are computed using only basic arithmetic operations, so there isn't much variation in accuracy over differing platforms. Note that only results for the widest floating-point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon.

Table 81. Error rates for ellint_rc

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RC: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.4ε (Mean = 0.624ε))	Max = 0.995ε (Mean = 0.433ε)	Max = 0.962ε (Mean = 0.407ε)

Table 82. Error rates for ellint_rd

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Microsoft Visual C++ version 14.1 Win32 double
RD: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.59ε (Mean = 0.878ε))	Max = 2.73ε (Mean = 0.831ε)	Max = 2.16ε (Mean = 0.803ε)
RD: y = z	Max = 0.896ε (Mean = 0.022ε) (*GSL 2.1:* Max = 2.88ε (Mean = 0.839ε))	Max = 2.65ε (Mean = 0.82ε)	Max = 16.5ε (Mean = 0.843ε)
RD: x = y	Max = 0.824ε (Mean = 0.0272ε) (*GSL 2.1:* Max = 3.74ε (Mean = 0.84ε))	Max = 2.85ε (Mean = 0.865ε)	Max = 3.51ε (Mean = 0.816ε)
RD: x = 0, y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2ε (Mean = 0.656ε))	Max = 1.19ε (Mean = 0.522ε)	Max = 1.16ε (Mean = 0.497ε)
RD: x = y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.03ε (Mean = 0.418ε))	Max = 0.998ε (Mean = 0.387ε)	Max = 1.03ε (Mean = 0.418ε)
RD: x = 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.85ε (Mean = 0.781ε))	Max = 2.79ε (Mean = 0.883ε)	Max = 2.64ε (Mean = 0.894ε)

Table 83. Error rates for ellint_rg

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RG: Random Data	Max = 0.983ε (Mean = 0.0172ε) (*GSL 2.1:* Max = 0.983ε (Mean = 0.0172ε))	Max = 3.95ε (Mean = 0.951ε)	Max = 3.65ε (Mean = 0.929ε)
RG: two values 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε)	Max = 0ε (Mean = 0ε)
RG: All values the same or zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0.992ε (Mean = 0.288ε)	Max = 1.06ε (Mean = 0.348ε)
RG: two values the same	Max = 0.594ε (Mean = 0.0103ε) (*GSL 2.1:* Max = 0.594ε (Mean = 0.0103ε))	Max = 1.51ε (Mean = 0.404ε)	Max = 1.96ε (Mean = 0.374ε)
RG: one value zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 2.14ε (Mean = 0.722ε)	Max = 1.96ε (Mean = 0.674ε)

Table 84. Error rates for ellint_rf

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RF: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.73ε (Mean = 0.804ε))	Max = 2.54ε (Mean = 0.674ε)	Max = 2.02ε (Mean = 0.677ε)
RF: x = y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.999ε (Mean = 0.34ε))	Max = 0.991ε (Mean = 0.345ε)	Max = 0.999ε (Mean = 0.34ε)
RF: x = y or y = z or x = z	Max = 0.536ε (Mean = 0.00658ε) (*GSL 2.1:* Max = 2.89ε (Mean = 0.749ε))	Max = 1.95ε (Mean = 0.418ε)	Max = 1.21ε (Mean = 0.394ε)
RF: x = 0, y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.29ε (Mean = 0.527ε))	Max = 0.894ε (Mean = 0.338ε)	Max = 0.999ε (Mean = 0.407ε)
RF: z = 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.54ε (Mean = 0.781ε))	Max = 1.7ε (Mean = 0.539ε)	Max = 1.89ε (Mean = 0.587ε)

Table 85. Error rates for ellint_rj

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Microsoft Visual C++ version 14.1 Win32 double
RJ: Random data	Max = 0.52ε (Mean = 0.0184ε) (*GSL 2.1:* Max = 3.57ε (Mean = 0.704ε) And other failures.)	Max = 186ε (Mean = 6.67ε)	Max = 215ε (Mean = 7.66ε)
RJ: 4 Equal Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.03ε (Mean = 0.418ε))	Max = 0.998ε (Mean = 0.387ε)	Max = 1.03ε (Mean = 0.418ε)
RJ: 3 Equal Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.96ε (Mean = 1.06ε))	Max = 20.8ε (Mean = 0.986ε)	Max = 39.9ε (Mean = 1.17ε)
RJ: 2 Equal Values	Max = 0.6ε (Mean = 0.0228ε) (*GSL 2.1:* Max = 2.57ε (Mean = 0.754ε))	Max = 220ε (Mean = 6.64ε)	Max = 214ε (Mean = 5.28ε)
RJ: Equal z and p	Max = 0.742ε (Mean = 0.0166ε) (*GSL 2.1:* Max = 2.62ε (Mean = 0.699ε))	Max = 17.2ε (Mean = 1.16ε)	Max = 16.1ε (Mean = 1.14ε)

Implementation

The key of Carlson's algorithm [Carlson79] is the duplication theorem:

$$\begin{aligned} R_F(x,\ y,\ z) &= 2R_F(x+\lambda,\ y+\lambda,\ z+\lambda) \\ &= R_F\left(\frac{x+\lambda}{4},\ \frac{y+\lambda}{4},\ \frac{z+\lambda}{4}\right) \\ \lambda &= \sqrt{xy}+\sqrt{yz}+\sqrt{zx} \end{aligned}$$

By applying it repeatedly, *x*, *y*, *z* get closer and closer. When they are nearly equal, the special case equation

$$R_F(x,\ x,\ x) = \frac{1}{\sqrt{x}}$$

is used. More specifically, *[R F]* is evaluated from a Taylor series expansion to the fifth order. The calculations of the other three integrals are analogous, except for R_C which can be computed from elementary functions.

For $p < 0$ in $R_J(x, y, z, p)$ and $y < 0$ in $R_C(x, y)$, the integrals are singular and their Cauchy principal values are returned via the relations:

$$\begin{aligned} (y+q)R_J(x,\ y,\ z,\ -q) \quad &= \quad (p-y)R_J(x,\ y,\ z,\ p) - 3R_F(x,\ y,\ z) \\ &\quad +3\left(\frac{xyz}{xz+pq}\right)^{\frac{1}{2}}R_C(xz+pq,\ pq) \\ \text{with: } q > 0 \quad & \quad \text{and: } p = y + \frac{(z-y)(y-x)}{(y+q)} \end{aligned}$$

$$R_C(x, -y) = \left(\frac{x}{x+y}\right)^{\frac{1}{2}} R_C(x+y, y)$$

Elliptic Integrals of the First Kind - Legendre Form

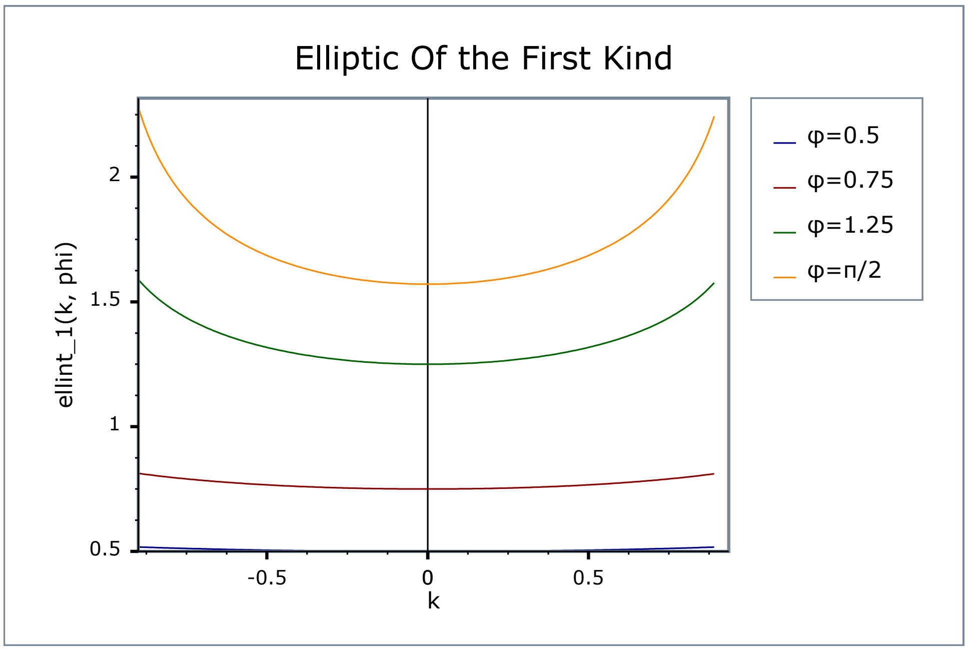
Synopsis

```
#include <boost/math/special_functions/ellint_1.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type ellint_1(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_1(T1 k, T2 phi, const Policy&);

template <class T>
calculated-result-type ellint_1(T k);

template <class T, class Policy>
calculated-result-type ellint_1(T k, const Policy&);

}} // namespaces
```

Description

These two functions evaluate the incomplete elliptic integral of the first kind *F(ϕ, k)* and its complete counterpart *K(k) = F(π/2, k)*.



The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types: when they are the same type then the result is the same type as the arguments.

```
template <class T1, class T2>
calculated-result-type ellint_1(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_1(T1 k, T2 phi, const Policy&);
```

Returns the incomplete elliptic integral of the first kind *F(ϕ, k)*:

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

Requires $k^2\sin^2(phi) < 1$, otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T>
calculated-result-type ellint_1(T k);

template <class T>
calculated-result-type ellint_1(T k, const Policy&);
```

Returns the complete elliptic integral of the first kind *K(k)*:

$$K(k) = F\left(\frac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}$$

Requires |k| < 1, otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

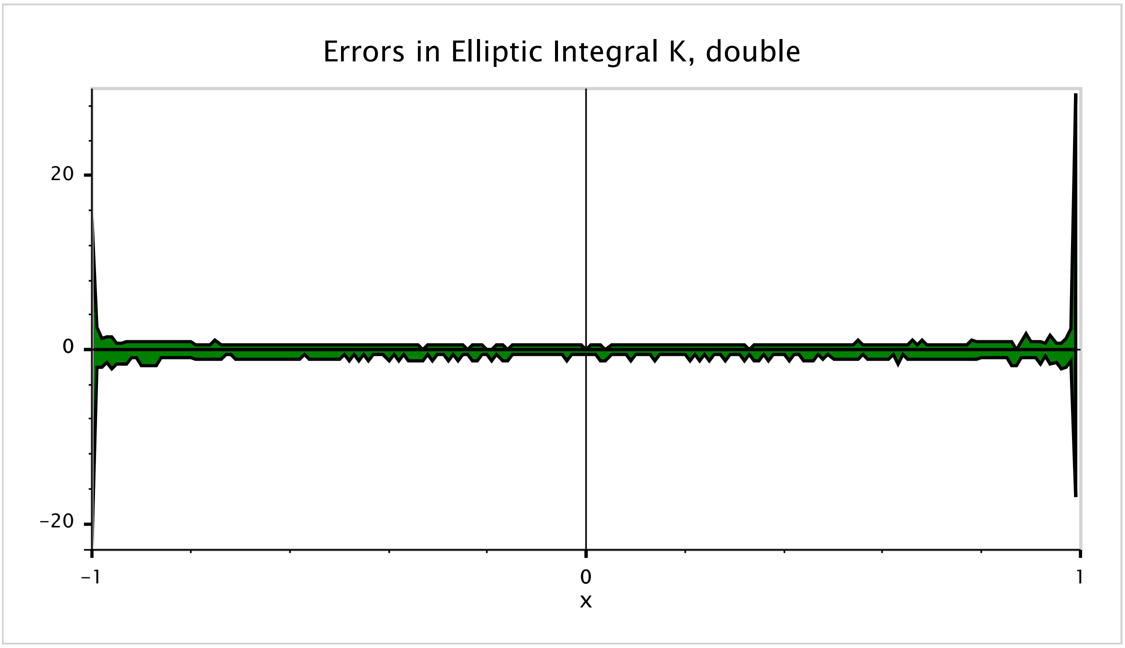
Accuracy

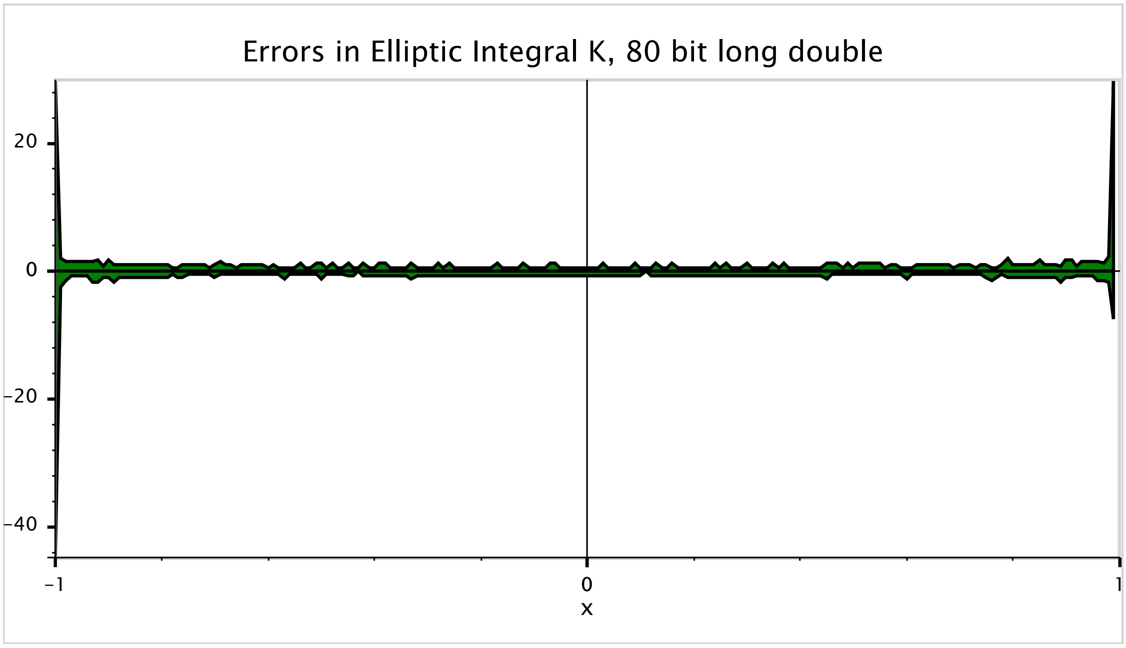
These functions are computed using only basic arithmetic operations, so there isn't much variation in accuracy over differing platforms. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon.

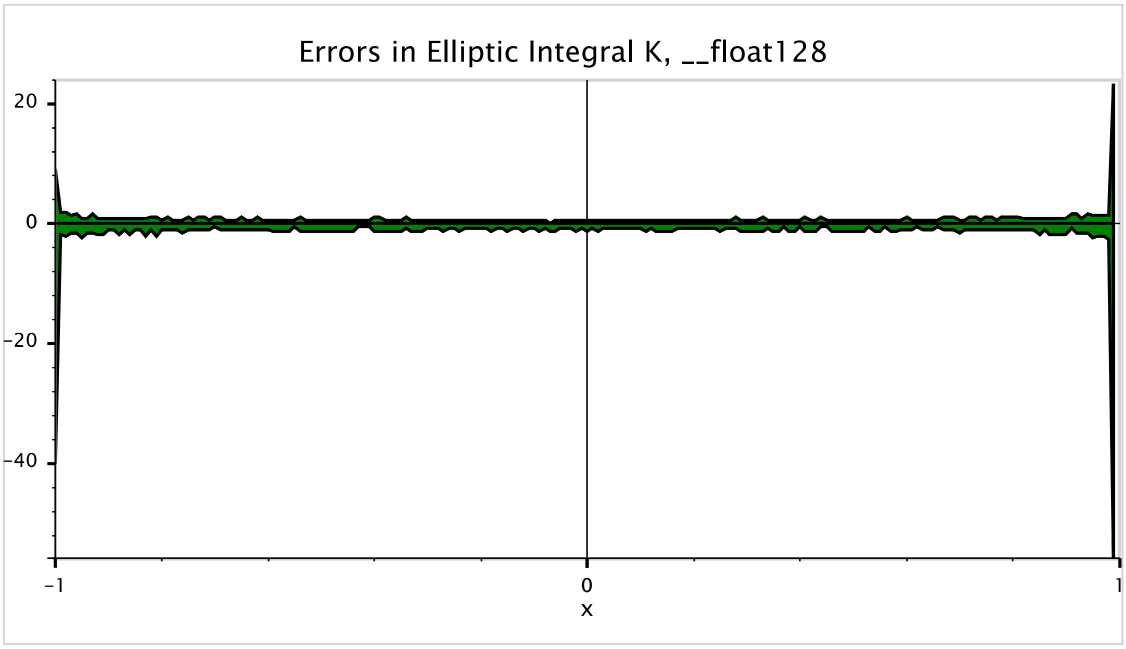
Table 86. Error rates for ellint_1

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral F: Mathworld Data	Max = 0.94ε (Mean = 0.509ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.919ε (Mean = 0.544ε))	Max = 0.94ε (Mean = 0.509ε)	Max = 0.919ε (Mean = 0.542ε)
Elliptic Integral F: Random Data	Max = 1.57ε (Mean = 0.56ε) (*<cmath>:* Max = 2.56ε (Mean = 0.816ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.99ε (Mean = 0.797ε))	Max = 1.57ε (Mean = 0.561ε)	Max = 2.26ε (Mean = 0.631ε)

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The tests use a mixture of spot test values calculated using the online calculator at functions.wolfram.com, and random test data generated using NTL::RR at 1000-bit precision and this implementation.

Implementation

These functions are implemented in terms of Carlson's integrals using the relations:

$$\begin{aligned} F(-\varphi, k) &= -F(\varphi, k) \\ F(\varphi + m\pi, k) &= F(\varphi, k) + 2mK(k) \\ F(\varphi, k) &= \sin\varphi R_F\left(\cos^2\varphi,\ 1 - k^2\sin^2\varphi,\ 1\right) \end{aligned}$$

and

$$K(k) \quad = \quad R_F\left(0, 1 - k^2,\ 1\right)$$

Elliptic Integrals of the Second Kind - Legendre Form

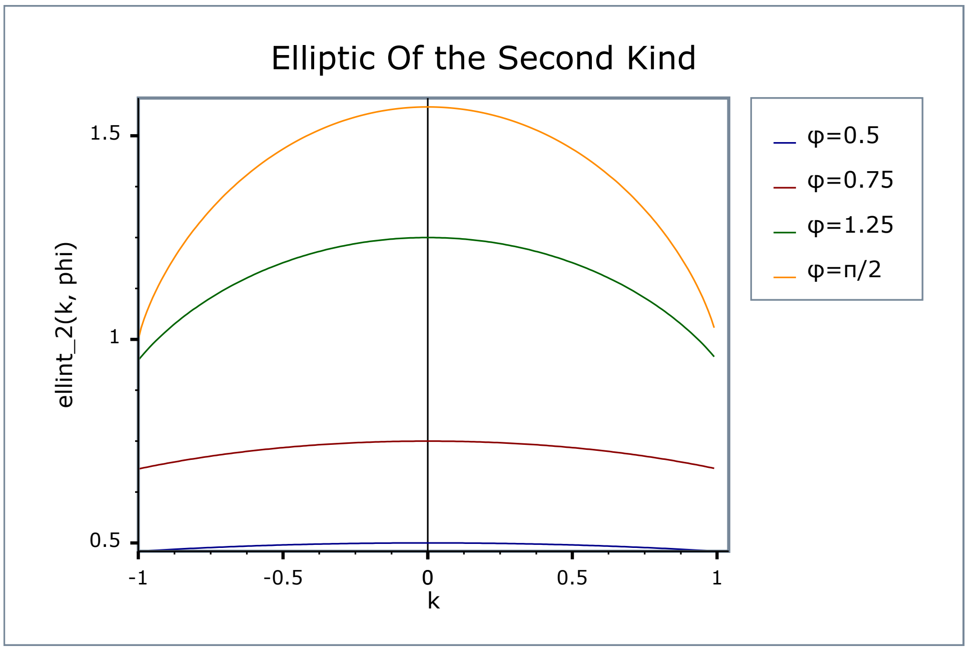
Synopsis

```
#include <boost/math/special_functions/ellint_2.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type ellint_2(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_2(T1 k, T2 phi, const Policy&);

template <class T>
calculated-result-type ellint_2(T k);

template <class T, class Policy>
calculated-result-type ellint_2(T k, const Policy&);

}} // namespaces
```

Description

These two functions evaluate the incomplete elliptic integral of the second kind *E(φ, k)* and its complete counterpart *E(k) = E(π/2, k)*.



The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types: when they are the same type then the result is the same type as the arguments.

```
template <class T1, class T2>
calculated-result-type ellint_2(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_2(T1 k, T2 phi, const Policy&);
```

Returns the incomplete elliptic integral of the second kind *E(ϕ, k)*:

$$E(\varphi,\, k) = \int_0^{\varphi} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$$

Requires $k^2\sin^2(phi) < 1$, otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T>
calculated-result-type ellint_2(T k);

template <class T>
calculated-result-type ellint_2(T k, const Policy&);
```

Returns the complete elliptic integral of the second kind *E(k)*:

$$E(k) = E\left(\frac{\pi}{2},\, k\right) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$$

Requires |k| < 1, otherwise returns the result of domain_error.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

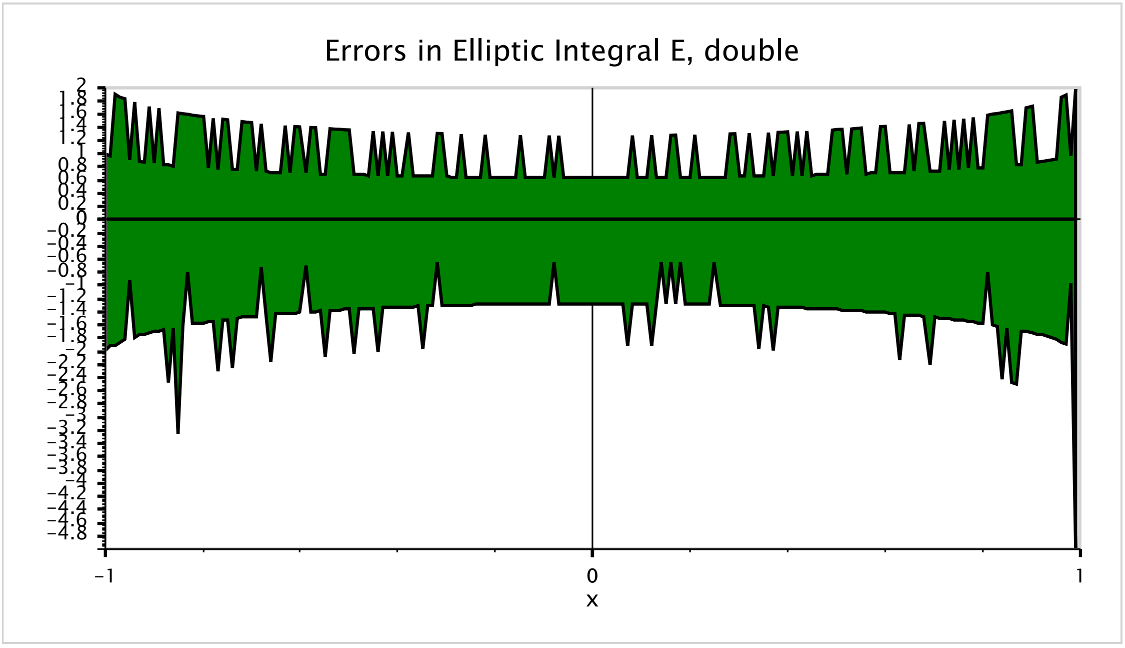
Accuracy

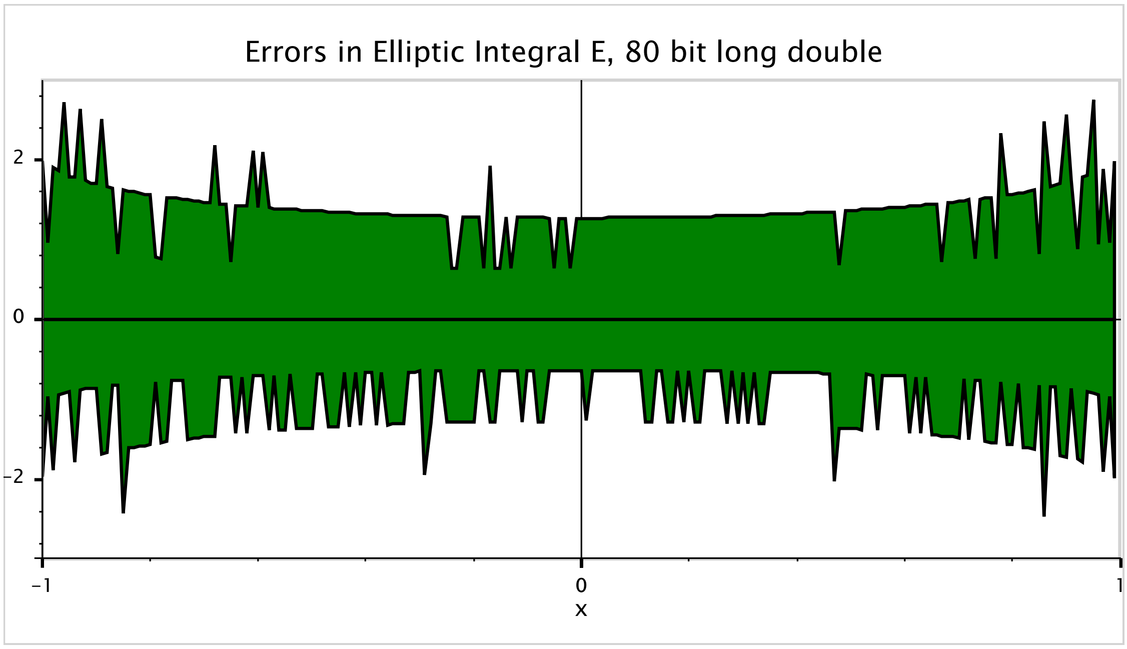
These functions are computed using only basic arithmetic operations, so there isn't much variation in accuracy over differing platforms. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon.

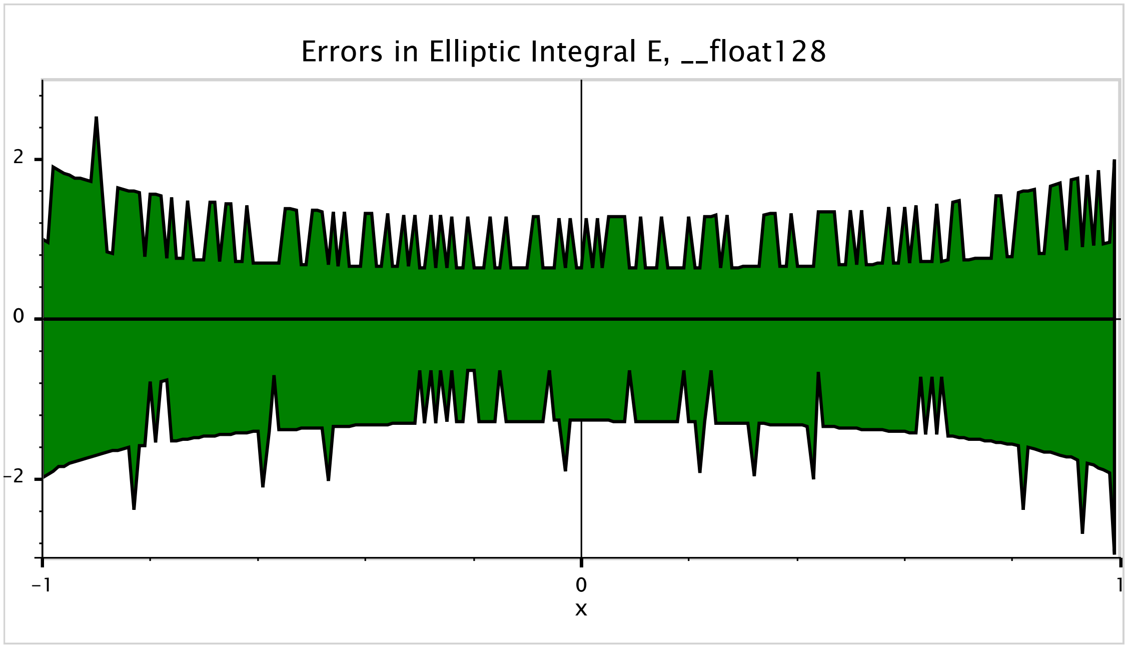
Table 87. Error rates for ellint_2

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral E: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.63ε (Mean = 0.325ε))	Max = 0.656ε (Mean = 0.317ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0.656ε (Mean = 0.317ε)	Max = 1.31ε (Mean = 0.727ε)
Elliptic Integral E: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.4ε (Mean = 1.16ε))	Max = 2.05ε (Mean = 0.632ε) (*<cmath>:* Max = 3.08e+04ε (Mean = 3.84e+03ε))	Max = 2.05ε (Mean = 0.632ε)	Max = 2.23ε (Mean = 0.639ε)
Elliptic Integral E: Small Angles	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.5ε (Mean = 0.118ε))	Max = 1ε (Mean = 0.283ε) (*<cmath>:* Max = 2ε (Mean = 0.333ε))	Max = 1ε (Mean = 0.283ε)	Max = 1ε (Mean = 0.421ε)

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The tests use a mixture of spot test values calculated using the online calculator at functions.wolfram.com, and random test data generated using NTL::RR at 1000-bit precision and this implementation.

Implementation

These functions are implemented in terms of Carlson's integrals using the relations:

$$\begin{aligned} E(-\varphi, k) &= -E(\varphi, k) \\ E(\varphi + m\pi, k) &= E(\varphi, k) + 2mE(k) \quad ; \quad \varphi \notin \left[0, \tfrac{\pi}{2}\right] \\ E(\varphi, k) &= \sin\varphi R_F\left(\cos^2\varphi,\ 1 - k^2\sin^2\varphi,\ 1\right) - \tfrac{1}{3}k^2\sin^3\varphi R_D\left(\cos^2\varphi,\ 1 - k^2\sin^2\varphi,\ 1\right) \end{aligned}$$

and

$$E(k) = R_F(0, 1 - k^2, 1) - \frac{1}{3}k^2 R_D(0, 1 - k^2, 1)$$

Elliptic Integrals of the Third Kind - Legendre Form

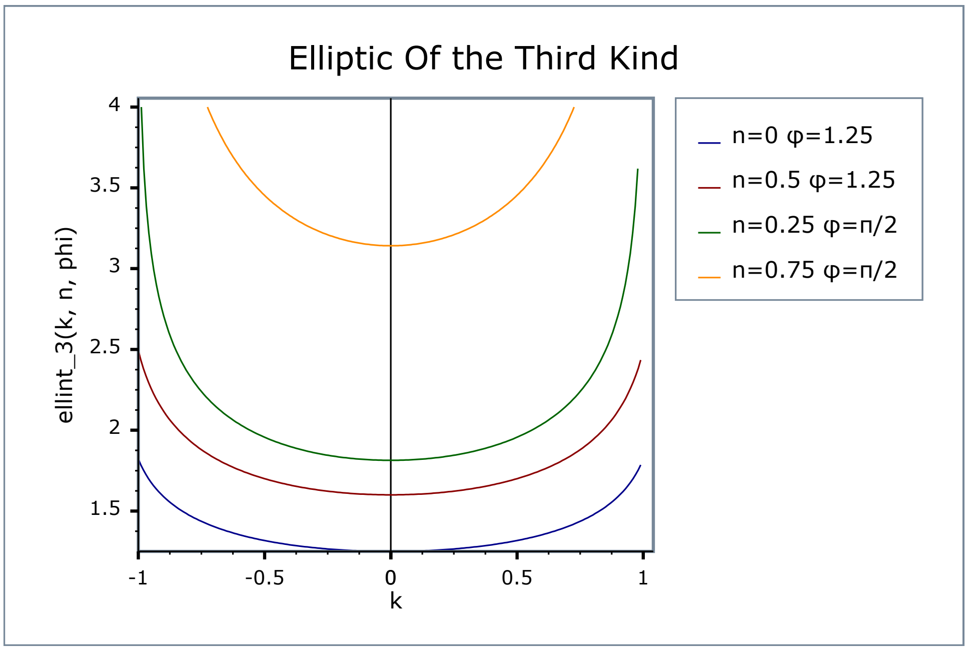
Synopsis

```
#include <boost/math/special_functions/ellint_3.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2, class T3>
calculated-result-type ellint_3(T1 k, T2 n, T3 phi);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_3(T1 k, T2 n, T3 phi, const Policy&);

template <class T1, class T2>
calculated-result-type ellint_3(T1 k, T2 n);

template <class T1, class T2, class Policy>
calculated-result-type ellint_3(T1 k, T2 n, const Policy&);

}} // namespaces
```

Description

These two functions evaluate the incomplete elliptic integral of the third kind *Π(n, φ, k)* and its complete counterpart *Π(n, k) = E(n, π/2, k)*.



The return type of these functions is computed using the *result type calculation rules* when the arguments are of different types: when they are the same type then the result is the same type as the arguments.

```
template <class T1, class T2, class T3>
calculated-result-type ellint_3(T1 k, T2 n, T3 phi);

template <class T1, class T2, class T3, class Policy>
calculated-result-type ellint_3(T1 k, T2 n, T3 phi, const Policy&);
```

Returns the incomplete elliptic integral of the third kind $\Pi(n, \phi, k)$:

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

Requires $k^2 sin^2(phi) < 1$ and $n < 1/sin^2(\phi)$, otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1, class T2>
calculated-result-type ellint_3(T1 k, T2 n);

template <class T1, class T2, class Policy>
calculated-result-type ellint_3(T1 k, T2 n, const Policy&);
```

Returns the complete elliptic integral of the first kind $\Pi(n, k)$:

$$\prod(n,\ k) = \prod\left(n,\ \frac{\pi}{2},\ k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - k^2\sin^2\theta}}$$

Requires $|k| < 1$ and $n < 1$, otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

These functions are computed using only basic arithmetic operations, so there isn't much variation in accuracy over differing platforms. Note that only results for the widest floating point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon.

Table 88. Error rates for ellint_3

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral PI: Mathworld Data	Max = 475ε (Mean = 86.3ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.48e+05ε (Mean = 2.54e+04ε) And other failures.)	Max = 475ε (Mean = 86.3ε)	Max = 565ε (Mean = 102ε)
Elliptic Integral PI: Random Data	Max = 4.54ε (Mean = 0.895ε) (*<cmath>:* Max = 3.37e+20ε (Mean = 3.47e+19ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 633ε (Mean = 50.1ε))	Max = 4.49ε (Mean = 0.885ε)	Max = 8.33ε (Mean = 0.971ε)
Elliptic Integral PI: Large Random Data	Max = 3.7ε (Mean = 0.893ε) (*<cmath>:* Max = 2.52e+18ε (Mean = 4.83e+17ε) And other failures.)	Max = 0.557ε (Mean = 0.0389ε) (*GSL 2.1:* Max = 40.1ε (Mean = 7.77ε))	Max = 3.7ε (Mean = 0.892ε)	Max = 2.86ε (Mean = 0.944ε)

Testing

The tests use a mixture of spot test values calculated using the online calculator at functions.wolfram.com, and random test data generated using NTL::RR at 1000-bit precision and this implementation.

Implementation

The implementation for Π(n, ϕ, k) first siphons off the special cases:

$$\Pi(0, \phi, k) = F(\phi, k)$$

$$\Pi(n, \pi/2, k) = \Pi(n, k)$$

and

$$\begin{aligned}\prod(n, \varphi, 0) &= \sqrt{\frac{1}{1-n}}\tan^{-1}\left(\sqrt{1-n}\tan\varphi\right) &&; n < 1\\ &= \sqrt{\frac{1}{n-1}}\tanh^{-1}\left(\sqrt{n-1}\tan\varphi\right) &&; n > 1\\ &= \tan\varphi &&; n = 1\end{aligned}$$

Then if n < 0 the relations (A&S 17.7.15/16):

$$\begin{aligned}
\sqrt{(1-n)\left(1-\tfrac{k^2}{n}\right)}\prod(n,\varphi,k) &= \sqrt{(1-N)\left(1-\tfrac{k^2}{N}\right)}\prod(N,\varphi,k)\\
&+ \tfrac{k^2}{p_2}F(\varphi,k)\\
&+ \tan^{-1}\left(\tfrac{p_2}{2}\tfrac{\sin 2\varphi}{?(\varphi)}\right)\\
N &= \tfrac{(k^2-n)}{(1-n)}\\
p_2 &= \sqrt{-\tfrac{n}{1-n}(k^2-n)}
\end{aligned}$$

are used to shift *n* to the range [0, 1].

Then the relations:

Π(n, -ϕ, k) = -Π(n, ϕ, k)

Π(n, ϕ+mπ, k) = Π(n, ϕ, k) + 2mΠ(n, k) ; n <= 1

Π(n, ϕ+mπ, k) = Π(n, ϕ, k) ; n > 1 [1]

are used to move ϕ to the range [0, π/2].

The functions are then implemented in terms of Carlson's integrals using the relations:

$$\prod(n,\varphi,k) = \sin\varphi R_F\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1\right)+\tfrac{n}{3}\sin^3\varphi R_J\left(\cos^2\varphi,\ 1-k^2\sin^2\varphi,\ 1,1-n\sin^2\varphi\right)$$

and

$$\prod(n,k) = R_F\left(0,1-k^2,\ 1\right)+\tfrac{n}{3}R_J\left(0,1-k^2,\ 1,1-n\right)$$

[1] I haven't been able to find a literature reference for this relation, but it appears to be the convention used by Mathematica. Intuitively the first *2 * m * Π(n, k)* terms cancel out as the derivative alternates between +∞ and -∞.

Elliptic Integral D - Legendre Form

Synopsis

```
#include <boost/math/special_functions/ellint_d.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type ellint_d(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_d(T1 k, T2 phi, const Policy&);

template <class T1>
calculated-result-type ellint_d(T1 k);

template <class T1, class Policy>
calculated-result-type ellint_d(T1 k, const Policy&);

}} // namespaces
```

Description

These two functions evaluate the incomplete elliptic integral $D(\phi, k)$ and its complete counterpart $D(k) = D(\pi/2, k)$.

The return type of these functions is computed using the *result type calculation rules* when the arguments are of different types: when they are the same type then the result is the same type as the arguments.

```
template <class T1, class T2>
calculated-result-type ellint_d(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type ellint_3(T1 k, T2 phi, const Policy&);
```

Returns the incomplete elliptic integral:

$$\begin{aligned}
D(\varphi, k) &= \int_0^{\varphi} \frac{\sin^2\theta}{\sqrt{(1-k^2\sin^2\theta)}}\,d\theta \\
&= \int_0^{\sin\varphi} \frac{t^2}{\sqrt{1-t^2}\sqrt{1-k^2t^2}}\,dt \\
&= \frac{(F(\varphi, k) - E(\varphi, k))}{k^2} \\
&= \tfrac{1}{3}R_D\left(c-1, c-k^2, c\right) \quad ; \quad c = \frac{1}{\sin^2\varphi}
\end{aligned}$$

Requires $k^2 sin^2(phi) < 1$, otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
template <class T1>
calculated-result-type ellint_d(T1 k);

template <class T1, class Policy>
calculated-result-type ellint_d(T1 k, const Policy&);
```

Returns the complete elliptic integral *D(k) = D(π/2, k)*

Requires *-1 <= k <= 1* otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

These functions are trivially computed in terms of other elliptic integrals and generally have very low error rates (a few epsilon) unless parameter ϕ is very large, in which case the usual trigonometric function argument-reduction issues apply.

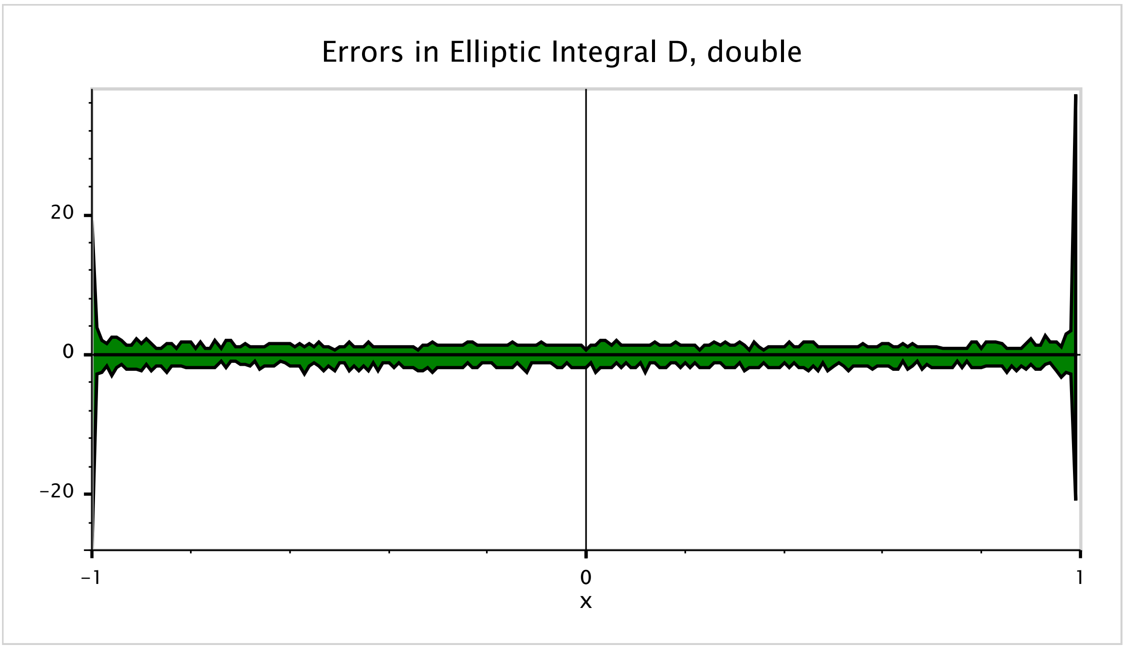
Table 89. Error rates for ellint_d (complete)

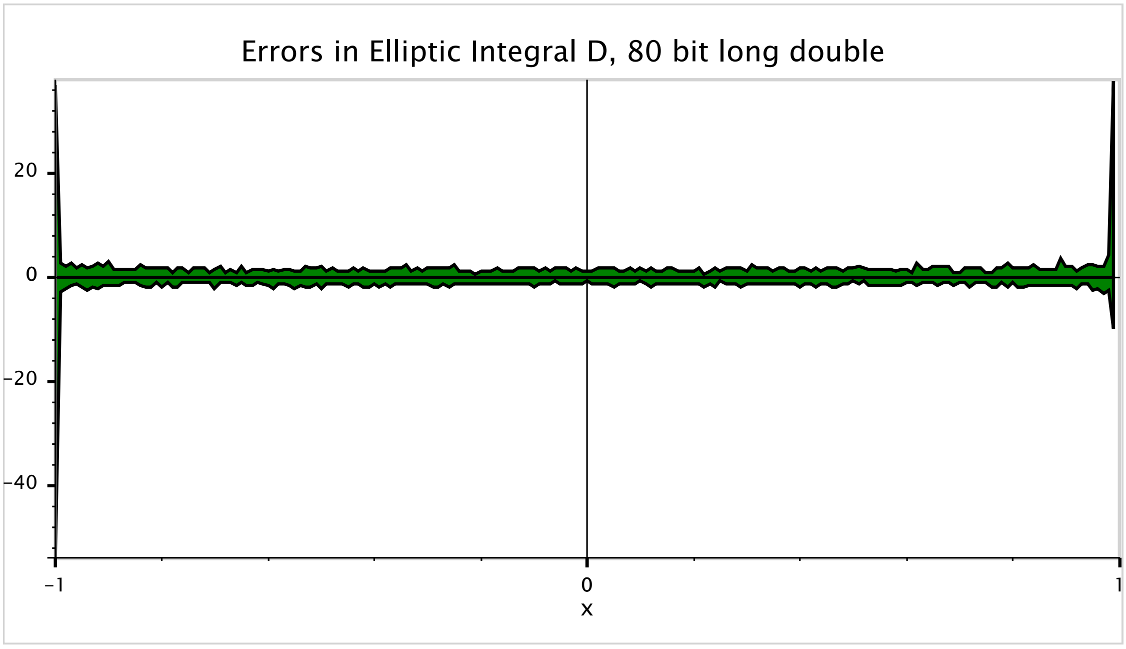
	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral E: Mathworld Data	Max = 0.637ε (Mean = 0.368ε)	Max = 1.27ε (Mean = 0.735ε)	Max = 1.27ε (Mean = 0.735ε)	Max = 0.637ε (Mean = 0.368ε)
Elliptic Integral D: Random Data	Max = 0ε (Mean = 0ε)	Max = 1.27ε (Mean = 0.334ε)	Max = 1.27ε (Mean = 0.334ε)	Max = 1.27ε (Mean = 0.355ε)

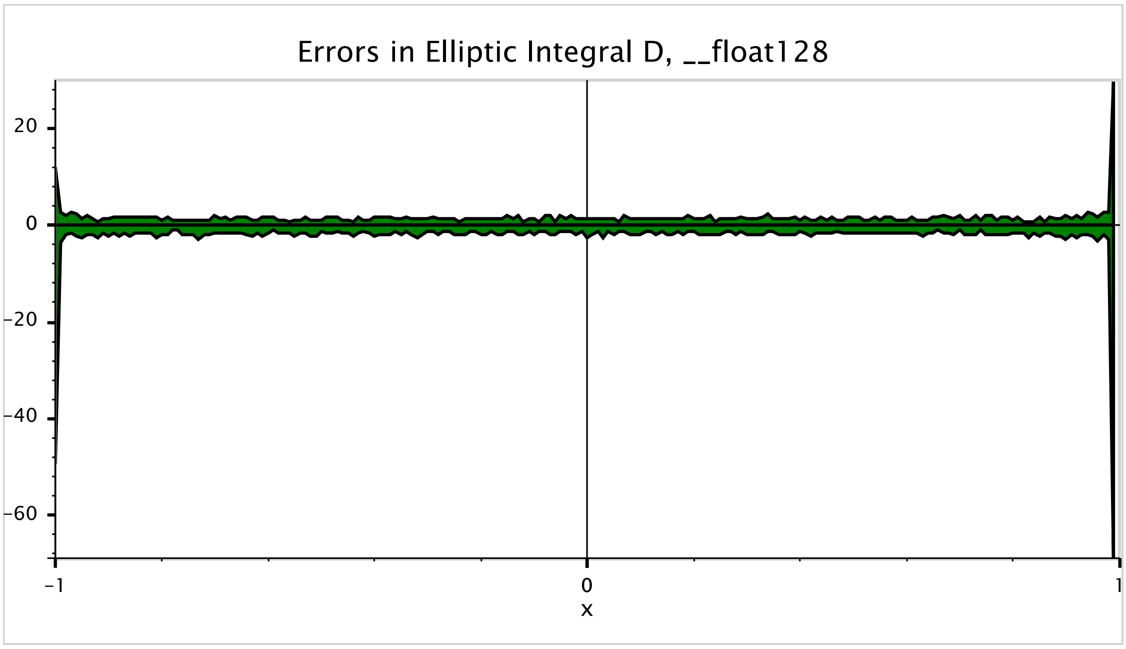
Table 90. Error rates for ellint_d

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral E: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.862ε (Mean = 0.568ε))	Max = 1.3ε (Mean = 0.813ε)	Max = 1.3ε (Mean = 0.813ε)	Max = 0.862ε (Mean = 0.457ε)
Elliptic Integral D: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.01ε (Mean = 0.928ε))	Max = 2.51ε (Mean = 0.883ε)	Max = 2.51ε (Mean = 0.883ε)	Max = 2.87ε (Mean = 0.805ε)

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.


Errors in Elliptic Integral D, double
20
0
−20
−1
0
1
x


Errors in Elliptic Integral D, 80 bit long double
20
0
−20
−40
−1
0
1
x



Testing

The tests use a mixture of spot test values calculated using values calculated at Wolfram Alpha, and random test data generated using MPFR at 1000-bit precision and a deliberately naive implementation in terms of the Legendre integrals.

Implementation

The implementation for D(ϕ, k) first performs argument reduction using the relations:

$$D(-\phi, k) = -D(\phi, k)$$

and

$$D(n\pi+\phi, k) = 2nD(k) + D(\phi, k)$$

to move ϕ to the range [0, π/2].

The functions are then implemented in terms of Carlson's integral R_D using the relation:

$$\begin{aligned} D(\varphi, k) &= \int_0^{\varphi} \frac{\sin^2\theta}{\sqrt{(1 - k^2\sin^2\theta)}} d\theta \\ &= \int_0^{\sin\varphi} \frac{t^2}{\sqrt{1-t^2}\sqrt{1-k^2t^2}} dt \\ &= \frac{(F(\varphi, k) - E(\varphi, k))}{k^2} \\ &= \tfrac{1}{3}R_D\left(c-1, c-k^2, c\right) \quad ; \quad c = \frac{1}{\sin^2\varphi} \end{aligned}$$

Jacobi Zeta Function

Synopsis

```
#include <boost/math/special_functions/jacobi_zeta.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type jacobi_zeta(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type jacobi_zeta(T1 k, T2 phi, const Policy&);

}} // namespaces
```

Description

This function evaluates the Jacobi Zeta Function *Z(ϕ, k)*

$$\begin{aligned} Z(\varphi, k) &= E(\varphi, k) - \frac{E(k)F(\varphi, k)}{K(k)} \\ &= \frac{k^2}{3K(k)}\sin\varphi\cos\varphi\sqrt{(1 - k^2\sin^2\varphi)}\,R_J(0,1 - k^2,\ 1,1 - k^2\sin^2\varphi) \end{aligned}$$

Please note the use of ϕ, and *k* as the parameters, the function is often defined as *Z(ϕ, m)* with $m = k^2$, see for example Weisstein, Eric W. "Jacobi Zeta Function." From MathWorld--A Wolfram Web Resource. Or else as *Z(x, k)* with *ϕ = am(x, k)*, where *am* is the Jacobi amplitude function which is equivalent to *asin(jacobi_elliptic(k, x))*.

The return type of this function is computed using the *result type calculation rules* when the arguments are of different types: when they are the same type then the result is the same type as the arguments.

Requires *-1 <= k <= 1*, otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that there is no complete analogue of this function (where ϕ = π / 2) as this takes the value 0 for all *k*.

Accuracy

These functions are trivially computed in terms of other elliptic integrals and generally have very low error rates (a few epsilon) unless parameter ϕ is very large, in which case the usual trigonometric function argument-reduction issues apply.

Table 91. Error rates for jacobi_zeta

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Elliptic Integral Jacobi Zeta: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.66ε (Mean = 0.48ε)	Max = 1.66ε (Mean = 0.48ε)	Max = 1.52ε (Mean = 0.357ε)
Elliptic Integral Jacobi Zeta: Random Data	Max = 0ε (Mean = 0ε)	Max = 2.99ε (Mean = 0.824ε)	Max = 3.96ε (Mean = 1.06ε)	Max = 3.89ε (Mean = 0.824ε)
Elliptic Integral Jacobi Zeta: Large Phi Values	Max = 0ε (Mean = 0ε)	Max = 2.92ε (Mean = 0.951ε)	Max = 3.05ε (Mean = 1.13ε)	Max = 2.52ε (Mean = 0.977ε)

Testing

The tests use a mixture of spot test values calculated using values calculated at Wolfram Alpha, and random test data generated using MPFR at 1000-bit precision and a deliberately naive implementation in terms of the Legendre integrals.

Implementation

The implementation for Z(ϕ, k) first makes the argument ϕ positive using:

Z(-ϕ, k) = -Z(ϕ, k)

The function is then implemented in terms of Carlson's integral R_J using the relation:

$$\begin{aligned} Z(\varphi, k) &= E(\varphi, k) - \frac{E(k)F(\varphi, k)}{K(k)} \\ &= \frac{k^2}{3K(k)}\sin\varphi\cos\varphi\sqrt{(1-k^2\sin^2\varphi)}\,R_J(0,1-k^2,\ 1,1-k^2\sin^2\varphi) \end{aligned}$$

There is one special case where the above relation fails: when *k = 1*, in that case the function simplifies to

Z(ϕ, 1) = sign(cos(ϕ)) sin(ϕ)

Example

A simple example comparing use of Wolfram Alpha with Boost.Math (including much higher precision using Boost.Multiprecision) is jacobi_zeta_example.cpp.

Heuman Lambda Function

Synopsis

```
#include <boost/math/special_functions/heuman_lambda.hpp>
```

```
namespace boost { namespace math {

template <class T1, class T2>
calculated-result-type heuman_lambda(T1 k, T2 phi);

template <class T1, class T2, class Policy>
calculated-result-type heuman_lambda(T1 k, T2 phi, const Policy&);

}} // namespaces
```

Description

This function evaluates the Heuman Lambda Function $\Lambda_0(\phi, k)$

$$\begin{aligned}\Lambda_0(\varphi,\, k) &= \frac{F\left(\varphi,\, \sqrt{1-k^2}\right)}{K\left(\sqrt{1-k^2}\right)} + \tfrac{2}{\pi}K(k)Z\left(\varphi,\, \sqrt{1-k^2}\right) \\ &= \tfrac{2}{\pi}\frac{(1-k^2)\sin\varphi\cos\varphi}{\Delta}\left(R_F\left(0,1-k^2,\, 1\right) + \tfrac{k^2}{3\Delta^2}R_J\left(0,1-k^2,\, 1,1-\tfrac{k^2}{\Delta^2}\right)\right) \quad ;\ \Delta^2 =\end{aligned}$$

The return type of this function is computed using the *result type calculation rules* when the arguments are of different types: when they are the same type then the result is the same type as the arguments.

Requires $-1 <= k <= 1$, otherwise returns the result of domain_error (outside this range the result would be complex).

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Note that there is no complete analogue of this function (where $\phi = \pi / 2$) as this takes the value 1 for all *k*.

Accuracy

These functions are trivially computed in terms of other elliptic integrals and generally have very low error rates (a few epsilon) unless parameter ϕ is very large, in which case the usual trigonometric function argument-reduction issues apply.

Table 92. Error rates for heuman_lambda

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Elliptic Integral Jacobi Zeta: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.89ε (Mean = 0.887ε)	Max = 1.89ε (Mean = 0.887ε)	Max = 1.08ε (Mean = 0.734ε)
Elliptic Integral Heuman Lambda: Random Data	Max = 0ε (Mean = 0ε)	Max = 3.82ε (Mean = 0.609ε)	Max = 3.82ε (Mean = 0.608ε)	Max = 2.12ε (Mean = 0.588ε)

Testing

The tests use a mixture of spot test values calculated using values calculated at Wolfram Alpha, and random test data generated using MPFR at 1000-bit precision and a deliberately naive implementation in terms of the Legendre integrals.

Implementation

The function is then implemented in terms of Carlson's integrals R_J and R_F using the relation:

$$\begin{aligned} \Lambda_0(\varphi, k) &= \frac{F(\varphi, \sqrt{1-k^2})}{K(\sqrt{1-k^2})} + \tfrac{2}{\pi} K(k) Z(\varphi, \sqrt{1-k^2}) \\ &= \frac{2}{\pi} \frac{(1-k^2)\sin\varphi\cos\varphi}{\Delta} \left(R_F(0, 1-k^2, 1) + \frac{k^2}{3\Delta^2} R_J\left(0, 1-k^2, 1, 1-\frac{k^2}{\Delta^2}\right)\right) \quad ; \Delta^2 = \end{aligned}$$

This relation fails for $|\phi| >= \pi/2$ in which case the definition in terms of the Jacobi Zeta is used.

Jacobi Elliptic Functions

Overview of the Jacobi Elliptic Functions

There are twelve Jacobi Elliptic functions, of which the three copolar functions *sn*, *cn* and *dn* are the most important as the other nine can be computed from these three [2] [3] [4].

These functions each take two arguments: a parameter, and a variable as described below.

Like all elliptic functions these can be parameterised in a number of ways:

- In terms of a parameter m.
- In terms of the elliptic modulus k where $m = k^2$.
- In terms of the modular angle α, where $m = sin^2\alpha$.

In our implementation, these functions all take the elliptic modulus k as the parameter.

In addition the variable u is sometimes expressed as an amplitude ϕ, in our implementation we always use u.

Finally note that our functions all take the elliptic modulus k as the **first** argument - this is for alignment with the Elliptic Integrals (but is different from other implementations, for example Mathworks).

A simple example comparing use of Wolfram Alpha with Boost.Math (including much higher precision using Boost.Multiprecision) is jacobi_zeta_example.cpp.

There are twelve functions for computing the twelve individual Jacobi elliptic functions: jacobi_cd, jacobi_cn, jacobi_cs, jacobi_dc, jacobi_dn, jacobi_ds, jacobi_nc, jacobi_nd, jacobi_ns, jacobi_sc, jacobi_sd and jacobi_sn.

They are all called as for example:

```
jacobi_cs(k, u);
```

Note however that these individual functions are all really thin wrappers around the function jacobi_elliptic which calculates the three copolar functions *sn*, *cn* and *dn* in a single function call.

Tip

If you need more than one of these functions for a given set of arguments, it's most efficient to use jacobi_elliptic.

[2] Wikipedia: Jacobi elliptic functions
[3] Weisstein, Eric W. "Jacobi Elliptic Functions." From MathWorld - A Wolfram Web Resource.
[4] Digital Library of Mathematical Functions: Jacobian Elliptic Functions, Reinhardt, W. P., Walker, O. L.

Jacobi Elliptic SN, CN and DN

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U, class V>
 calculated-result-type jacobi_elliptic(T k, U u, V* pcn, V* pdn);

 template <class T, class U, class V, class Policy>
 calculated-result-type jacobi_elliptic(T k, U u, V* pcn, V* pdn, const Policy&);

}} // namespaces
```

Description

The function jacobi_elliptic calculates the three copolar Jacobi elliptic functions *sn(u, k)*, *cn(u, k)* and *dn(u, k)*. The returned value is *sn(u, k)*, and if provided, **pcn* is set to *cn(u, k)*, and **pdn* is set to *dn(u, k)*.

The functions are defined as follows, given:

$$u = \int_0^{\varphi} \frac{d\varphi}{\sqrt{(1 - k^2 \sin^2\varphi)}}$$

The the angle ϕ is called the *amplitude* and:

$$\begin{aligned} sn(u, k) &= \sin\varphi \\ cn(u, k) &= \cos\varphi \\ dn(u, k) &= \sqrt{(1 - k^2 \sin^2\varphi)} \end{aligned}$$

Note

ϕ is called the amplitude. k is called the elliptic modulus.

Caution

Rather like other elliptic functions, the Jacobi functions are expressed in a variety of different ways. In particular, the parameter *k* (the modulus) may also be expressed using a modular angle α, or a parameter *m*. These are related by:

$$k = \sin \alpha$$

$$m = k^2 = \sin^2\alpha$$

So that the function *sn* (for example) may be expressed as either:

sn(u, k)

sn(u \ α)

sn(u | m)

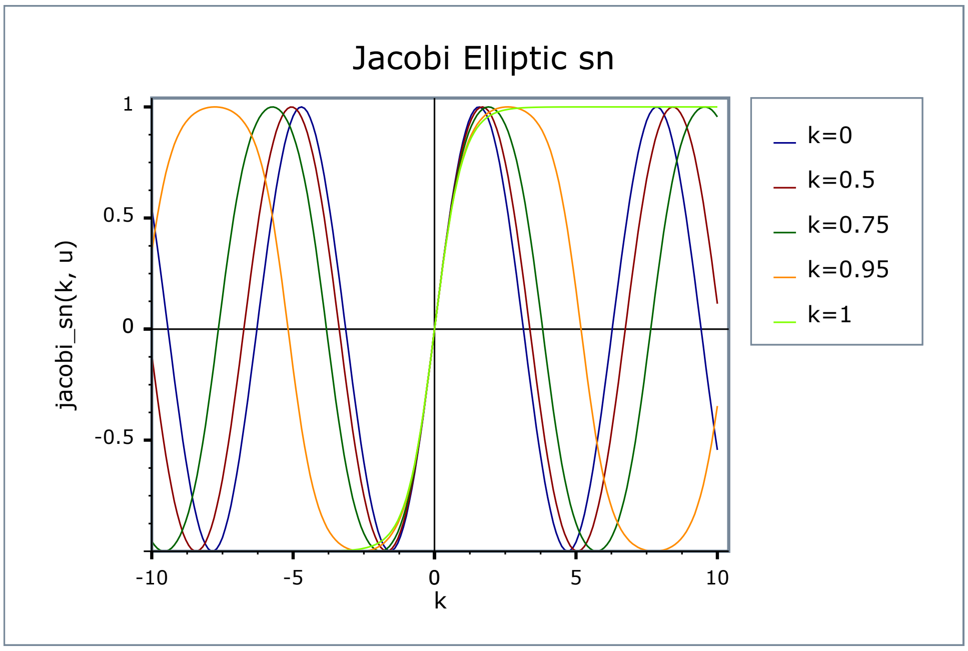
To further complicate matters, some texts refer to the *complement of the parameter m*, or 1 - m, where:

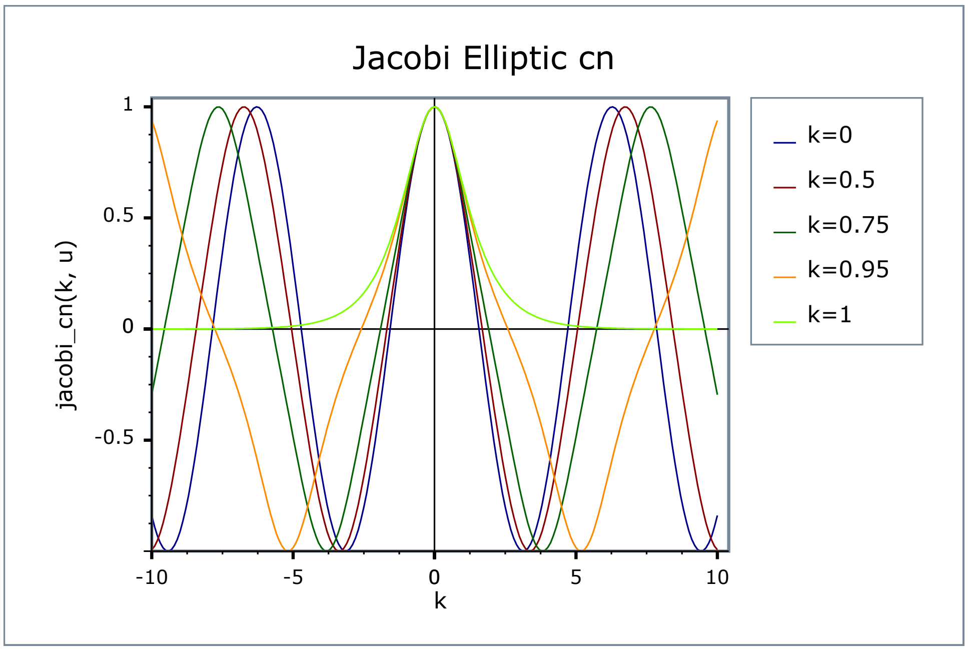
$$1 - m = 1 - k^2 = \cos^2\alpha$$

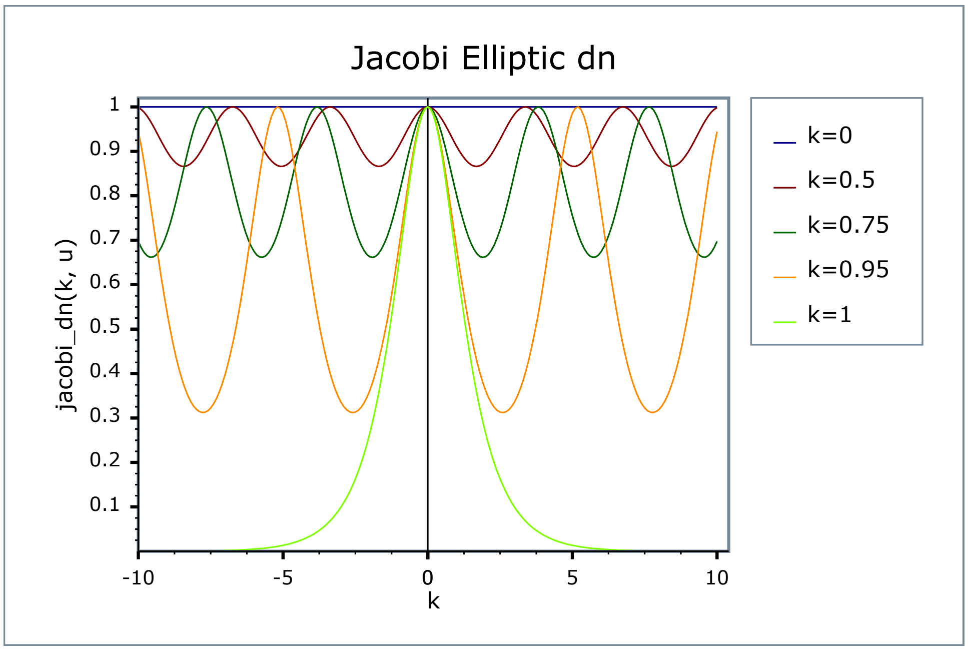
This implementation uses *k* throughout, and makes this the first argument to the functions: this is for alignment with the elliptic integrals which match the requirements of the Technical Report on C++ Library Extensions. However, you should be extra careful when using these functions!

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

The following graphs illustrate how these functions change as *k* changes: for small *k* these are sine waves, while as *k* tends to 1 they become hyperbolic functions:







Accuracy

These functions are computed using only basic arithmetic operations and trigomometric functions, so there isn't much variation in accuracy over differing platforms. Typically errors are trivially small for small angles, and as is typical for cyclic functions, grow as the angle increases. Note that only results for the widest floating-point type on the system are given as narrower types have effectively zero error. All values are relative errors in units of epsilon.

Table 93. Error rates for jacobi_cn

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 17.3ε (Mean = 4.29ε) And other failures.)	Max = 71.6ε (Mean = 19.3ε)	Max = 71.6ε (Mean = 19.4ε)	Max = 45.8ε (Mean = 11.4ε)
Jacobi Elliptic: Random Data	Max = 0.816ε (Mean = 0.0563ε) (*GSL 2.1:* Max = 2.43ε (Mean = 0.803ε))	Max = 1.68ε (Mean = 0.443ε)	Max = 1.68ε (Mean = 0.454ε)	Max = 1.83ε (Mean = 0.455ε)
Jacobi Elliptic: Random Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 55.2ε (Mean = 1.64ε) And other failures.)	Max = 10.4ε (Mean = 0.594ε)	Max = 10.4ε (Mean = 0.602ε)	Max = 26.2ε (Mean = 1.17ε)
Jacobi Elliptic: Modulus near 1	Max = 0.919ε (Mean = 0.127ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 675ε (Mean = 87.1ε)	Max = 675ε (Mean = 86.8ε)	Max = 513ε (Mean = 126ε)
Jacobi Elliptic: Large Phi	Max = 14.2ε (Mean = 0.927ε) (*GSL 2.1:* Max = 5.92e+03ε (Mean = 477ε))	Max = 2.97e+04ε (Mean = 1.9e+03ε)	Max = 2.97e+04ε (Mean = 1.9e+03ε)	Max = 3.27e+04ε (Mean = 1.93e+03ε)

Table 94. Error rates for jacobi_dn

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.82ε (Mean = 1.18ε) And other failures.)	Max = 49ε (Mean = 14ε)	Max = 49ε (Mean = 14ε)	Max = 34.3ε (Mean = 8.71ε)
Jacobi Elliptic: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3ε (Mean = 0.61ε))	Max = 1.53ε (Mean = 0.473ε)	Max = 1.53ε (Mean = 0.481ε)	Max = 1.52ε (Mean = 0.466ε)
Jacobi Elliptic: Random Small Values	Max = 0.5ε (Mean = 0.0122ε) (*GSL 2.1:* Max = 1.5ε (Mean = 0.391ε) And other failures.)	Max = 22.4ε (Mean = 0.777ε)	Max = 22.4ε (Mean = 0.763ε)	Max = 16.1ε (Mean = 0.685ε)
Jacobi Elliptic: Modulus near 1	Max = 2.28ε (Mean = 0.194ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 3.75e+03ε (Mean = 293ε)	Max = 3.75e+03ε (Mean = 293ε)	Max = 6.24e+03ε (Mean = 482ε)
Jacobi Elliptic: Large Phi	Max = 14.1ε (Mean = 0.897ε) (*GSL 2.1:* Max = 121ε (Mean = 22ε))	Max = 2.82e+04ε (Mean = 1.79e+03ε)	Max = 2.82e+04ε (Mean = 1.79e+03ε)	Max = 1.67e+04ε (Mean = 1e+03ε)

Table 95. Error rates for jacobi_sn

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 588ε (Mean = 146ε) And other failures.)	Max = 341ε (Mean = 80.7ε)	Max = 341ε (Mean = 80.7ε)	Max = 481ε (Mean = 113ε)
Jacobi Elliptic: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.02ε (Mean = 1.07ε))	Max = 2.01ε (Mean = 0.584ε)	Max = 2.01ε (Mean = 0.593ε)	Max = 1.92ε (Mean = 0.567ε)
Jacobi Elliptic: Random Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 11.7ε (Mean = 1.65ε) And other failures.)	Max = 1.99ε (Mean = 0.347ε)	Max = 1.99ε (Mean = 0.347ε)	Max = 2.11ε (Mean = 0.385ε)
Jacobi Elliptic: Modulus near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 109ε (Mean = 7.35ε)	Max = 109ε (Mean = 7.38ε)	Max = 23.2ε (Mean = 1.85ε)
Jacobi Elliptic: Large Phi	Max = 12ε (Mean = 0.771ε) (*GSL 2.1:* Max = 4.54e+04ε (Mean = 2.63e+03ε))	Max = 2.45e+04ε (Mean = 1.51e+03ε)	Max = 2.45e+04ε (Mean = 1.51e+03ε)	Max = 4.36e+04ε (Mean = 2.54e+03ε)

Testing

The tests use a mixture of spot test values calculated using the online calculator at functions.wolfram.com, and random test data generated using MPFR at 1000-bit precision and this implementation.

Implementation

For $k > 1$ we apply the relations:

$$\begin{aligned} \mu &= \frac{1}{k^2} \\ v &= uk \\ sn(u,\,k) &= \frac{sn(v,\,\mu)}{k} \\ cn(u,\,k) &= dn(v,\,\mu) \\ dn(u,\,k) &= cn(v,\,k) \end{aligned}$$

Then filter off the special cases:

$sn(0, k) = 0$ and $cn(0, k) = dn(0, k) = 1$

$sn(u, 0) = sin(u), cn(u, 0) = cos(u)$ *and* $dn(u, 0) = 1$

$sn(u, 1) = tanh(u), cn(u, 1) = dn(u, 1) = 1 / cosh(u)$

And for $k^4 < \varepsilon$ we have:

$$\begin{aligned} sn(u,\ k) &\approx \sin(u) - \tfrac{1}{4}k^2(u - \sin(u)\cos(u))\cos(u) \\ cn(u,\ k) &\approx \cos(u) + \tfrac{1}{4}k^2(u - \sin(u)\cos(u))\sin(u) \\ dn(u,\ k) &\approx 1 - \tfrac{1}{2}k^2\sin^2(u) \end{aligned}$$

Otherwise the values are calculated using the method of arithmetic geometric means.

Jacobi Elliptic Function cd

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_cd(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_cd(T k, U u, const Policy& pol);

}} // namespaces
```

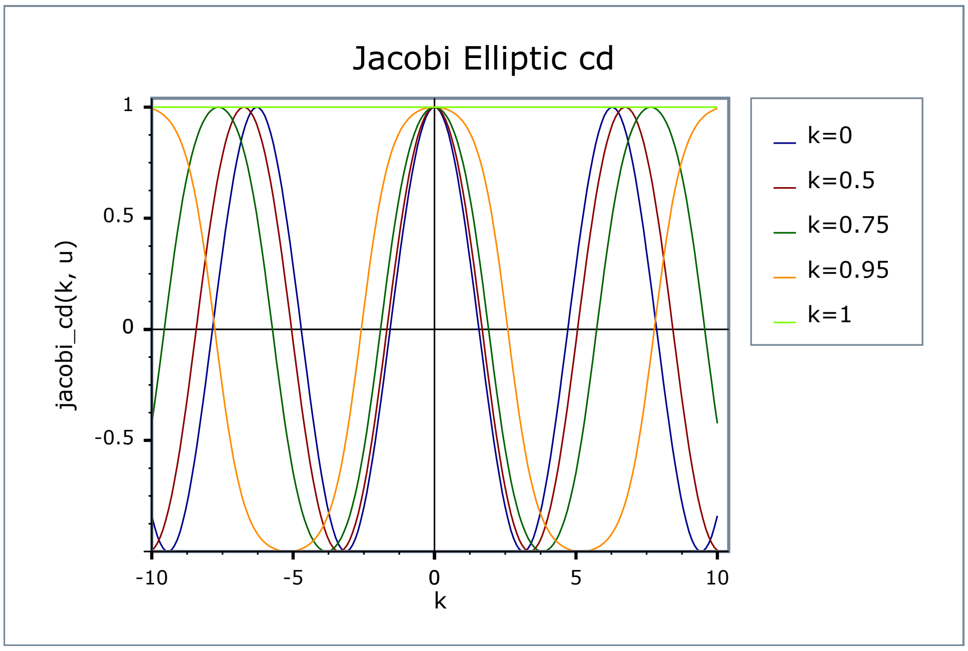
Description

This function returns the Jacobi elliptic function *cd*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

cd(u, k) = cn(u, k) / dn(u, k)



Jacobi Elliptic Function cn

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

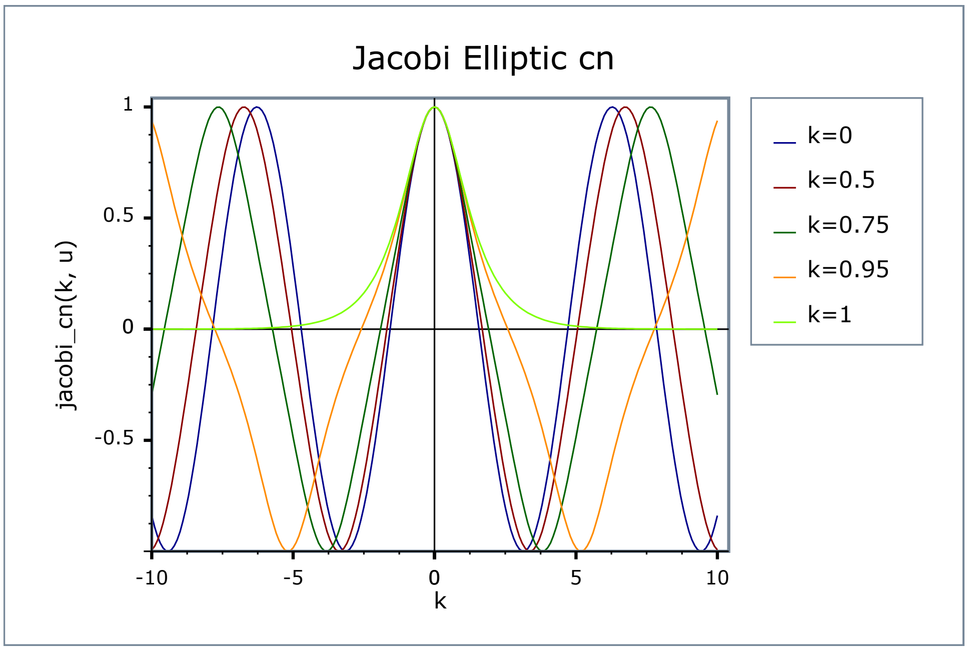
```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_cn(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_cn(T k, U u, const Policy& pol);

}} // namespaces
```

Description

This function returns the Jacobi elliptic function *cn*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic.



Jacobi Elliptic Function cs

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_cs(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_cs(T k, U u, const Policy& pol);

}} // namespaces
```

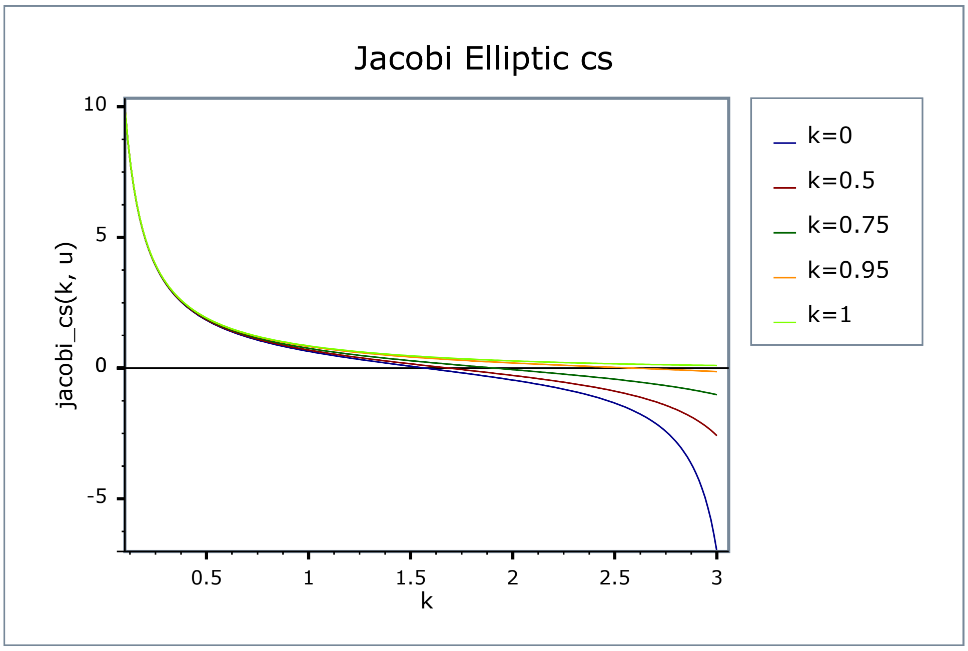
Description

This function returns the Jacobi elliptic function *cs*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

$cs(u, k) = cn(u, k) / sn(u, k)$



Jacobi Elliptic Function dc

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_dc(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_dc(T k, U u, const Policy& pol);

}} // namespaces
```

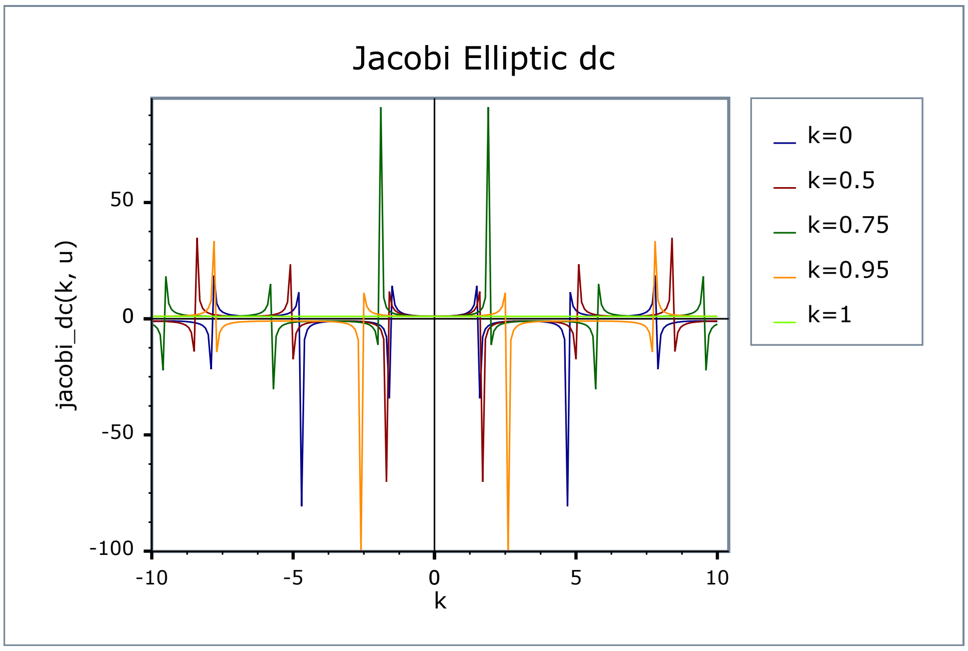
Description

This function returns the Jacobi elliptic function *dc*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

$$dc(u, k) = dn(u, k) / cn(u, k)$$



Jacobi Elliptic Function dn

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

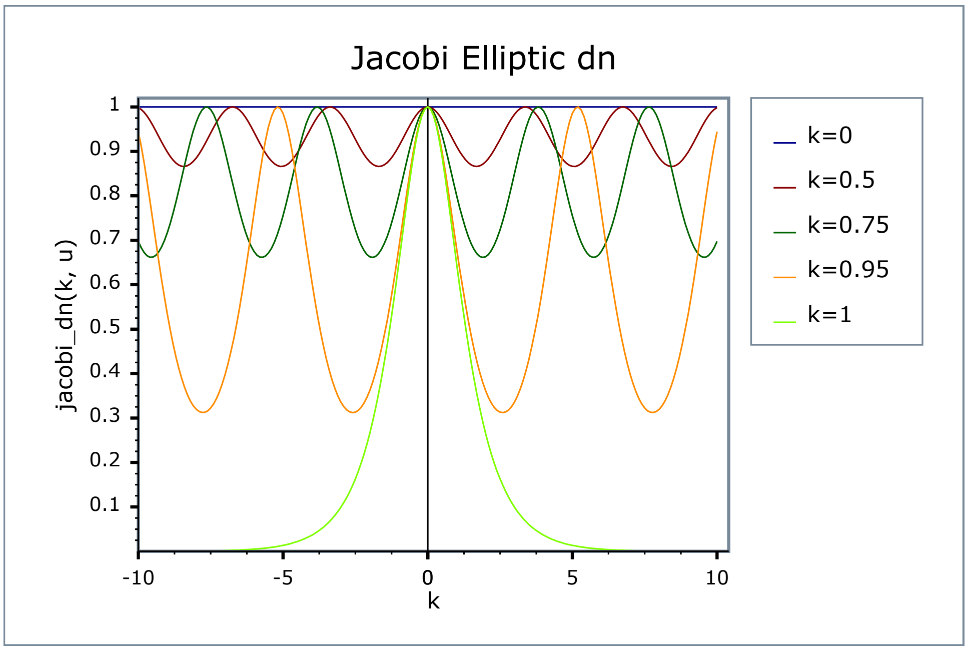
```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_dn(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_dn(T k, U u, const Policy& pol);

}} // namespaces
```

Description

This function returns the Jacobi elliptic function *dn*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic.



Jacobi Elliptic Function ds

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_ds(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_ds(T k, U u, const Policy& pol);

}} // namespaces
```

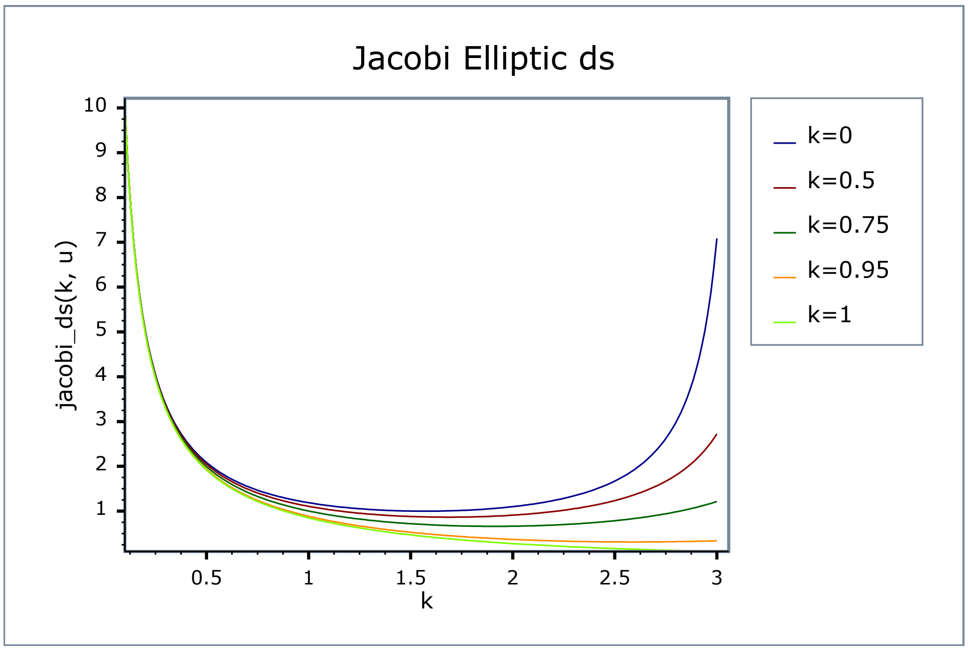
Description

This function returns the Jacobi elliptic function *ds*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

ds(u, k) = dn(u, k) / sn(u, k)



Jacobi Elliptic Function nc

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_nc(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_nc(T k, U u, const Policy& pol);

}} // namespaces
```

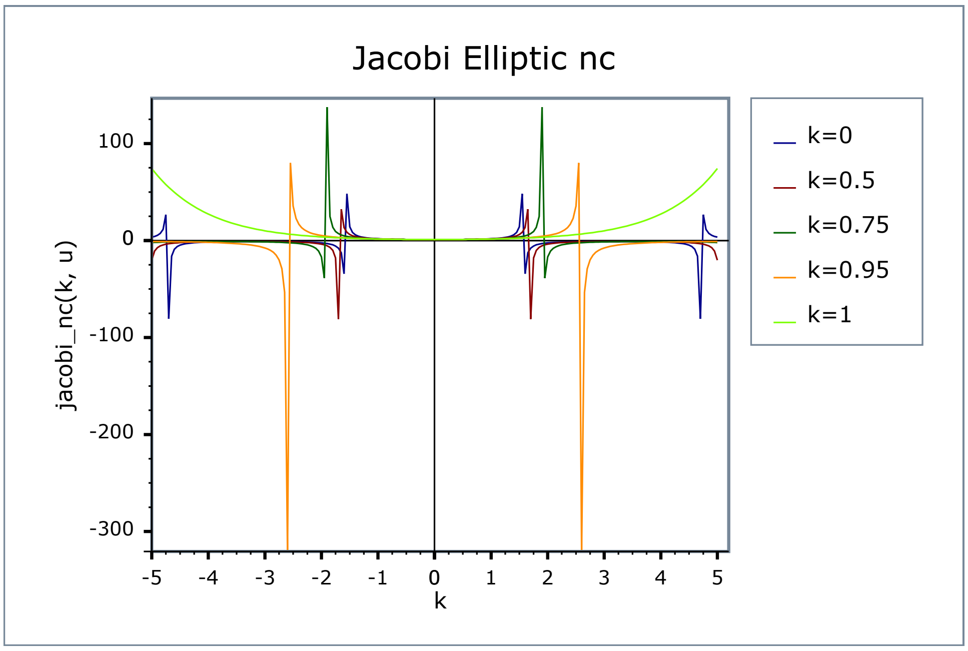
Description

This function returns the Jacobi elliptic function *nc*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

nc(u, k) = 1 / cn(u, k)



Jacobi Elliptic Function nd

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_nd(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_nd(T k, U u, const Policy& pol);

}} // namespaces
```

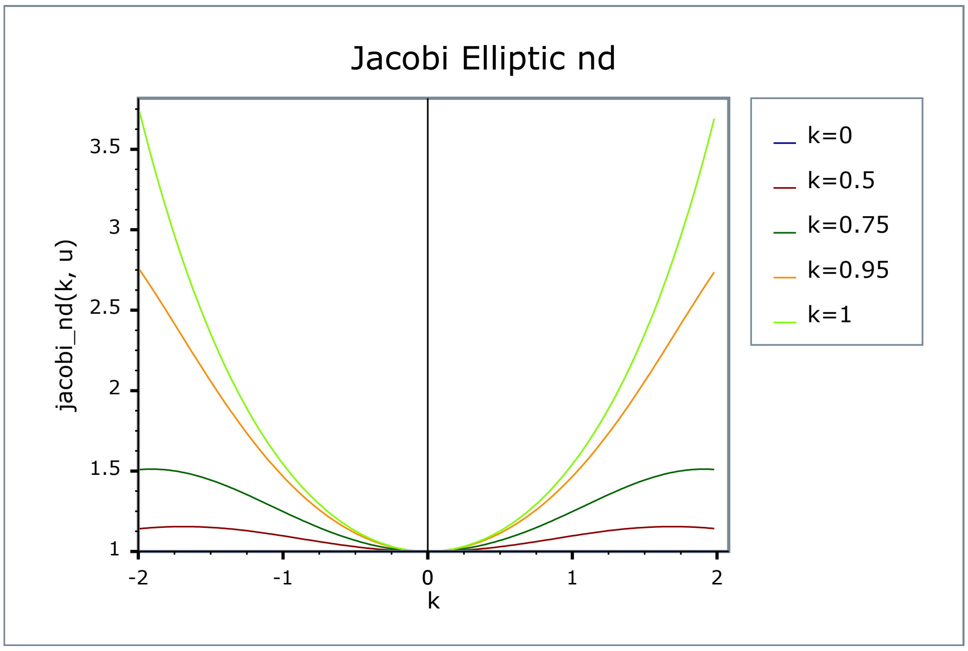
Description

This function returns the Jacobi elliptic function *nd*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

nd(u, k) = 1 / dn(u, k)



Jacobi Elliptic Function ns

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_ns(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_ns(T k, U u, const Policy& pol);

}} // namespaces
```

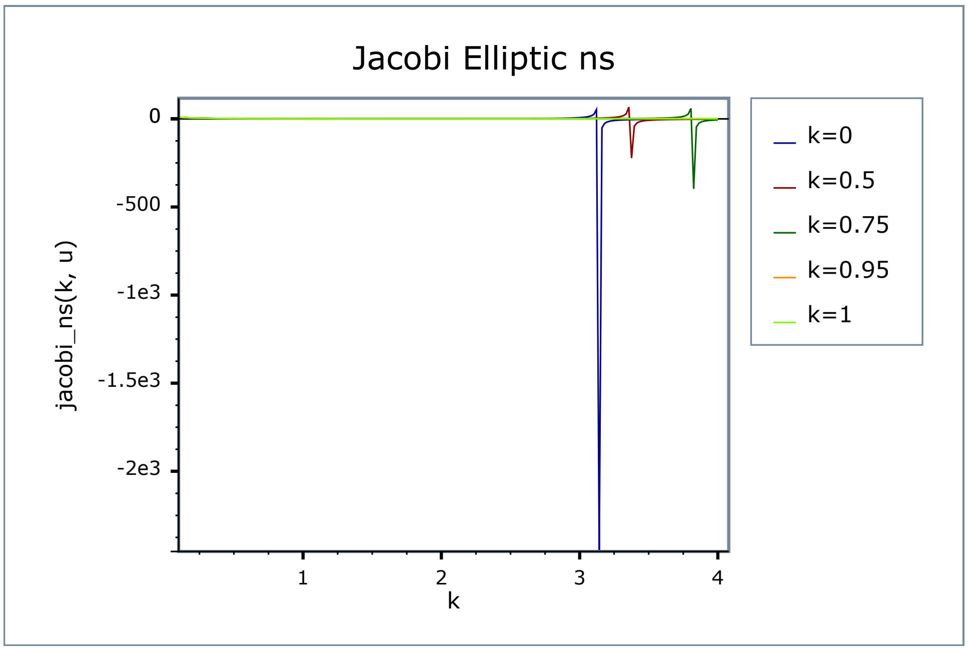
Description

This function returns the Jacobi elliptic function *ns*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

ns(u, k) = 1 / sn(u, k)



Jacobi Elliptic Function sc

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_sc(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_sc(T k, U u, const Policy& pol);

}} // namespaces
```

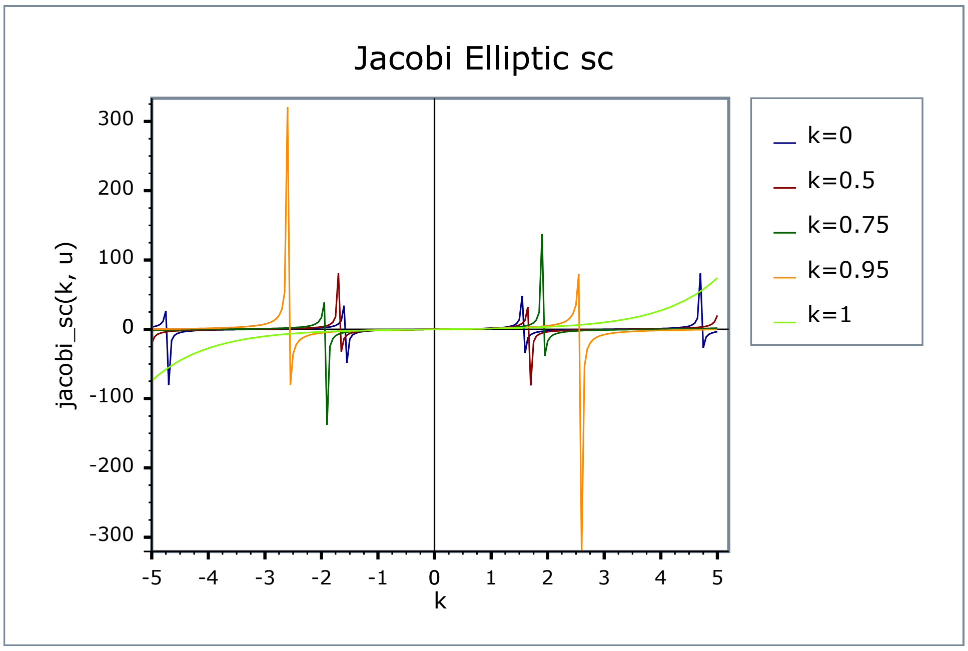
Description

This function returns the Jacobi elliptic function *sc*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

sc(u, k) = sn(u, k) / cn(u, k)



Jacobi Elliptic Function sd

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_sd(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_sd(T k, U u, const Policy& pol);

}} // namespaces
```

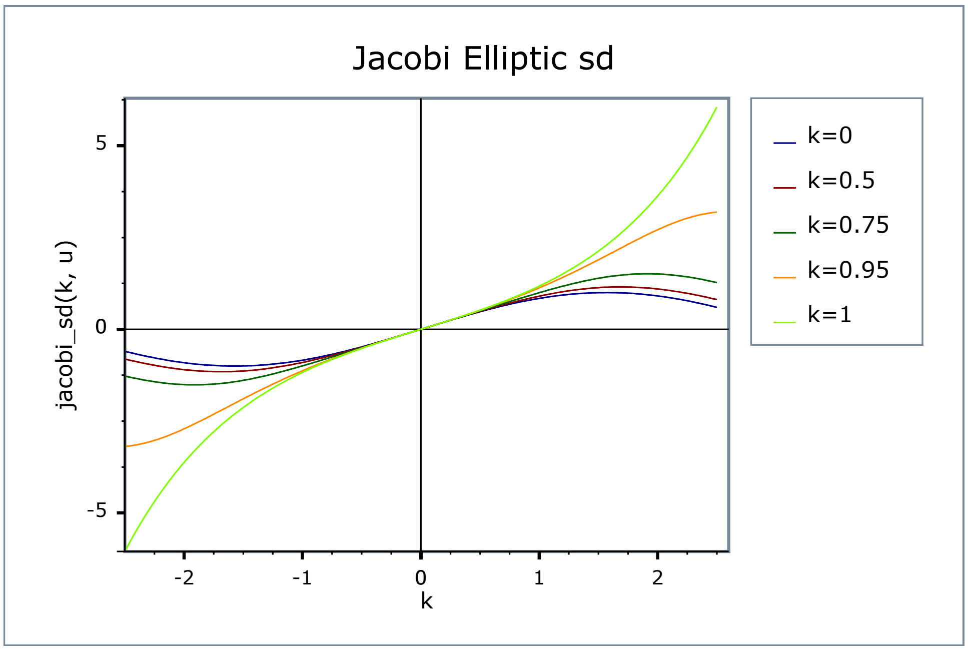
Description

This function returns the Jacobi elliptic function *sd*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic, with:

sd(u, k) = sn(u, k) / dn(u, k)



Jacobi Elliptic Function sn

Synopsis

```
#include <boost/math/special_functions/jacobi_elliptic.hpp>
```

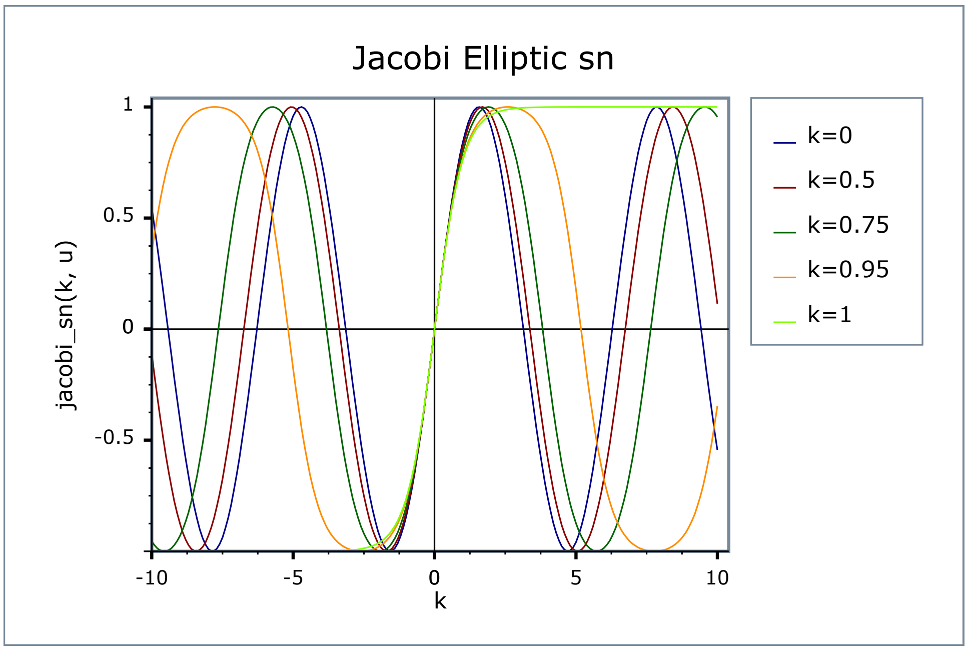
```
namespace boost { namespace math {

 template <class T, class U>
 calculated-result-type jacobi_sn(T k, U u);

 template <class T, class U, class Policy>
 calculated-result-type jacobi_sn(T k, U u, const Policy& pol);

}} // namespaces
```

Description

This function returns the Jacobi elliptic function *sn*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is a trivial wrapper around jacobi_elliptic.



Lambert *W* function

Synopsis

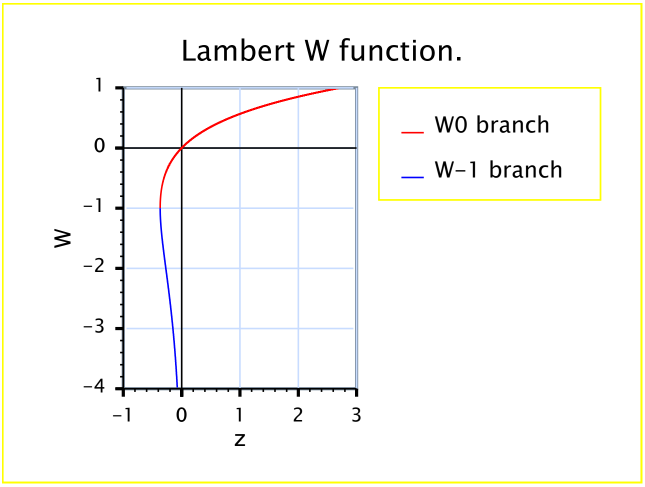
```
#include <boost/math/special_functions/lambert_w.hpp>
```

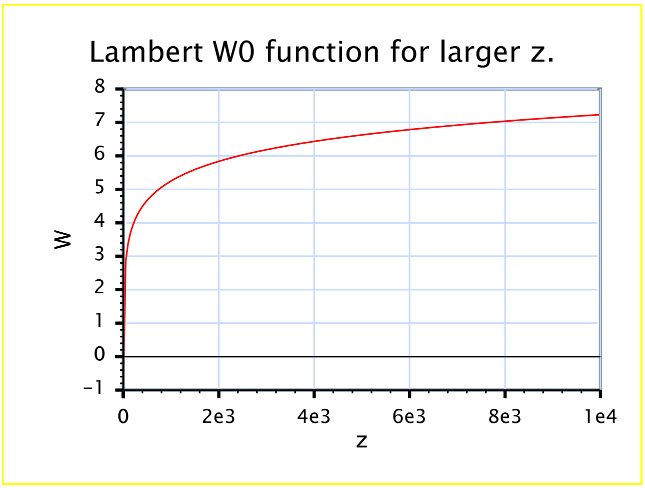
```
namespace boost { namespace math {

  template <class T>
  calculated-result-type lambert_w0(T z);                       // W0 branch, default policy.
  template <class T>
  calculated-result-type lambert_wm1(T z);                     // W-1 branch, default policy.
  template <class T>
  calculated-result-type lambert_w0_prime(T z);                   // W0 branch 1st derivative.
  template <class T>
  calculated-result-type lambert_wm1_prime(T z);                 // W-1 branch 1st derivative.

  template <class T, class Policy>
  calculated-result-type lambert_w0(T z, const Policy&);          // W0 with policy.
  template <class T, class Policy>
  calculated-result-type lambert_wm1(T z, const Policy&);         // W-1 with policy.
  template <class T, class Policy>
  calculated-result-type lambert_w0_prime(T z, const Policy&);   // W0 derivative with policy.
  template <class T, class Policy>
  calculated-result-type lambert_wm1_prime(T z, const Policy&);  // W-1 derivative with policy.

 } // namespace boost
 } // namespace math
```

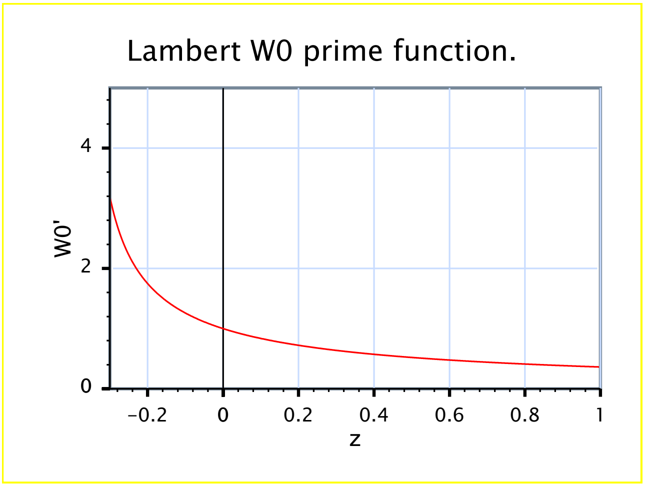
Description

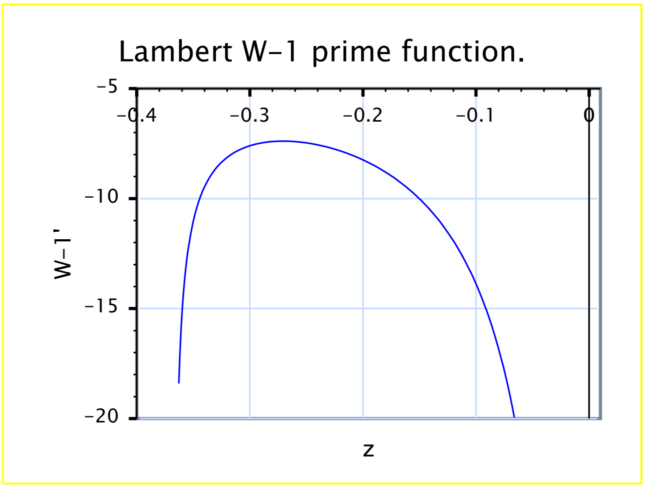
The Lambert W function is the solution of the equation $W(z)e^{W(z)} = z$. It is also called the Omega function, the inverse of $f(W) = We^{W}$.

On the interval [0, ∞), there is just one real solution. On the interval $(-e^{-1}, 0)$, there are two real solutions, generating two branches which we will denote by W_0 and W_{-1}. In Boost.Math, we call these principal branches `lambert_w0` and `lambert_wm1`; their derivatives are labelled `lambert_w0_prime` and `lambert_wm1_prime`.









There is a singularity where the branches meet at e^{-1} ≅ `-0.367879`. Approaching this point, the condition number of function evaluation tends to infinity, and the only method of recovering high accuracy is use of higher precision.

This implementation computes the two real branches W_0 and W_{-1} with the functions `lambert_w0` and `lambert_wm1`, and their derivatives, `lambert_w0_prime` and `lambert_wm1_prime`. Complex arguments are not supported.

The final Policy argument is optional and can be used to control how the function deals with errors. Refer to Policies for more details and see examples below.

Applications of the Lambert *W* function

The Lambert *W* function has a myriad of applications. Corless et al. provide a summary of applications, from the mathematical, like iterated exponentiation and asymptotic roots of trinomials, to the real-world, such as the range of a jet plane, enzyme kinetics, water movement in soil, epidemics, and diode current (an example replicated here). Since the publication of their landmark paper, there have been many more applications, and also many new implementations of the function, upon which this implementation builds.

Examples

The most basic usage of the Lambert-*W* function is demonstrated below:

```
#include <boost/math/special_functions/lambert_w.hpp> // For lambert_w function.

using boost::math::lambert_w0;
using boost::math::lambert_wm1;
```

```
std::cout.precision(std::numeric_limits<double>::max_digits10);
// Show all potentially significant decimal digits,
std::cout << std::showpoint << std::endl;
// and show significant trailing zeros too.

double z = 10.;
double r = lambert_w0(z); // Default policy for double.
std::cout << "lambert_w0(z) = " << r << std::endl;
// lambert_w0(z) = 1.7455280027406994
```

Other floating-point types can be used too, here `float`, including user-defined types like Boost.Multiprecision. It is convenient to use a function like `show_value` to display all (and only) potentially significant decimal digits, including any significant trailing zeros, (`std::numeric_limits<T>::max_digits10`) for the type `T`.

```
float z = 10.F;
float r;
r = lambert_w0(z);        // Default policy digits10 = 7, digits2 = 24
std::cout << "lambert_w0(";
show_value(z);
std::cout << ") = ";
show_value(r);
std::cout << std::endl;   // lambert_w0(10.0000000) = 1.74552798
```

Example of an integer argument to `lambert_w0`, showing that an `int` literal is correctly promoted to a `double`.

```
std::cout.precision(std::numeric_limits<double>::max_digits10);
double r = lambert_w0(10);                        // Pass an int argument "10" that should ↵
be promoted to double argument.
std::cout << "lambert_w0(10) = " << r << std::endl;  // lambert_w0(10) = 1.7455280027406994
double rp = lambert_w0(10);
std::cout << "lambert_w0(10) = " << rp << std::endl;
// lambert_w0(10) = 1.7455280027406994
auto rr = lambert_w0(10);                          // C++11 needed.
std::cout << "lambert_w0(10) = " << rr << std::endl;
// lambert_w0(10) = 1.7455280027406994 too, showing that rr has been promoted to double.
```

Using Boost.Multiprecision types to get much higher precision is painless.

```
cpp_dec_float_50 z("10");
// Note construction using a decimal digit string "10",
// NOT a floating-point double literal 10.
cpp_dec_float_50 r;
r = lambert_w0(z);
std::cout << "lambert_w0("; show_value(z); std::cout << ") = ";
show_value(r);
std::cout << std::endl;
// lambert_w0(10.000000000000000000000000000000000000000000000000000000000000000000000000000) ↵
=
//   1.7455280027406993830743012648753899115352881290809413313533156980404446940000000
```

Warning

When using multiprecision, take very great care not to construct or assign non-integers from `double`, `float` ... silently losing precision. Use `"1.234567890123456789012345678"` rather than `1.234567890123456789012345678`.

Using multiprecision types, it is all too easy to get multiprecision precision wrong!

```
cpp_dec_float_50 z(0.77777777777777777777777777777777777777777777777777777777777777777777777);
// Compiler evaluates the nearest double-precision binary representation,
// from the max_digits10 of the floating_point literal double 0.7777777777777777777777777777...,
// so any extra digits in the multiprecision type
// beyond max_digits10 (usually 17) are random and meaningless.
cpp_dec_float_50 r;
r = lambert_w0(z);
std::cout << "lambert_w0(";
show_value(z);
std::cout << ") = "; show_value(r);
std::cout << std::endl;
// lambert_w0(0.7777777777777777901135891625017393380403518676757812500000000000000000000000000)
//   = 0.48086152073210493501934682309060873341910109230469724725005039758139532631901386
```

Note

See spurious non-seven decimal digits appearing after digit #17 in the argument 0.7777777777777777...!

And similarly constructing from a literal `double 0.9`, with more random digits after digit number 17.

```
cpp_dec_float_50 z(0.9); // Construct from floating_point literal double 0.9.
cpp_dec_float_50 r;
r = lambert_w0(0.9);
std::cout << "lambert_w0(";
show_value(z);
std::cout << ") = "; show_value(r);
std::cout << std::endl;
// lambert_w0(0.9000000000000000222044604925031308084726333618164062500000000000000000000000000)
//   = 0.5298329656334344051060725178103893995285034179687500000000000000000000000000000
std::cout << "lambert_w0(0.9) = " << lambert_w0(static_cast<double>(0.9))
// lambert_w0(0.9)
//   = 0.52983296563343441
  << std::endl;
```

Note how the `cpp_float_dec_50` result is only as correct as from a `double = 0.9`.

Now see the correct result for all 50 decimal digits constructing from a decimal digit string "0.9":

```
cpp_dec_float_50 z("0.9");     // Construct from decimal digit string.
cpp_dec_float_50 r;
r = lambert_w0(z);
std::cout << "lambert_w0(";
show_value(z);
std::cout << ") = "; show_value(r);
std::cout << std::endl;
// 0.90000000000000000000000000000000000000000000000000000000000000000000000000000000)
// = 0.52983296563343441213336643954546304857788132269804249284012528304239956413801252
```

Note the expected zeros for all places up to 50 - and the correct Lambert *W* result!

(It is just as easy to compute even higher precisions, at least to thousands of decimal digits, but not shown here for brevity. See lambert_w_simple_examples.cpp for comparison of an evaluation at 1000 decimal digit precision with Wolfram Alpha).

Policies can be used to control what action to take on errors:

```
// Define an error handling policy:
typedef policy<
  domain_error<throw_on_error>,
  overflow_error<ignore_error> // possibly unwise?
> my_throw_policy;

std::cout.precision(std::numeric_limits<double>::max_digits10);
// Show all potentially significant decimal digits,
std::cout << std::showpoint << std::endl;
// and show significant trailing zeros too.
double z = +1;
std::cout << "Lambert W (" << z << ") = " << lambert_w0(z) << std::endl;
// Lambert W (1.0000000000000000) = 0.56714329040978384
std::cout << "\nLambert W (" << z << ", my_throw_policy()) = "
  << lambert_w0(z, my_throw_policy()) << std::endl;
// Lambert W (1.0000000000000000, my_throw_policy()) = 0.56714329040978384
```

An example error message:

```
Error in function boost::math::lambert_wm1<RealType>(<RealType>):
Argument z = 1 is out of range (z <= 0) for Lambert W-1 branch! (Try Lambert W0 branch?)
```

Showing an error reported if a value is passed to `lambert_w0` that is out of range, (and was probably meant to be passed to `lambert_wm1` instead).

```
double z = +1.;
double r = lambert_wm1(z);
std::cout << "lambert_wm1(+1.) = " << r << std::endl;
```

The full source of these examples is at lambert_w_simple_examples.cpp

Diode Resistance Example

A typical example of a practical application is estimating the current flow through a diode with series resistance from a paper by Banwell and Jayakumar.

Having the Lambert *W* function available makes it simple to reproduce the plot in their paper (Fig 2) comparing estimates using with Lambert *W* function and some actual measurements. The colored curves show the effect of various series resistance on the current compared to an extrapolated line in grey with no internal (or external) resistance.

Two formulae relating the diode current and effect of series resistance can be combined, but yield an otherwise intractable equation relating the current versus voltage with a varying series resistance. This was reformulated as a generalized equation in terms of the Lambert W function:

Banwell and Jakaumar equation 5

$I(V) = \mu V_T / R_S \;\; W_0(I_0 R_S / (\mu V_T))$

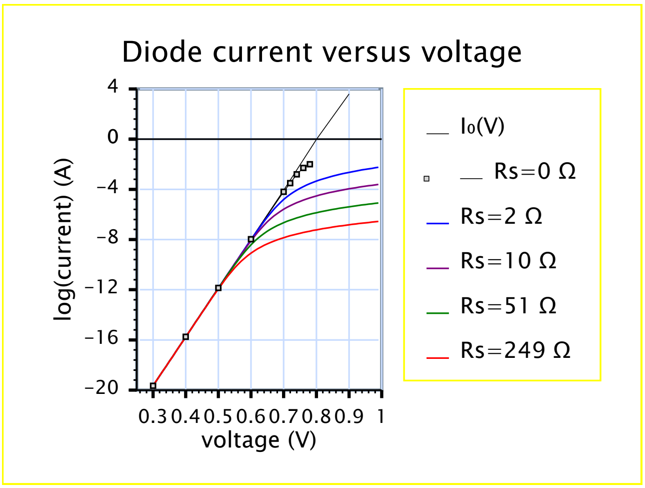
Using these variables

```
double nu = 1.0; // Assumed ideal.
double vt = v_thermal(25); // v thermal, Shockley equation, expect about 25 mV at room temper↲
ature.
double boltzmann_k = 1.38e-23; // joules/kelvin
double temp = 273 + 25;
double charge_q = 1.6e-19; // column
vt = boltzmann_k * temp / charge_q;
std::cout << "V thermal " << vt << std::endl; // V thermal 0.0257025 = 25 mV
double rsat = 0.;
double isat = 25.e-15; //  25 fA;
std::cout << "Isat = " << isat << std::endl;
double re = 0.3;  // Estimated from slope of straight section of graph (equation 6).
double v = 0.9;
double icalc = iv(v, vt, 249., re, isat);
std::cout << "voltage = " << v << ", current = " << icalc << ", " << log(icalc) << std::endl; // ↲
voltage = 0.9, current = 0.00108485, -6.82631
```

the formulas can be rendered in C++

```
double iv(double v, double vt, double rsat, double re, double isat, double nu = 1.)
{
  // V thermal 0.0257025 = 25 mV
  // was double i = (nu * vt/r) * lambert_w((i0 * r) / (nu * vt)); equ 5.

  rsat = rsat + re;
  double i = nu * vt / rsat;
 // std::cout << "nu * vt / rsat = " << i << std::endl; // 0.000103223

  double x = isat * rsat / (nu * vt);
//  std::cout << "isat * rsat / (nu * vt) = " << x << std::endl;

  double eterm = (v + isat * rsat) / (nu * vt);
 // std::cout << "(v + isat * rsat) / (nu * vt) = " << eterm << std::endl;

  double e = exp(eterm);
//  std::cout << "exp(eterm) = " << e << std::endl;

  double w0 = lambert_w0(x * e);
//  std::cout << "w0 = " << w0 << std::endl;
  return i * w0 - isat;
} // double iv
```

to reproduce their Fig 2:



The plotted points for no external series resistance (derived from their published plot as the raw data are not publicly available) are used to extrapolate back to estimate the intrinsic emitter resistance as 0.3 ohm. The effect of external series resistance is visible when the colored lines start to curve away from the straight line as voltage increases.

See lambert_w_diode.cpp and lambert_w_diode_graph.cpp for details of the calculation.

Existing implementations

The principal value of the Lambert *W* function is implemented in the Wolfram Language as `ProductLog[k, z]`, where `k` is the branch.

The symbolic algebra program Maple also computes Lambert *W* to an arbitrary precision.

Controlling the compromise between Precision and Speed

Floating-point types `double` and `float`

This implementation provides good precision and excellent speed for __fundamental `float` and `double`.

All the functions usually return values within a few Unit in the last place (ULP) for the floating-point type, except for very small arguments very near zero, and for arguments very close to the singularity at the branch point.

By default, this implementation provides the best possible speed. Very slightly average higher precision and less bias might be obtained by adding a Halley step refinement, but at the cost of more than doubling the runtime.

Floating-point types larger than double

For floating-point types with precision greater than `double` and `float` fundamental (built-in) types, a `double` evaluation is used as a first approximation followed by Halley refinement, using a single step where it can be predicted that this will be sufficient, and only using Halley iteration when necessary. Higher precision types are always going to be **very, very much slower**.

The 'best' evaluation (the nearest representable) can be achieved by `static_casting` from a higher precision type, typically a Boost.Multiprecision type like `cpp_bin_float_50`, but at the cost of increasing run-time 100-fold; this has been used here to provide some of our reference values for testing.

For example, we get a reference value using a high precision type, for example;

```
using boost::multiprecision::cpp_bin_float_50;
```

that uses Halley iteration to refine until it is as precise as possible for this `cpp_bin_float_50` type.

As a further check we can compare this with a Wolfram Alpha computation using command `N[ProductLog[10.], 50]` to get 50 decimal digits and similarly `N[ProductLog[10.], 17]` to get the nearest representable for 64-bit `double` precision.

```
  using boost::multiprecision::cpp_bin_float_50;
  using boost::math::float_distance;

  cpp_bin_float_50 z("10."); // Note use a decimal digit string, not a double 10.
  cpp_bin_float_50 r;
  std::cout.precision(std::numeric_limits<cpp_bin_float_50>::digits10);

  r = lambert_w0(z); // Default policy.
  std::cout << "lambert_w0(z) cpp_bin_float_50  = " << r << std::endl;
  //lambert_w0(z) cpp_bin_float_50  = 1.7455280027406993830743012648753899115352881290809
  //       [N[productlog[10], 50]] == 1.7455280027406993830743012648753899115352881290809
  std::cout.precision(std::numeric_limits<double>::max_digits10);
  std::cout << "lambert_w0(z) static_cast from cpp_bin_float_50  = "
    << static_cast<double>(r) << std::endl;
  // double lambert_w0(z) static_cast from cpp_bin_float_50  = 1.7455280027406994
  // [N[productlog[10], 17]]                                == 1.7455280027406994
 std::cout << "bits different from Wolfram = "
   << static_cast<int>(float_distance(static_cast<double>(r), 1.7455280027406994))
   << std::endl; // 0
```

giving us the same nearest representable using 64-bit `double` as `1.7455280027406994`.

However, the rational polynomial and Fukushima Schroder approximations are so good for type `float` and `double` that negligible improvement is gained from a `double` Halley step.

This is shown with lambert_w_precision_example.cpp for Lambert W_0:

```
using boost::math::lambert_w_detail::lambert_w_halley_step;
using boost::math::epsilon_difference;
using boost::math::relative_difference;

std::cout << std::showpoint << std::endl; // and show any significant trailing zeros too.
std::cout.precision(std::numeric_limits<double>::max_digits10); // 17 decimal digits for double.

cpp_bin_float_50 z50("1.23"); // Note: use a decimal digit string, not a double 1.23!
double z = static_cast<double>(z50);
cpp_bin_float_50 w50;
w50 = lambert_w0(z50);
std::cout.precision(std::numeric_limits<cpp_bin_float_50>::max_digits10); // 50 decimal digits.
std::cout << "Reference Lambert W (" << z << ") =\n                                    ↵
    "
  << w50 << std::endl;
std::cout.precision(std::numeric_limits<double>::max_digits10); // 17 decimal digits for double.
double wr = static_cast<double>(w50);
std::cout << "Reference Lambert W (" << z << ") =    " << wr << std::endl;

double w = lambert_w0(z);
std::cout << "Rat/poly Lambert W  (" << z << ")  =   " << lambert_w0(z) << std::endl;
// Add a Halley step to the value obtained from rational polynomial approximation.
double ww = lambert_w_halley_step(lambert_w0(z), z);
std::cout << "Halley Step Lambert W (" << z << ") =  " << lambert_w_halley_step(lam↵
bert_w0(z), z) << std::endl;

std::cout << "absolute difference from Halley step = " << w - ww << std::endl;
std::cout << "relative difference from Halley step = " << relative_differ↵
ence(w, ww) << std::endl;
std::cout << "epsilon difference from Halley step  = " << epsilon_difference(w, ww) << std::endl;
std::cout << "epsilon for float =                     " << std::numeric_limits<double>::epsi↵
lon() << std::endl;
std::cout << "bits different from Halley step  =     " << static_cast<int>(float_dis↵
tance(w, ww)) << std::endl;
```

with this output:

```
Reference Lambert W (1.2299999999999999822364316059974953532218933105468750) =
0.64520356959320237759035605255334853830173300262666480
Reference Lambert W (1.2300000000000000) =    0.64520356959320235
Rat/poly Lambert W  (1.2300000000000000)  =   0.64520356959320224
Halley Step Lambert W (1.2300000000000000) =  0.64520356959320235
absolute difference from Halley step = -1.1102230246251565e-16
relative difference from Halley step = 1.7207329236029286e-16
epsilon difference from Halley step  = 0.77494921535422934
epsilon for float =                    2.2204460492503131e-16
bits different from Halley step  =     1
```

and then for W_{-1}:

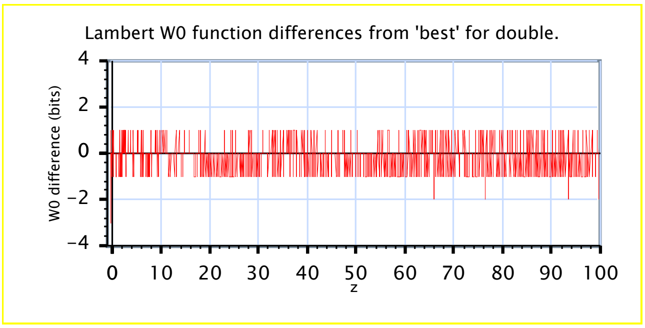
```
using boost::math::lambert_w_detail::lambert_w_halley_step;
using boost::math::epsilon_difference;
using boost::math::relative_difference;

std::cout << std::showpoint << std::endl; // and show any significant trailing zeros too.
std::cout.precision(std::numeric_limits<double>::max_digits10); // 17 decimal digits for double.

cpp_bin_float_50 z50("-0.123"); // Note: use a decimal digit string, not a double -1.234!
double z = static_cast<double>(z50);
cpp_bin_float_50 wm1_50;
wm1_50 = lambert_wm1(z50);
std::cout.precision(std::numeric_limits<cpp_bin_float_50>::max_digits10); // 50 decimal digits.
std::cout << "Reference Lambert W-1 (" << z << ") =\n                                              ↵
          "
  << wm1_50 << std::endl;
std::cout.precision(std::numeric_limits<double>::max_digits10); // 17 decimal digits for double.
double wr = static_cast<double>(wm1_50);
std::cout << "Reference Lambert W-1 (" << z << ") =    " << wr << std::endl;

double w = lambert_wm1(z);
std::cout << "Rat/poly Lambert W-1 (" << z << ")  =    " << lambert_wm1(z) << std::endl;
// Add a Halley step to the value obtained from rational polynomial approximation.
double ww = lambert_w_halley_step(lambert_wm1(z), z);
std::cout << "Halley Step Lambert W (" << z << ") =    " << lambert_w_halley_step(lam↵
bert_wm1(z), z) << std::endl;

std::cout << "absolute difference from Halley step = " << w - ww << std::endl;
std::cout << "relative difference from Halley step = " << relative_differ↵
ence(w, ww) << std::endl;
std::cout << "epsilon difference from Halley step  = " << epsilon_difference(w, ww) << std::endl;
std::cout << "epsilon for float =                    " << std::numeric_limits<double>::epsi↵
lon() << std::endl;
std::cout << "bits different from Halley step  =     " << static_cast<int>(float_dis↵
tance(w, ww)) << std::endl;
```

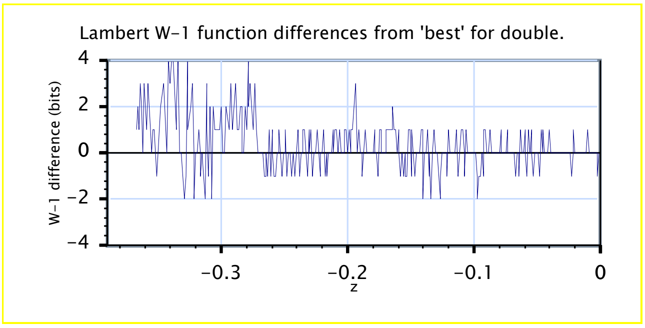
with this output:

```
Reference Lambert W-1 (-0.12299999999999999822364316059974953532218933105468750) =
-3.2849102557740360179084675531714935199110302996513384
Reference Lambert W-1 (-0.12300000000000000) =    -3.2849102557740362
Rat/poly Lambert W-1 (-0.12300000000000000)  =    -3.2849102557740357
Halley Step Lambert W (-0.12300000000000000) =    -3.2849102557740362
absolute difference from Halley step = 4.4408920985006262e-16
relative difference from Halley step = 1.3519066740696092e-16
epsilon difference from Halley step  = 0.60884463935795785
epsilon for float =                    2.2204460492503131e-16
bits different from Halley step  =     -1
```

Distribution of differences from 'best' `double` evaluations

The distribution of differences from 'best' are shown in these graphs comparing `double` precision evaluations with reference 'best' z50 evaluations using `cpp_bin_float_50` type reduced to `double` with `static_cast<double(z50)` :





As noted in the implementation section, the distribution of these differences is somewhat biased for Lambert W_{-1} and this might be reduced using a `double` Halley step at small runtime cost. But if you are seriously concerned to get really precise computations, the only way is using a higher precision type and then reduce to the desired type. Fortunately, Boost.Multiprecision makes this very easy to program, if much slower.

Edge and Corner cases

The W_0 Branch

The domain of W_0 is $[-e^{-1}, \infty)$. Numerically,

- `lambert_w0(-1/e)` is exactly -1.
- `lambert_w0(z)` for `z < -1/e` throws a `domain_error`, or returns `NaN` according to the policy.
- `lambert_w0(std::numeric_limits<T>::infinity())` throws an `overflow_error`.

(An infinite argument probably indicates that something has already gone wrong, but if it is desired to return infinity, this case should be handled before calling `lambert_w0`).

W_{-1} Branch

The domain of W_{-1} is $[-e^{-1}, 0)$. Numerically,

- `lambert_wm1(-1/e)` is exactly -1.
- `lambert_wm1(0)` returns -∞ (or the nearest equivalent if `std::has_infinity == false`).
- `lambert_wm1(-std::numeric_limits<T>::min())` returns the maximum (most negative) possible value of Lambert *W* for the type T.
  For example, for `double`: lambert_wm1(-2.2250738585072014e-308) = -714.96865723796634
  and for `float`: lambert_wm1(-1.17549435e-38) = -91.8567734
- `z < -std::numeric_limits<T>::min()`, means that z is zero or denormalized (if `std::numeric_limits<T>::has_denorm_min == true`), for example: `r = lambert_wm1(-std::numeric_limits<double>::denorm_min());` and an overflow_error exception is thrown, and will give a message like:

> Error in function boost::math::lambert_wm1<RealType>(<RealType>): Argument z = -4.9406564584124654e-324 is too small (z < -std::numeric_limits<T>::min so denormalized) for Lambert W-1 branch!

Denormalized values are not supported for Lambert W_{-1} (because not all floating-point types denormalize), and anyway it only covers a tiny fraction of the range of possible z arguments values.

Compilers

The `lambert_w.hpp` code has been shown to work on most C++98 compilers. (Apart from requiring C++11 extensions for using of `std::numeric_limits<>::max_digits10` in some diagnostics. Many old pre-c++11 compilers provide this extension but may require enabling to use, for example using b2/bjam the lambert_w examples use this command:

```
[ run lambert_w_basic_example.cpp  : : : [ requires cxx11_numeric_limits ] ]
```

See jamfile.v2.)

For details of which compilers are expected to work see lambert_w tests and examples in:
Boost Test Summary report for master branch (used for latest release)
Boost Test Summary report for latest developer branch.

As expected, debug mode is very much slower than release.

Diagnostics Macros

Several macros are provided to output diagnostic information (potentially **much** output). These can be statements, for example:

`#define BOOST_MATH_INSTRUMENT_LAMBERT_W_TERMS`

placed **before** the `lambert_w` include statement

`#include <boost/math/special_functions/lambert_w.hpp>`,

or defined on the project compile command-line: `/DBOOST_MATH_INSTRUMENT_LAMBERT_W_TERMS`,

or defined in a jamfile.v2: `<define>BOOST_MATH_INSTRUMENT_LAMBERT_W_TERMS`

```
// #define-able macros
BOOST_MATH_INSTRUMENT_LAMBERT_W_HALLEY                     // Halley refinement diagnostics.
BOOST_MATH_INSTRUMENT_LAMBERT_W_PRECISION                  // Precision.
BOOST_MATH_INSTRUMENT_LAMBERT_WM1                          // W1 branch diagnostics.
BOOST_MATH_INSTRUMENT_LAMBERT_WM1_HALLEY                   // Halley refinement diagnostics ↵
only for W-1 branch.
BOOST_MATH_INSTRUMENT_LAMBERT_WM1_TINY                   // K > 64, z > -1.026438969951130 3e-
26
BOOST_MATH_INSTRUMENT_LAMBERT_WM1_LOOKUP                 // Show results from W-1 lookup table.
BOOST_MATH_INSTRUMENT_LAMBERT_W_SCHROEDER                 // Schroeder refinement diagnostics.
BOOST_MATH_INSTRUMENT_LAMBERT_W_TERMS                      // Number of terms used for near-
singularity series.
BOOST_MATH_INSTRUMENT_LAMBERT_W_SINGULARITY_SERIES        // Show evaluation of series near ↵
branch singularity.
BOOST_MATH_INSTRUMENT_LAMBERT_W_SMALL_Z_SERIES
BOOST_MATH_INSTRUMENT_LAMBERT_W_SMALL_Z_SERIES_ITERATIONS  // Show evaluation of series for ↵
small z.
```

Implementation

There are many previous implementations, each with increasing accuracy and/or speed. See references below.

For most of the range of *z* arguments, some initial approximation followed by a single refinement, often using Halley or similar method, gives a useful precision. For speed, several implementations avoid evaluation of a iteration test using the exponential function, estimating that a single refinement step will suffice, but these rarely get to the best result possible. To get a better precision, additional refinements, probably iterative, are needed for example, using Halley or Schröder methods.

For C++, the most precise results possible, closest to the nearest representable for the C++ type being used, it is usually necessary to use a higher precision type for intermediate computation, finally static-casting back to the smaller desired result type. This strategy is used by Maple and Wolfram Alpha, for example, using arbitrary precision arithmetic, and some of their high-precision values are used for testing this library. This method is also used to provide some Boost.Test values using Boost.Multiprecision, typically, a 50 decimal digit type like `cpp_bin_float_50` `static_cast` to a `float`, `double` or `long double` type.

For *z* argument values near the singularity and near zero, other approximations may be used, possibly followed by refinement or increasing number of series terms until a desired precision is achieved. At extreme arguments near to zero or the singularity at the branch point, even this fails and the only method to achieve a really close result is to cast from a higher precision type.

In practical applications, the increased computation required (often towards a thousand-fold slower and requiring much additional code for Boost.Multiprecision) is not justified and the algorithms here do not implement this. But because the Boost.Lambert_W algorithms has been tested using Boost.Multiprecision, users who require this can always easily achieve the nearest representation for fundamental (built-in) types - if the application justifies the very large extra computation cost.

Evolution of this implementation

One compact real-only implementation was based on an algorithm by Thomas Luu, Thesis, University College London (2015), (see routine 11 on page 98 for his Lambert W algorithm) and his Halley refinement is used iteratively when required. A first implementation was based on Thomas Luu's code posted at Boost Trac #11027. It has been implemented from Luu's algorithm but templated on `RealType` parameter and result and handles both fundamental (built-in) types (`float, double, long double`), Boost.Multiprecision, and also has been tested successfully with a proposed fixed_point type.

A first approximation was computed using the method of Barry et al (see references 5 & 6 below). This was extended to the widely used TOMS443 FORTRAN and C++ versions by John Burkardt using Schroeder refinement(s). (For users only requiring an accuracy of relative accuracy of 0.02%, Barry's function alone might suffice, but a better rational function approximation method has since been developed for this implementation).

We also considered using Newton-Raphson iteration method.

```
f(w) = w e^w -z = 0 // Luu equation 6.37
f'(w) = e^w (1 + w), Wolfram alpha (d)/(dw)(f(w) = w exp(w) - z) = e^w (w + 1)
if (f(w) / f'(w) -1 < tolerance
w1 = w0 - (expw0 * (w0 + 1)); // Refine new Newton/Raphson estimate.
```

but concluded that since the Newton-Raphson method takes typically 6 iterations to converge within tolerance, whereas Halley usually takes only 1 to 3 iterations to achieve an result within 1 Unit in the last place (ULP), so the Newton-Raphson method is unlikely to be quicker than the additional cost of computing the 2nd derivative for Halley's method.

Halley refinement uses the simplified formulae obtained from Wolfram Alpha

```
[2(z exp(z)-w) d/dx (z exp(z)-w)] / [2 (d/dx (z exp(z)-w))^2 - (z exp(z)-w) d^2/dx^2 (z exp(z)-w)]
```

Implementing Compact Algorithms

The most compact algorithm can probably be implemented using the log approximation of Corless et al. followed by Halley iteration (but is also slowest and least precise near zero and near the branch singularity).

Implementing Faster Algorithms

More recently, the Tosio Fukushima has developed an even faster algorithm, avoiding any transcendental function calls as these are necessarily expensive. The current implementation of Lambert W_{-1} is based on his algorithm starting with a translation from Fukushima's FORTRAN into C++ by Darko Veberic.

Many applications of the Lambert W function make many repeated evaluations for Monte Carlo methods; for these applications speed is very important. Luu, and Chapeau-Blondeau and Monir provide typical usage examples.

Fukushima improves the important observation that much of the execution time of all previous iterative algorithms was spent evaluating transcendental functions, usually `exp`. He has put a lot of work into avoiding any slow transcendental functions by using lookup tables and bisection, finishing with a single Schroeder refinement, without any check on the final precision of the result (necessarily evaluating an expensive exponential).

Theoretical and practical tests confirm that Fukushima's algorithm gives Lambert W estimates with a known small error bound (several Unit in the last place (ULP)) over nearly all the range of z argument.

A mean difference was computed to express the typical error and is often about 0.5 epsilon, the theoretical minimum. Using the Boost.Math float_distance, we can also express this as the number of bits that are different from the nearest representable or 'exact' or 'best' value. The number and distribution of these few bits differences was studied by binning, including their sign. Bins for (signed) 0, 1, 2 and 3 and 4 bits proved suitable.

However, though these give results within a few machine epsilon of the nearest representable result, they do not get as close as is very often possible with further refinement, nrealy always to within one or two machine epsilon.

More significantly, the evaluations of the sum of all signed differences using the Fukshima algorithm show a slight bias, being more likely to be a bit or few below the nearest representation than above; bias might have unwanted effects on some statistical computations.

Fukushima's method also does not cover the full range of z arguments of 'float' precision and above.

For this implementation of Lambert W_0, John Maddock used the Boost.Math Remez algorithm method program to devise a rational function for several ranges of argument for the W_0 branch of Lambert W function. These minimax rational approximations are combined for an algorithm that is both smaller and faster.

Sadly it has not proved practical to use the same Remez algorithm method for Lambert W_{-1} branch and so the Fukushima algorithm is retained for this branch.

An advantage of both minimax rational Remez algorithm approximations is that the **distribution** from the reference values is reasonably random and insignificantly biased.

For example, table below a test of Lambert W_0 10000 values of argument covering the main range of possible values, 10000 comparisons from z = 0.0501 to 703, in 0.001 step factor 1.05 when module 7 == 0

Table 96. Fukushima Lambert W_0 and typical improvement from a single Halley step.

Method	Exact	One_bit	Two_bits	Few_bits	inexact	bias
Schroeder W_0	8804	1154	37	5	1243	-1193
after Halley step	9710	288	2	0	292	22

Lambert W_0 values computed using the Fukushima method with Schroeder refinement gave about 1/6 `lambert_w0` values that are one bit different from the 'best', and < 1% that are a few bits 'wrong'. If a Halley refinement step is added, only 1 in 30 are even one bit different, and only 2 two-bits 'wrong'.

Table 97. Rational polynomial Lambert W_0 and typical improvement from a single Halley step.

Method	Exact	One_bit	Two_bits	Few_bits	inexact	bias
rational/polynomial	7135	2863	2	0	2867	-59
after Halley step	9724	273	3	0	279	5

With the rational polynomial approximation method, there are a third one-bit from the best and none more than two-bits. Adding a Halley step (or iteration) reduces the number that are one-bit different from about a third down to one in 30; this is unavoidable 'computational noise'. An extra Halley step would double the runtime for a tiny gain and so is not chosen for this implementation, but remains a option, as detailed above.

For the Lambert W_{-1} branch, the Fukushima algorithm is used.

Table 98. Lambert W_{-1} using Fukushima algorithm.

Method	Exact	One_bit	Two_bits	Few_bits	inexact	bias
Fukushima W_{-1}	7167	2704	129	0	2962	-160
plus Halley step	7379	2529	92	0	2713	549

Lookup tables

For speed during the bisection, Fukushima's algorithm computes lookup tables of powers of e and z for integral Lambert W. There are 64 elements in these tables. The FORTRAN version (and the C++ translation by Veberic) computed these (once) as `static` data. This is slower, may cause trouble with multithreading, and is slightly inaccurate because of rounding errors from repeated(64) multiplications.

In this implementation the array values have been computed using Boost.Multiprecision 50 decimal digit and output as C++ arrays 37 decimal digit `long double` literals using `max_digits10` precision

```
std::cout.precision(std::numeric_limits<cpp_bin_float_quad>::max_digits10);
```

The arrays are as `const` and `constexpr` and `static` as possible (for the compiler version), using BOOST_STATIC_CONSTEXPR macro. (See lambert_w_lookup_table_generator.cpp The precision was chosen to ensure that if used as `long double` arrays, then the values output to lambert_w_lookup_table.ipp will be the nearest representable value for the type chose by a `typedef` in lambert_w.hpp.

```
typedef double lookup_t; // Type for lookup table (`double` or `float`, or even `long double`?)
```

This is to allow for future use at higher precision, up to platforms that use 128-bit (hardware or software) for their `long double` type.

The accuracy of the tables was confirmed using Wolfram Alpha and agrees at the 37th decimal place, so ensuring that the value is exactly read into even 128-bit `long double` to the nearest representation.

Higher precision

For types more precise than `double`, Fukushima reported that it was best to use the `double` estimate as a starting point, followed by refinement using Halley iterations or other methods; our experience confirms this.

Using Boost.Multiprecision it is simple to compute very high precision values of Lambert W at least to thousands of decimal digits over most of the range of z arguments.

For this reason, the lookup tables and bisection are only carried out at low precision, usually `double`, chosen by the `typedef double lookup_t`. Unlike the FORTRAN version, the lookup tables of Lambert_W of integral values are precomputed as C++ static arrays of floating-point literals. The default is a `typedef` setting the type to `double`. To allow users to vary the precision from `float` to `long double` these are computed to 128-bit precision to ensure that even platforms with `long double` do not lose precision.

The FORTRAN version and translation only permits the z argument to be the largest items in these lookup arrays, `wm0s[64] = 3.99049`, producing an error message and returning `NaN`. So 64 is the largest possible value ever returned from the `lambert_w0` function. This is far from the `std::numeric_limits<>::max()` for even `float`s. Therefore this implementation uses an approximation or 'guess' and Halley's method to refine the result. Logarithmic approximation is discussed at length by R.M.Corless et al. (page 349). Here we use the first two terms of equation 4.19:

```
T lz = log(z);
T llz = log(lz);
guess = lz - llz + (llz / lz);
```

This gives a useful precision suitable for Halley refinement.

Similarly, for Lambert W_{-1} branch, tiny values very near zero, W > 64 cannot be computed using the lookup table. For this region, an approximation followed by a few (usually 3) Halley refinements. See wm1_near_zero.

For the less well-behaved regions for Lambert W_0 z arguments near zero, and near the branch singularity at $-1/e$, some series functions are used.

Small values of argument z near zero

When argument z is small and near zero, there is an efficient and accurate series evaluation method available (implemented in `lambert_w0_small_z`). There is no equivalent for the W_{-1} branch as this only covers argument `z < -1/e`. The cutoff used `abs(z) < 0.05` is as found by trial and error by Fukushima.

Coefficients of the inverted series expansion of the Lambert W function around `z = 0` are computed following Fukushima using 17 terms of a Taylor series computed using Wolfram Mathematica with

```
InverseSeries[Series[z Exp[z],{z,0,17}]]
```

See Tosio Fukushima, Journal of Computational and Applied Mathematics 244 (2013), page 86.

To provide higher precision constants (34 decimal digits) for types larger than `long double`,

```
InverseSeries[Series[z Exp[z],{z,0,34}]]
```

were also computed, but for current hardware it was found that evaluating a `double` precision and then refining with Halley's method was quicker and more accurate.

Decimal values of specifications for built-in floating-point types below are 21 digits precision == `std::numeric_limits<T>::max_digits10` for `long double`.

Specializations for `lambert_w0_small_z` are provided for `float`, `double`, `long double`, `float128` and for Boost.Multiprecision types.

The `tag_type` selection is based on the value `std::numeric_limits<T>::max_digits10` (and **not** on the floating-point type T). This distinguishes between `long double` types that commonly vary between 64 and 80-bits, and also compilers that have a `float` type using 64 bits and/or `long double` using 128-bits.

As noted in the implementation section above, it is only possible to ensure the nearest representable value by casting from a higher precision type, computed at very, very much greater cost.

For multiprecision types, first several terms of the series are tabulated and evaluated as a polynomial: (this will save us a bunch of expensive calls to `pow`). Then our series functor is initialized "as if" it had already reached term 18, enough evaluation of built-in 64-bit double and float (and 80-bit `long double`) types. Finally the functor is called repeatedly to compute as many additional series terms as necessary to achive the desired precision, set from `get_epsilon` (or terminated by `evaluation_error` on reaching the set iteration limit `max_series_iterations`).

A little more than one decimal digit of precision is gained by each additional series term. This allows computation of Lambert W near zero to at least 1000 decimal digit precision, given sufficient compute time.

Argument z near the singularity at -1/e between branches W_0 and W_{-1}

Variants of Function `lambert_w_singularity_series` are used to handle *z* arguments which are near to the singularity at `z = -exp(-1) = -3.6787944` where the branches W_0 and W_{-1} join.

T. Fukushima / Journal of Computational and Applied Mathematics 244 (2013) 77-89 describes using Wolfram Mathematica

```
InverseSeries\[Series\[sqrt\[2(p Exp\[1 + p\] + 1)\], {p,-1, 20}\]\]
```

to provide his Table 3, page 85.

This implementation used Wolfram Mathematica to obtain 40 series terms at 50 decimal digit precision

```
N\[InverseSeries\[Series\[Sqrt\[2(p Exp\[1 + p\] + 1)\], { p,-1,40 }\]\], 50\]

-1+p-p^2/3+(11 p^3)/72-(43 p^4)/540+(769 p^5)/17280-(221 p^6)/8505+(680863 p^7)/43545600 ...
```

These constants are computed at compile time for the full precision for any `RealType T` using the original rationals from Fukushima Table 3.

Longer decimal digits strings are rationals pre-evaluated using Wolfram Mathematica. Some integer constants overflow, so largest size available is used, suffixed by `uLL`.

Above the 14th term, the rationals exceed the range of `unsigned long long` and are replaced by pre-computed decimal values at least 21 digits precision == `max_digits10` for `long double`.

A macro `BOOST_MATH_TEST_VALUE` (defined in test_value.hpp) taking a decimal floating-point literal was used to allow testing with both built-in floating-point types like `double` which have contructors taking literal decimal values like `3.14`, **and** also multiprecision and other User-defined Types that only provide full-precision construction from decimal digit strings like `"3.14"`. (Construction of multiprecision types from built-in floating-point types only provides the precision of the built-in type, like `double`, only 17 decimal digits).

Tip

Be exceeding careful not to silently lose precision by constructing multiprecision types from literal decimal types, usually `double`. Use decimal digit strings like "3.1459" instead. See examples.

Fukushima's implementation used 20 series terms; it was confirmed that using more terms does not usefully increase accuracy.

Lambert W_{-1} arguments values very near zero.

The lookup tables of Fukushima have only 64 elements, so that the z argument nearest zero is -1.0264389699511303e-26, that corresponds to a maximum Lambert W_{-1} value of 64.0. Fukushima's implementation did not cater for z argument values that are smaller (nearer to zero), but this implementation adds code to accept smaller (but not denormalised) values of z. A crude approximation for these very small values is to take the exponent and multiply by ln[10] ~= 2.3. We also tried the approximation first proposed by Corless et al. using ln(-z), (equation 4.19 page 349) and then tried improving by a 2nd term -ln(ln(-z)), and finally the ratio term -ln(ln(-z))/ln(-z).

For a z very close to z = -1.0264389699511303e-26 when W = 64, when effect of ln(ln(-z) term, and ratio L1/L2 is greatest, the possible 'guesses' are

```
z = -1.e-
26, w = -64.02, guess = -64.0277, ln(-z) = -59.8672, ln(-ln(-z) = 4.0921, llz/lz = -0.0684
```

whereas at the minimum (unnormalized) z

```
z = -2.2250e-
308, w = -714.9, guess = -714.9687, ln(-z) = -708.3964, ln(-ln(-z) = 6.5630, llz/lz = -0.0092
```

Although the addition of the 3rd ratio term did not reduce the number of Halley iterations needed, it might allow return of a better low precision estimate **without any Halley iterations**. For the worst case near w = 64, the error in the 'guess' is 0.008, ratio 0.0001 or 1 in 10,000 digits 10 ~= 4. Two log evalutations are still needed, but is probably over an order of magnitude faster.

Halley's method was then used to refine the estimate of Lambert W_{-1} from this guess. Experiments showed that although all approximations reached with Unit in the last place (ULP) of the closest representable value, the computational cost of the log functions was easily paid by far fewer iterations (typically from 8 down to 4 iterations for double or float).

Halley refinement

After obtaining a double approximation, for `double`, `long double` and `quad` 128-bit precision, a single iteration should suffice because Halley iteration should triple the precision with each step (as long as the function is well behaved - and it is), and since we have at least half of the bits correct already, one Halley step is ample to get to 128-bit precision.

Lambert W Derivatives

The derivatives are computed using the formulae in Wikipedia.

Testing

Initial testing of the algorithm was done using a small number of spot tests.

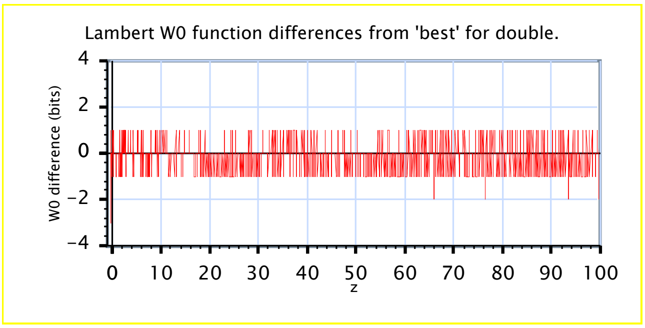
After it was established that the underlying algorithm (including unlimited Halley refinements with a tight terminating criterion) was correct, some tables of Lambert W values were computed using a 100 decimal digit precision Boost.Multiprecision `cpp_dec_float_100` type and saved as a C++ program that will initialise arrays of values of z arguments and lambert_W0 (`lambert_w_mp_high_values.ipp` and `lambert_w_mp_low_values.ipp`).

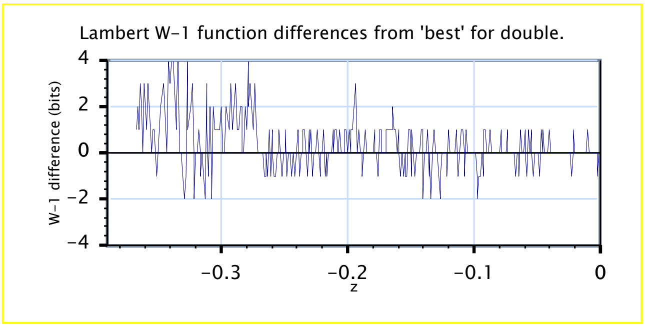
(A few of these pairs were checked against values computed by Wolfram Alpha to try to guard against mistakes; all those tested agreed to the penultimate decimal place, so they can be considered reliable to at least 98 decimal digits precision).

A macro `BOOST_MATH_TEST_VALUE` was used to allow tests with any real type, both fundamental (built-in) types and Boost.Multiprecision. (This is necessary because fundamental (built-in) types have a constructor from floating-point literals like 3.1459F, 3.1459 or 3.1459L whereas Boost.Multiprecision types may lose precision unless constructed from decimal digits strings like "3.1459").

The 100-decimal digits precision pairs were then used to assess the precision of less-precise types, including Boost.Multiprecision `cpp_bin_float_quad` and `cpp_bin_float_50`. `static_casting` from the high precision types should give the closest representable value of the less-precise type; this is then be used to assess the precision of the Lambert W algorithm.

Tests using confirm that over nearly all the range of z arguments, nearly all estimates are the nearest representable value, a minority are within 1 Unit in the last place (ULP) and only a very few 2 ULP.





For the range of z arguments over the range -0.35 to 0.5, a different algorithm is used, but the same technique of evaluating reference values using a Boost.Multiprecision `cpp_dec_float_100` was used. For extremely small z arguments, near zero, and those extremely near the singularity at the branch point, precision can be much lower, as might be expected.

See source at: lambert_w_simple_examples.cpp test_lambert_w.cpp contains routine tests using Boost.Test. lambert_w_errors_graph.cpp generating error graphs.

Testing with quadrature

A further method of testing over a wide range of argument z values was devised by Nick Thompson (cunningly also to test the recently written quadrature routines including Boost.Multiprecision !). These are definite integral formulas involving the W function that are exactly known constants, for example, LambertW0(1/(z²) == √(2π), see Definite Integrals. Some care was needed to avoid overflow and underflow as the integral function must evaluate to a finite result over the entire range.

Other implementations

The Lambert W has also been discussed in a Boost thread.

This also gives link to a prototype version by which also gives complex results `(x < -exp(-1)`, about -0.367879). Balazs Cziraki 2016 Physicist, PhD student at Eotvos Lorand University, ELTE TTK Institute of Physics, Budapest. has also produced a prototype C++ library that can compute the Lambert W function for floating point **and complex number types**. This is not implemented here but might be completed in the future.

Acknowledgements

- Thanks to Wolfram for use of their invaluable online Wolfram Alpha service.
- Thanks for Mark Chapman for performing offline Wolfram computations.

References

1. NIST Digital Library of Mathematical Functions. http://dlmf.nist.gov/4.13.F1.
2. Lambert W Poster, R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffery and D. E. Knuth, On the Lambert W function Advances in Computational Mathematics, Vol 5, (1996) pp 329-359.

3. TOMS443, Andrew Barry, S. J. Barry, Patricia Culligan-Hensley, Algorithm 743: WAPR - A Fortran routine for calculating real values of the W-function,
ACM Transactions on Mathematical Software, Volume 21, Number 2, June 1995, pages 172-181.
BISECT approximates the W function using bisection (GNU licence). Original FORTRAN77 version by Andrew Barry, S. J. Barry, Patricia Culligan-Hensley, this version by C++ version by John Burkardt.

4. TOMS743 Fortran 90 (updated 2014).

Initial guesses based on:

1. R.M.Corless, G.H.Gonnet, D.E.G.Hare, D.J.Jeffrey, and D.E.Knuth, On the Lambert W function, Adv.Comput.Math., vol. 5, pp. 329 to 359, (1996).

2. D.A. Barry, J.-Y. Parlange, L. Li, H. Prommer, C.J. Cunningham, and F. Stagnitti. Analytical approximations for real values of the Lambert W-function. Mathematics and Computers in Simulation, 53(1), 95-103 (2000).

3. D.A. Barry, J.-Y. Parlange, L. Li, H. Prommer, C.J. Cunningham, and F. Stagnitti. Erratum to analytical approximations for real values of the Lambert W-function. Mathematics and Computers in Simulation, 59(6):543-543, 2002.

4. C++ CUDA NVidia GPU C/C++ language support version of Luu algorithm, plog.

5. Thomas Luu, Thesis, University College London (2015), see routine 11, page 98 for Lambert W algorithm.

6. Having Fun with Lambert W(x) Function, Darko Veberic University of Nova Gorica, Slovenia IK, Forschungszentrum Karlsruhe, Germany, J. Stefan Institute, Ljubljana, Slovenia.

7. François Chapeau-Blondeau and Abdelilah Monir, Numerical Evaluation of the Lambert W Function and Application to Generation of Generalized Gaussian Noise With Exponent 1/2, IEEE Transactions on Signal Processing, 50(9) (2002) 2160 - 2165.

8. Toshio Fukushima, Precise and fast computation of Lambert W-functions without transcendental function evaluations, Journal of Computational and Applied Mathematics, 244 (2013) 77-89.

9. T.C. Banwell and A. Jayakumar, Electronic Letter, Feb 2000, 36(4), pages 291-2. Exact analytical solution for current flow through diode with series resistance. https://doi.org/10.1049/el:20000301

10. Princeton Companion to Applied Mathematics, 'The Lambert-W function', Section 1.3: Series and Generating Functions.

11. Cleve Moler, Mathworks blog The Lambert W Function

12. Digital Library of Mathematical Function, Lambert W function.

Zeta Functions

Riemann Zeta Function

Synopsis

```
#include <boost/math/special_functions/zeta.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type zeta(T z);

template <class T, class Policy>
calculated-result-type zeta(T z, const Policy&);

}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: the return type is `double` if T is an integer type, and T otherwise.

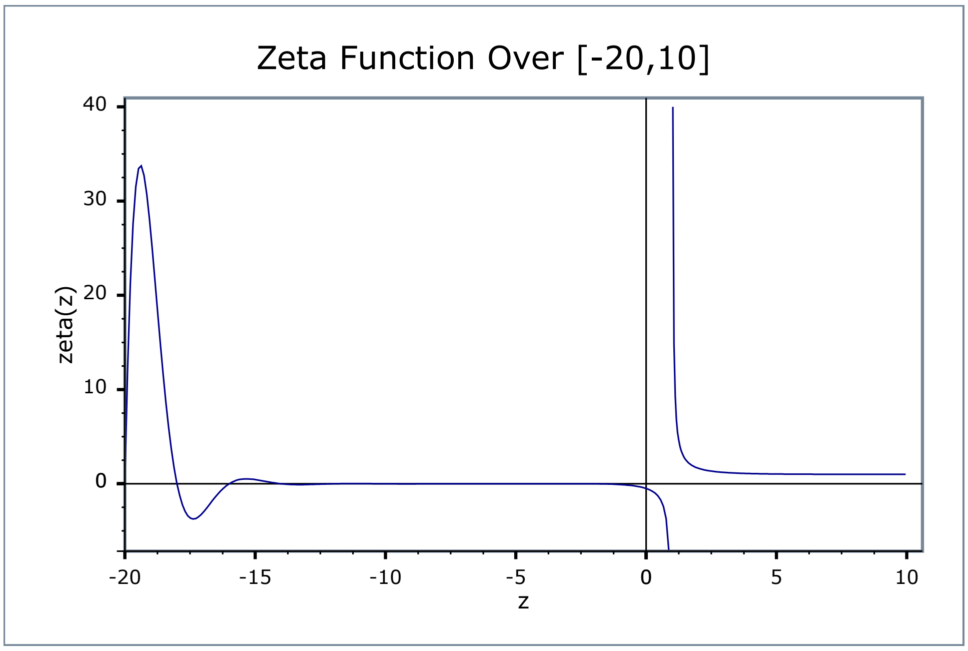
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

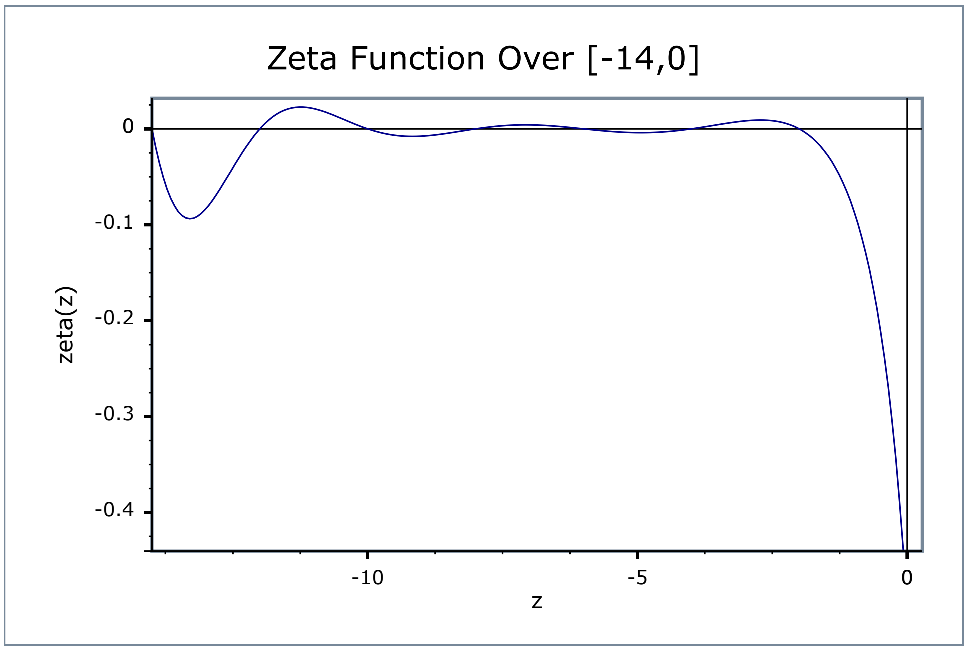
Description

```
template <class T>
calculated-result-type zeta(T z);

template <class T, class Policy>
calculated-result-type zeta(T z, const Policy&);
```

Returns the zeta function of z:

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$$





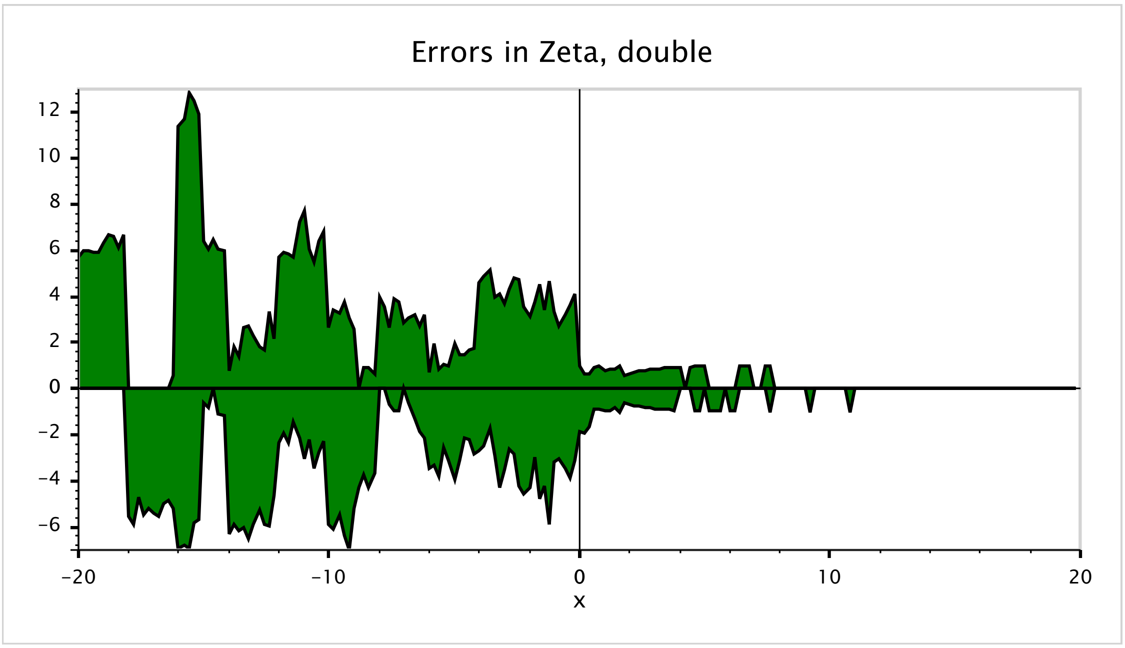
Accuracy

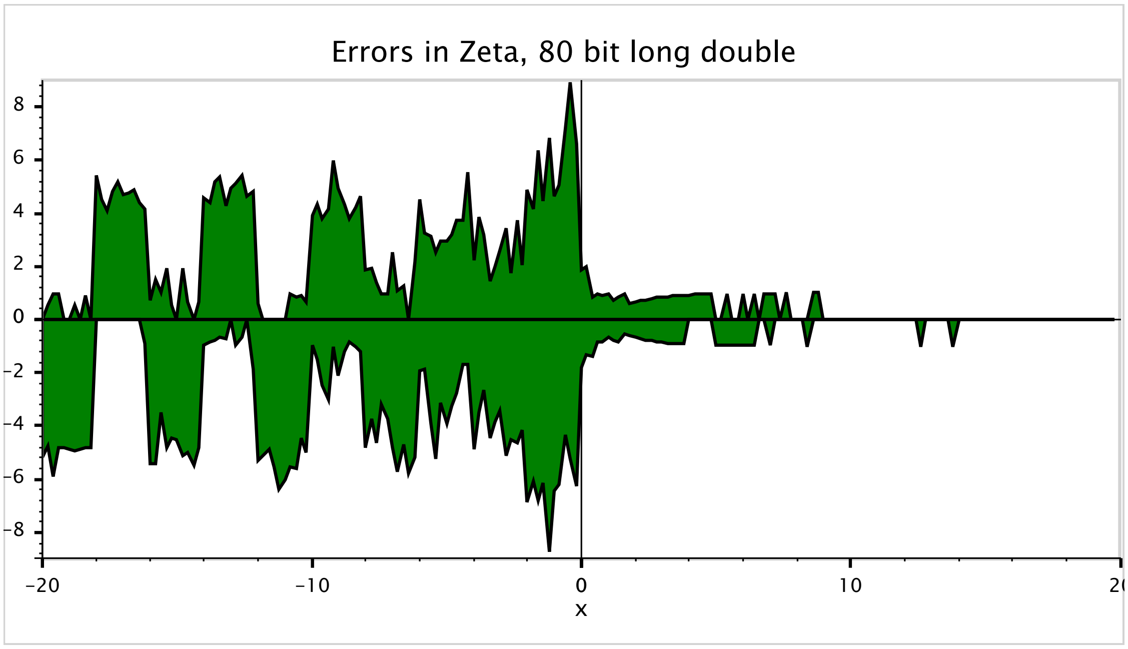
The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to the GSL-1.9 and Cephes libraries. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

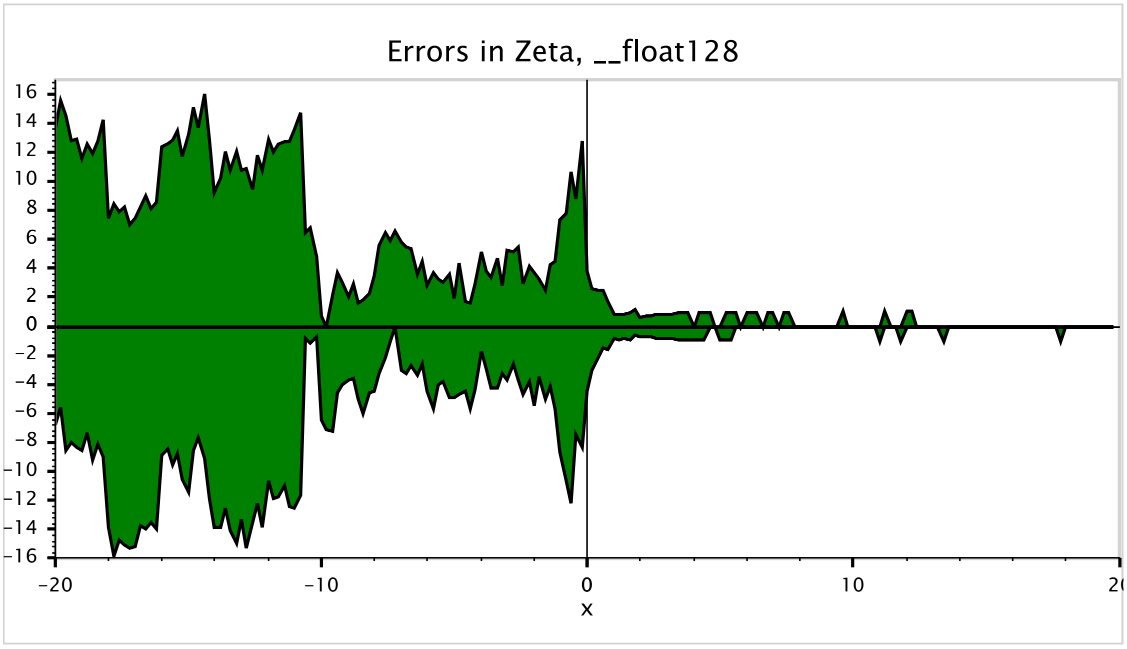
Table 99. Error rates for zeta

	GNU C++ version 7.1.0 **linux** **long double**	**GNU C++ version 7.1.0** **linux** **double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Zeta: Random values greater than 1	Max = 0.846ε (Mean = 0.0833ε) (*<cmath>:* Max = 5.45ε (Mean = 1ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 8.69ε (Mean = 1.03ε))	Max = 0.846ε (Mean = 0.0833ε)	Max = 0.836ε (Mean = 0.093ε)
Zeta: Random values less than 1	Max = 7.03ε (Mean = 2.93ε) (*<cmath>:* Max = 538ε (Mean = 59.3ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 137ε (Mean = 13.8ε))	Max = 70.1ε (Mean = 17.1ε)	Max = 6.84ε (Mean = 3.12ε)
Zeta: Values close to and greater than 1	Max = 0.995ε (Mean = 0.5ε) (*<cmath>:* Max = 1.9e+06ε (Mean = 5.11e+05ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.73ε (Mean = 4.07ε))	Max = 0.995ε (Mean = 0.5ε)	Max = 0.994ε (Mean = 0.421ε)
Zeta: Values close to and less than 1	Max = 0.998ε (Mean = 0.508ε) (*<cmath>:* Max = 8.53e+06ε (Mean = 1.87e+06ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.991ε (Mean = 0.28ε))	Max = 0.998ε (Mean = 0.508ε)	Max = 0.991ε (Mean = 0.375ε)
Zeta: Integer arguments	Max = 9ε (Mean = 3.06ε) (*<cmath>:* Max = 70.3ε (Mean = 17.4ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.75ε (Mean = 1.1ε))	Max = 28ε (Mean = 5.62ε)	Max = 9ε (Mean = 3ε)

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The tests for these functions come in two parts: basic sanity checks use spot values calculated using Mathworld's online evaluator, while accuracy checks use high-precision test values calculated at 1000-bit precision with NTL::RR and this implementation. Note that the generic and type-specific versions of these functions use differing implementations internally, so this gives us reasonably independent test data. Using our test data to test other "known good" implementations also provides an additional sanity check.

Implementation

All versions of these functions first use the usual reflection formulas to make their arguments positive:

$$\zeta(1-s) \quad = \quad 2\sin\left(\pi\frac{1-s}{2}\right)(2\pi^{-s})\Gamma(s)\zeta(s)$$

The generic versions of these functions are implemented using the series:

$$\zeta(s) \quad = \quad \frac{-1}{s^n\left(1-2^{1-s}\right)}\sum_{j=0}^{2n-1}\frac{e_j}{(j+1)^s} \quad + \quad \gamma(s) \quad ; \quad e_j = (-1)^j\left(\sum_{k=0}^{j-n}\frac{n\,!}{k\,!(n-k)\,!} - 2^n\right) \quad \wedge \quad \left|\gamma_n(s)\right| \; < \; =$$

When the significand (mantissa) size is recognised (currently for 53, 64 and 113-bit reals, plus single-precision 24-bit handled via promotion to double) then a series of rational approximations devised by JM are used.

For 0 < z < 1 the approximating form is:

$$\varsigma(s) = \frac{C + R(1-s) - s}{1-s}$$

For a rational approximation R(1-z) and a constant C.

For 1 < z < 4 the approximating form is:

$$\varsigma(s) = C + R(s-n) + \frac{1}{s-1}$$

For a rational approximation R(n-z) and a constant C and integer n.

For z > 4 the approximating form is:

$\zeta(z) = 1 + e^{R(z - n)}$

For a rational approximation R(z-n) and integer n, note that the accuracy required for R(z-n) is not full machine precision, but an absolute error of: ε/R(0). This saves us quite a few digits when dealing with large z, especially when ε is small.

Finally, there are some special cases for integer arguments, there are closed forms for negative or even integers:

$$\zeta(-n) = \frac{(-1)^n}{n+1}B_{n+1} \quad ; \quad n \in \mathbb{N}$$

$$\zeta(-2n) = 0 \quad ; \quad n \in \mathbb{N}$$

$$\zeta(2n) = \frac{(-1)^{n-1}2^{2n-1}\pi^{2n}}{(2n)\,!}B_{2n} \quad ; \quad n \in \mathbb{N}$$

and for positive odd integers we simply cache pre-computed values as these are of great benefit to some infinite series calculations.

Exponential Integrals

Exponential Integral En

Synopsis

```
#include <boost/math/special_functions/expint.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type expint(unsigned n, T z);

template <class T, class Policy>
calculated-result-type expint(unsigned n, T z, const Policy&);

}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: the return type is `double` if T is an integer type, and T otherwise.

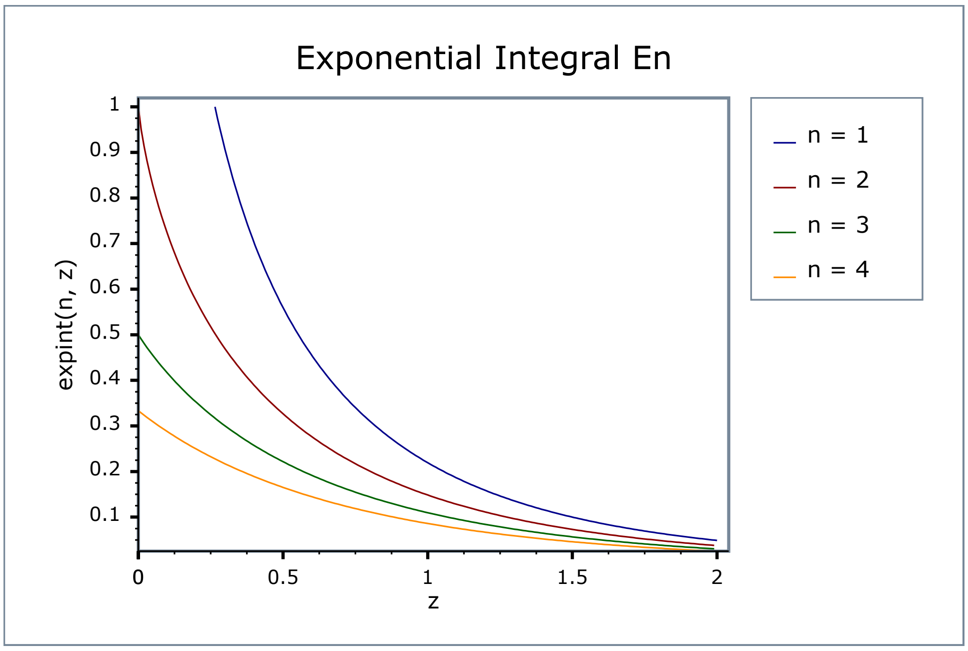
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Description

```
template <class T>
calculated-result-type expint(unsigned n, T z);

template <class T, class Policy>
calculated-result-type expint(unsigned n, T z, const Policy&);
```

Returns the exponential integral En of z:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}dt}{t^n}$$



Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to other libraries. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

Table 100. Error rates for expint (En)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Exponential Integral En	Max = 0.589ε (Mean = 0.0331ε) (*GSL 2.1:* Max = 58.5ε (Mean = 17.1ε))	Max = 9.97ε (Mean = 2.13ε)	Max = 9.97ε (Mean = 2.13ε)	Max = 7.16ε (Mean = 1.85ε)
Exponential Integral En: small z values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 115ε (Mean = 23.6ε))	Max = 1.99ε (Mean = 0.559ε)	Max = 1.99ε (Mean = 0.559ε)	Max = 2.62ε (Mean = 0.531ε)
Exponential Integral E1	Max = 0.556ε (Mean = 0.0625ε) (*GSL 2.1:* Max = 0.988ε (Mean = 0.469ε))	Max = 0.965ε (Mean = 0.414ε)	Max = 0.965ε (Mean = 0.408ε)	Max = 0.988ε (Mean = 0.486ε)

Testing

The tests for these functions come in two parts: basic sanity checks use spot values calculated using Mathworld's online evaluator, while accuracy checks use high-precision test values calculated at 1000-bit precision with NTL::RR and this implementation. Note that the generic and type-specific versions of these functions use differing implementations internally, so this gives us reasonably independent test data. Using our test data to test other "known good" implementations also provides an additional sanity check.

Implementation

The generic version of this function uses the continued fraction:

$$E_n(x) = \cfrac{e^{-x}}{n + x - \cfrac{n}{2 + n + x - \cfrac{2(n+1)}{4 + n + x - \cfrac{3(n+2)}{6 + n + x - \cfrac{4(n+3)}{8 + n + x - \ \dots}}}}}$$

for large *x* and the infinite series:

$$E_n(x) = \frac{(-z)^{n-1}}{(n-1)!}(\psi(n) - \log(z)) - \sum_{k=0, k\neq n-1}^{\infty} \frac{(-1)^k z^k}{(k-n+1)k!}$$

for small *x*.

Where the precision of *x* is known at compile time and is 113 bits or fewer in precision, then rational approximations devised by JM are used for the `n == 1` case.

For `x < 1` the approximating form is a minimax approximation:

$$E_1(x) = x + \log(x) + c + R(x)$$

and for `x > 1` a Chebyshev interpolated approximation of the form:

$$E_1(x) = \frac{e^{-x}}{x}\left(1 + R\left(\frac{1}{x}\right)\right)$$

is used.

Exponential Integral Ei

Synopsis

```
#include <boost/math/special_functions/expint.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type expint(T z);

template <class T, class Policy>
calculated-result-type expint(T z, const Policy&);

}} // namespaces
```

The return type of these functions is computed using the *result type calculation rules*: the return type is `double` if T is an integer type, and T otherwise.

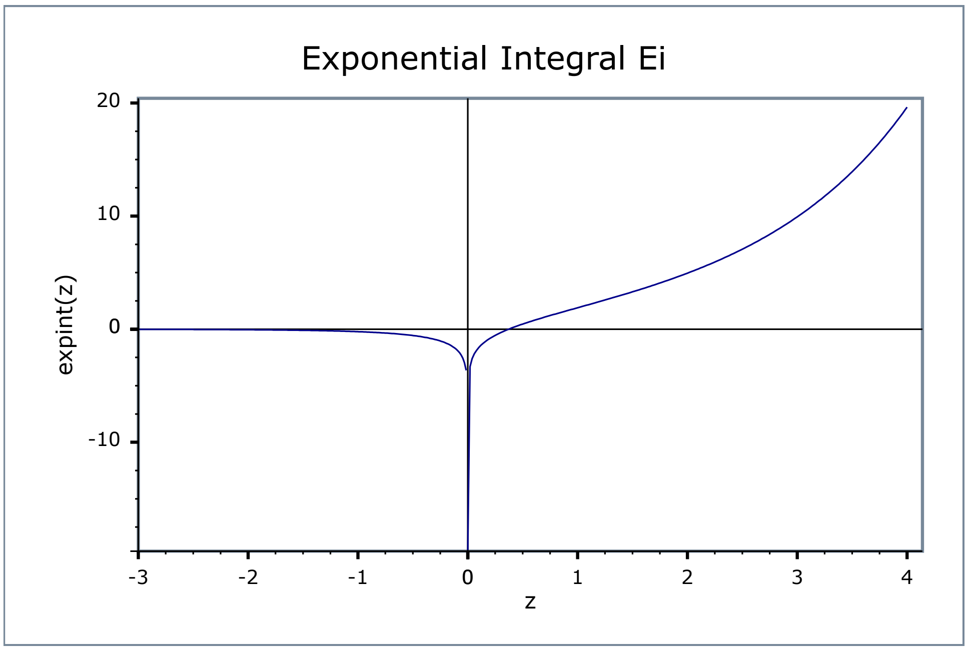
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Description

```
template <class T>
calculated-result-type expint(T z);

template <class T, class Policy>
calculated-result-type expint(T z, const Policy&);
```

Returns the exponential integral of z:

$$Ei(x) = -E_1(-x) = \int_{-x}^{\infty} \frac{e^{-t}dt}{t}$$



Accuracy

The following table shows the peak errors (in units of epsilon) found on various platforms with various floating point types, along with comparisons to Cody's SPECFUN implementation and the GSL-1.9 library. Unless otherwise specified any floating point type that is narrower than the one shown will have effectively zero error.

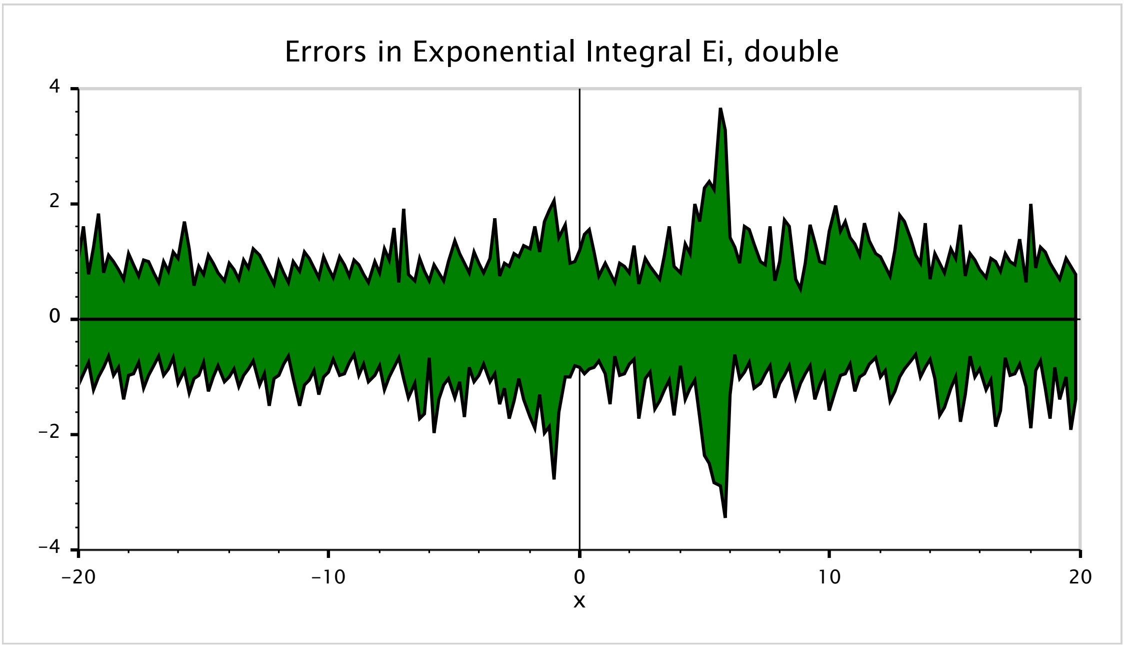
Table 101. Error rates for expint (Ei)

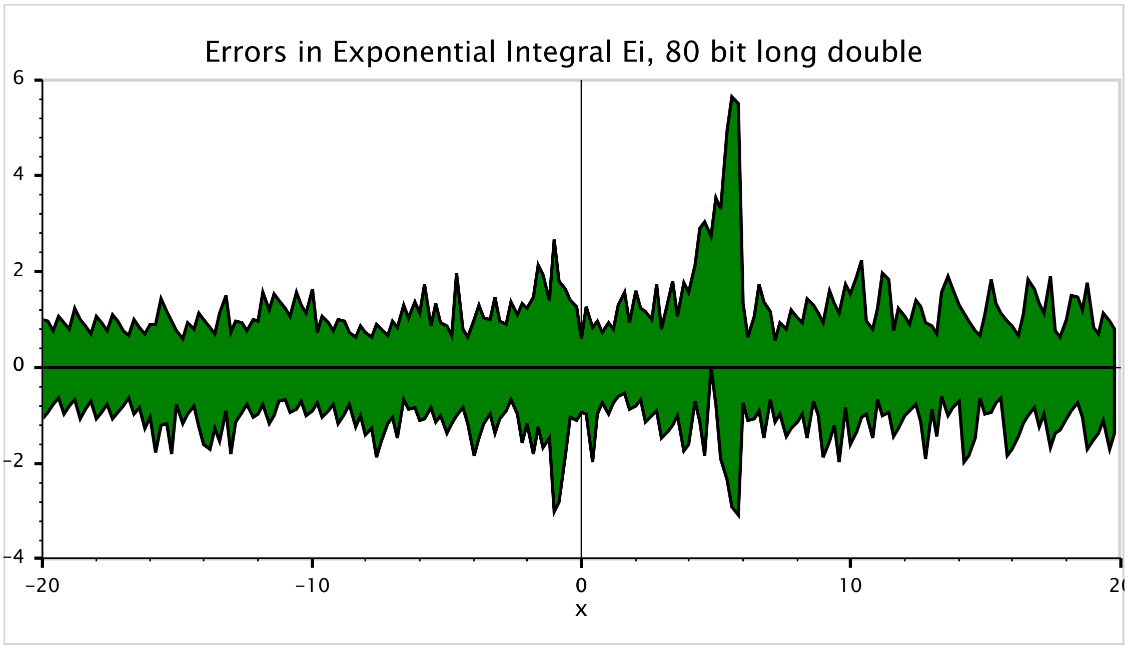
	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Exponential Integral Ei	Max = 5.05ε (Mean = 0.821ε) (*<cmath>:* Max = 14.1ε (Mean = 2.43ε) And other failures.)	Max = 0.994ε (Mean = 0.142ε) (*GSL 2.1:* Max = 8.96ε (Mean = 0.703ε))	Max = 5.05ε (Mean = 0.835ε)	Max = 1.43ε (Mean = 0.54ε)
Exponential Integral Ei: double exponent range	Max = 1.72ε (Mean = 0.593ε) (*<cmath>:* Max = 3.11ε (Mean = 1.13ε))	Max = 0.998ε (Mean = 0.156ε) (*GSL 2.1:* Max = 1.5ε (Mean = 0.612ε))	Max = 1.72ε (Mean = 0.607ε)	Max = 1.7ε (Mean = 0.66ε)
Exponential Integral Ei: long exponent range	Max = 1.98ε (Mean = 0.595ε) (*<cmath>:* Max = 1.93ε (Mean = 0.855ε))		Max = 1.98ε (Mean = 0.575ε)	

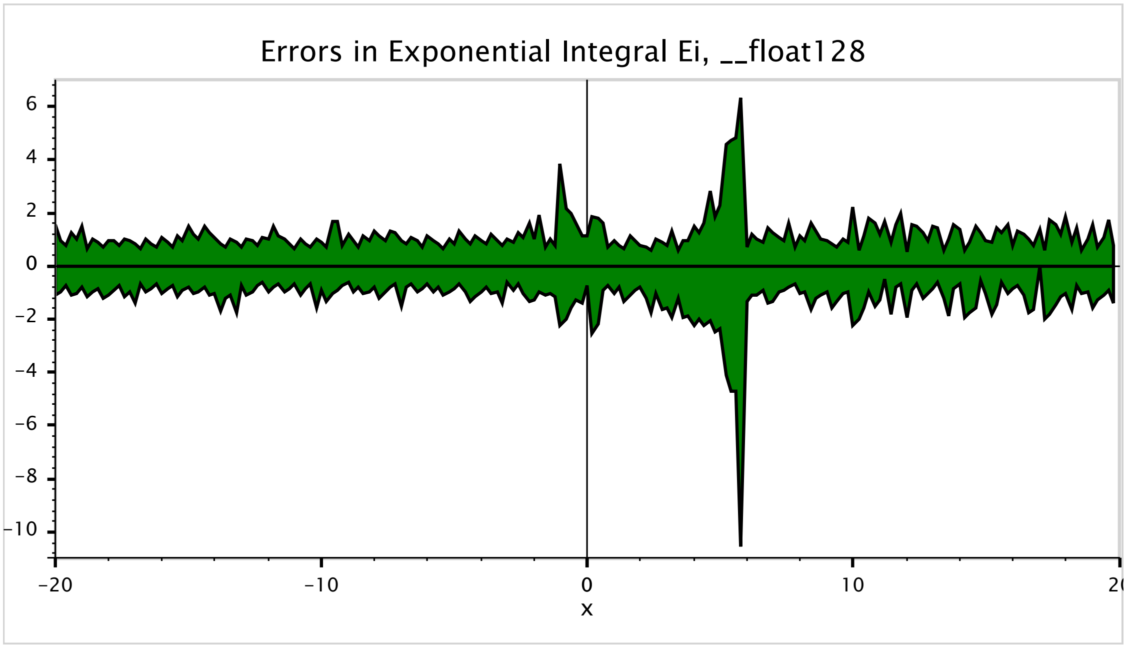
It should be noted that all three libraries tested above offer sub-epsilon precision over most of their range.

GSL has the greatest difficulty near the positive root of En, while Cody's SPECFUN along with this implementation increase their error rates very slightly over the range [4,6].

The following error plot are based on an exhaustive search of the functions domain, MSVC-15.5 at `double` precision, and GCC-7.1/Ubuntu for `long double` and `__float128`.







Testing

The tests for these functions come in two parts: basic sanity checks use spot values calculated using Mathworld's online evaluator, while accuracy checks use high-precision test values calculated at 1000-bit precision with NTL::RR and this implementation. Note that the generic and type-specific versions of these functions use differing implementations internally, so this gives us reasonably independent test data. Using our test data to test other "known good" implementations also provides an additional sanity check.

Implementation

For x < 0 this function just calls zeta(1, -x): which in turn is implemented in terms of rational approximations when the type of x has 113 or fewer bits of precision.

For x > 0 the generic version is implemented using the infinte series:

$$\mathrm{Ei}(z) = \log(z) + \gamma + \sum_{k=0}^{\infty} \frac{z^k}{kk!}$$

However, when the precision of the argument type is known at compile time and is 113 bits or less, then rational approximations devised by JM are used.

For 0 < z < 6 a root-preserving approximation of the form:

$$\mathrm{Ei}(z) = \log\left(\frac{z}{z_0}\right) + (z - z_0)R\left(\frac{z}{3} - 1\right)$$

is used, where z_0 is the positive root of the function, and R(z/3 - 1) is a minimax rational approximation rescaled so that it is evaluated over [-1,1]. Note that while the rational approximation over [0,6] converges rapidly to the minimax solution it is rather ill-conditioned in practice. Cody and Thacher [5] experienced the same issue and converted the polynomials into Chebeshev form to ensure stable computation. By experiment we found that the polynomials are just as stable in polynomial as Chebyshev form, *provided* they are computed over the interval [-1,1].

Over the a series of intervals [a,b] and [b,INF] the rational approximation takes the form:

$$\mathrm{Ei}(z) = z + \frac{e^z}{z}(c + R(t))$$

where *c* is a constant, and R(t) is a minimax solution optimised for low absolute error compared to *c*. Variable *t* is `1/z` when the range in infinite and `2z/(b-a) - (2a/(b-a) + 1)` otherwise: this has the effect of scaling z to the interval [-1,1]. As before rational approximations over arbitrary intervals were found to be ill-conditioned: Cody and Thacher solved this issue by converting the polynomials to their J-Fraction equivalent. However, as long as the interval of evaluation was [-1,1] and the number of terms carefully chosen, it was found that the polynomials *could* be evaluated to suitable precision: error rates are typically 2 to 3 epsilon which is comparible to the error rate that Cody and Thacher achieved using J-Fractions, but marginally more efficient given that fewer divisions are involved.

[5] W. J. Cody and H. C. Thacher, Jr., Rational Chebyshev approximations for the exponential integral $E_1(x)$, Math. Comp. 22 (1968), 641-649, and W. J. Cody and H. C. Thacher, Jr., Chebyshev approximations for the exponential integral Ei(x), Math. Comp. 23 (1969), 289-303.

Basic Functions

sin_pi

```
#include <boost/math/special_functions/sin_pi.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type sin_pi(T x);

template <class T, class Policy>
calculated-result-type sin_pi(T x, const Policy&);

}} // namespaces
```

Returns the sine of πx.

The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function performs exact all-integer arithmetic argument reduction before computing the sine of πx.

Table 102. Error rates for sin_pi

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
sin_pi and cos_pi	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.335ε)	Max = 0.996ε (Mean = 0.336ε)	Max = 0.99ε (Mean = 0.328ε)
sin_pi and cos_pi near integers and half integers	Max = 0ε (Mean = 0ε)	Max = 0.976ε (Mean = 0.293ε)	Max = 0.976ε (Mean = 0.293ε)	Max = 0.996ε (Mean = 0.343ε)

cos_pi

```
#include <boost/math/special_functions/cos_pi.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type cos_pi(T x);

template <class T, class Policy>
calculated-result-type cos_pi(T x, const Policy&);

}} // namespaces
```

Returns the cosine of πx.

The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function performs exact all-integer arithmetic argument reduction before computing the cosine of πx.

Table 103. Error rates for cos_pi

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
sin_pi and cos_pi	Max = 0ε (Mean = 0ε)	Max = 0.991ε (Mean = 0.302ε)	Max = 0.991ε (Mean = 0.302ε)	Max = 0.996ε (Mean = 0.284ε)
sin_pi and cos_pi near integers and half integers	Max = 0ε (Mean = 0ε)	Max = 0.976ε (Mean = 0.28ε)	Max = 0.976ε (Mean = 0.28ε)	Max = 0.996ε (Mean = 0.298ε)

log1p

```
#include <boost/math/special_functions/log1p.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type log1p(T x);

template <class T, class Policy>
calculated-result-type log1p(T x, const Policy&);

}} // namespaces
```

Returns the natural logarithm of `x+1`.

The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

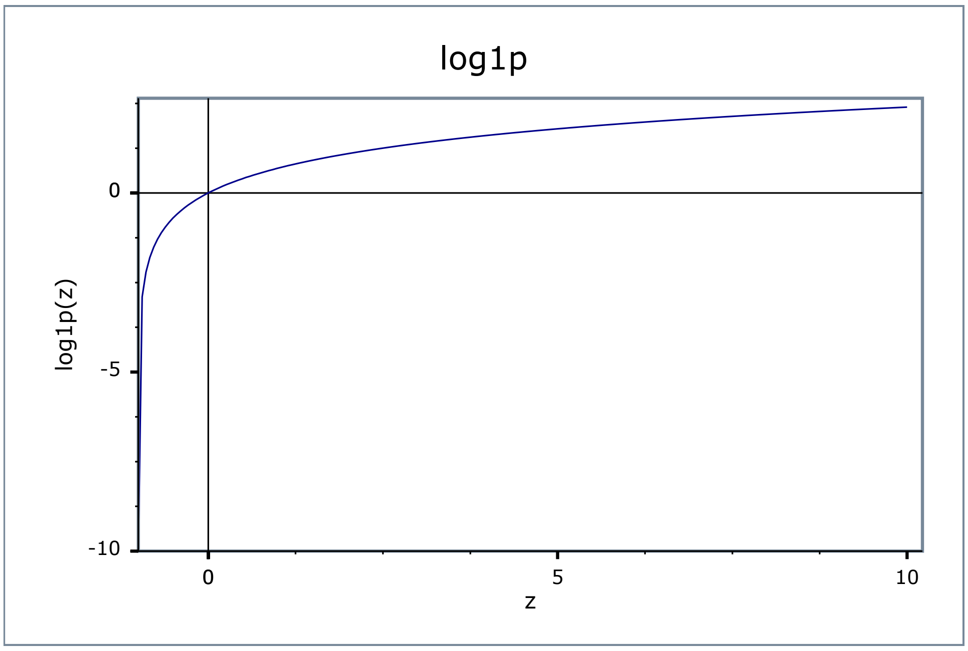
There are many situations where it is desirable to compute `log(x+1)`. However, for small `x` then `x+1` suffers from catastrophic cancellation errors so that `x+1 == 1` and `log(x+1) == 0`, when in fact for very small x, the best approximation to `log(x+1)` would be `x`. `log1p` calculates the best approximation to `log(1+x)` using a Taylor series expansion for accuracy (less than 2). Alternatively note that there are faster methods available, for example using the equivalence:

```
log(1+x) == (log(1+x) * x) / ((1+x) - 1)
```

However, experience has shown that these methods tend to fail quite spectacularly once the compiler's optimizations are turned on, consequently they are used only when known not to break with a particular compiler. In contrast, the series expansion method seems to be reasonably immune to optimizer-induced errors.

Finally when BOOST_HAS_LOG1P is defined then the `float/double/long double` specializations of this template simply forward to the platform's native (POSIX) implementation of this function.

The following graph illustrates the behaviour of log1p:



Accuracy

For built in floating point types `log1p` should have approximately 1 epsilon accuracy.

Table 104. Error rates for log1p

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Random test data	Max = 0.818ε (Mean = 0.227ε) (*<cmath>:* Max = 0.818ε (Mean = 0.227ε)) (*<math.h>:* Max = 0.818ε (Mean = 0.227ε))	Max = 0.846ε (Mean = 0.153ε) (*Rmath 3.2.3:* Max = 0.846ε (Mean = 0.153ε))	Max = 2.3ε (Mean = 0.66ε) (*<math.h>:* Max = 0.818ε (Mean = 0.249ε))	Max = 0.509ε (Mean = 0.057ε) (*<math.h>:* Max = 0.509ε (Mean = 0.057ε))

Testing

A mixture of spot test sanity checks, and random high precision test values calculated using NTL::RR at 1000-bit precision.

expm1

```
#include <boost/math/special_functions/expm1.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type expm1(T x);

template <class T, class Policy>
calculated-result-type expm1(T x, const Policy&);

}} // namespaces
```

Returns e^x - 1.

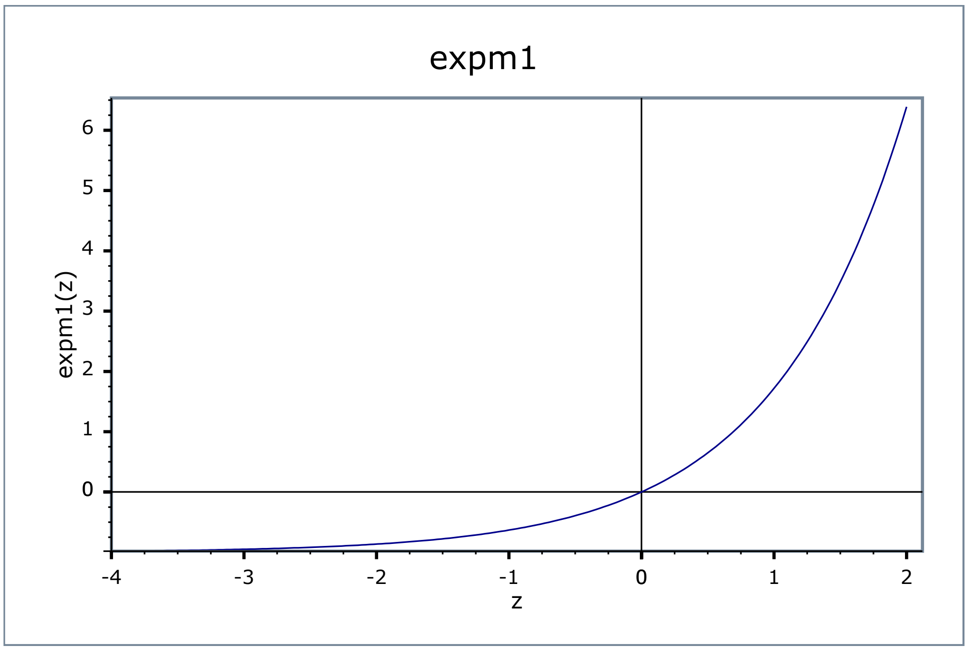
The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

For small x, then e^x is very close to 1, as a result calculating `e`^{x} ` - 1` results in catastrophic cancellation errors when x is small. `expm1` calculates `e`^{x} ` - 1` using rational approximations (for up to 128-bit long doubles), otherwise via a series expansion when x is small (giving an accuracy of less than 2).

Finally when BOOST_HAS_EXPM1 is defined then the `float/double/long double` specializations of this template simply forward to the platform's native (POSIX) implementation of this function.

The following graph illustrates the behaviour of expm1:



Accuracy

For built in floating point types `expm1` should have approximately 1 epsilon accuracy.

Table 105. Error rates for expm1

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Random test data	Max = 0.992ε (Mean = 0.402ε) (*<cmath>:* Max = 0.992ε (Mean = 0.402ε)) (*<math.h>:* Max = 0.992ε (Mean = 0.402ε))	Max = 0.793ε (Mean = 0.126ε) (*Rmath 3.2.3:* Max = 0.793ε (Mean = 0.126ε))	Max = 1.31ε (Mean = 0.428ε) (*<math.h>:* Max = 0.996ε (Mean = 0.426ε))	Max = 1.31ε (Mean = 0.496ε) (*<math.h>:* Max = 1.31ε (Mean = 0.496ε))

Testing

A mixture of spot test sanity checks, and random high precision test values calculated using NTL::RR at 1000-bit precision.

cbrt

```
#include <boost/math/special_functions/cbrt.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type cbrt(T x);

template <class T, class Policy>
calculated-result-type cbrt(T x, const Policy&);

}} // namespaces
```

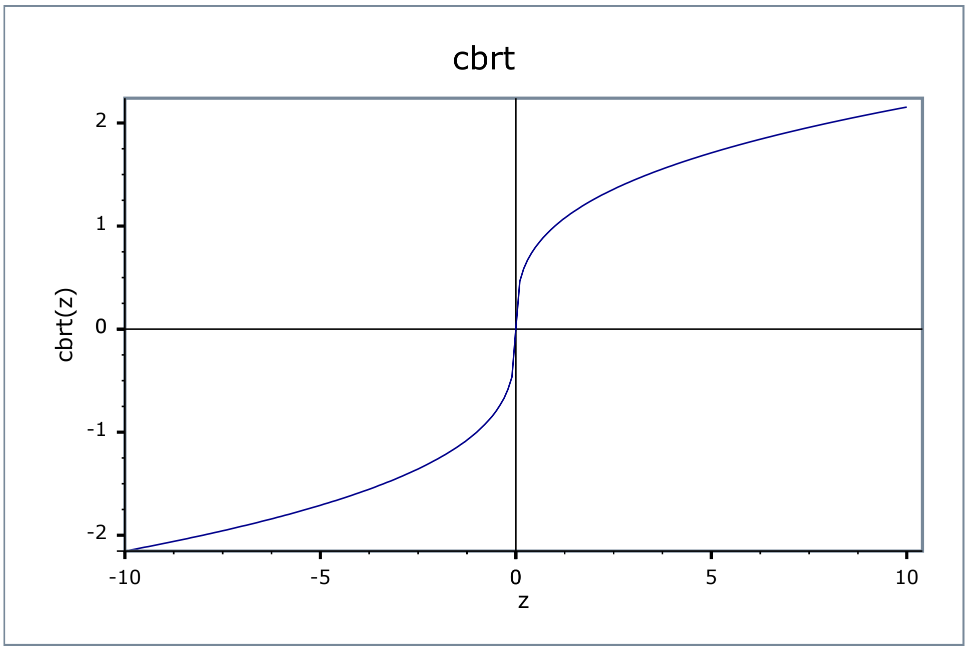
Returns the cubed root of x: $x^{1/3}$.

The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Implemented using Halley iteration.

The following graph illustrates the behaviour of cbrt:



Accuracy

For built in floating-point types `cbrt` should have approximately 2 epsilon accuracy.

Table 106. Error rates for cbrt

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
cbrt Function	Max = 0ε (Mean = 0ε)	Max = 1.34ε (Mean = 0.471ε) (*<cmath>:* Max = 1.34ε (Mean = 0.471ε)) (*<math.h>:* Max = 1.34ε (Mean = 0.471ε))	Max = 1.34ε (Mean = 0.471ε) (*<math.h>:* Max = 1.34ε (Mean = 0.471ε))	Max = 1.7ε (Mean = 0.565ε) (*<math.h>:* Max = 1.7ε (Mean = 0.565ε))

Testing

A mixture of spot test sanity checks, and random high precision test values calculated using NTL::RR at 1000-bit precision.

sqrt1pm1

```
#include <boost/math/special_functions/sqrt1pm1.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type sqrt1pm1(T x);

template <class T, class Policy>
calculated-result-type sqrt1pm1(T x, const Policy&);

}} // namespaces
```

Returns `sqrt(1+x) - 1`.

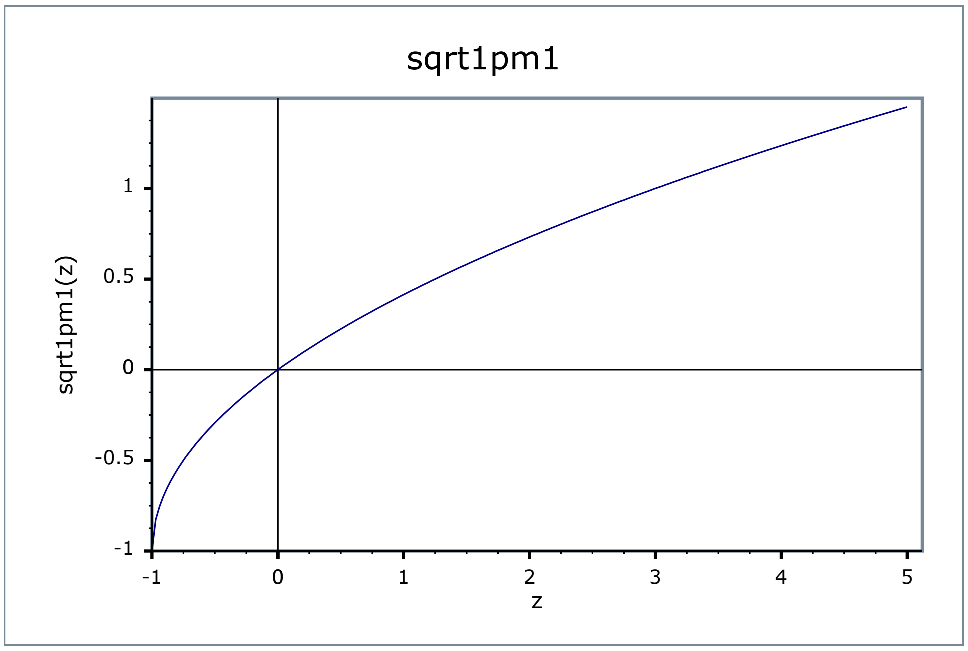
The return type of this function is computed using the *result type calculation rules*: the return is `double` when *x* is an integer type and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

This function is useful when you need the difference between sqrt(x) and 1, when x is itself close to 1.

Implemented in terms of `log1p` and `expm1`.

The following graph illustrates the behaviour of sqrt1pm1:



Accuracy

For built in floating-point types `sqrt1pm1` should have approximately 3 epsilon accuracy.

Table 107. Error rates for sqrt1pm1

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
sqrt1pm1	Max = 1.3ε (Mean = 0.404ε)	Max = 1.33ε (Mean = 0.404ε)	Max = 1.54ε (Mean = 0.563ε)	Max = 1.35ε (Mean = 0.497ε)

Testing

A selection of random high precision test values calculated using NTL::RR at 1000-bit precision.

powm1

```
#include <boost/math/special_functions/powm1.hpp>
```

```
namespace boost{ namespace math{

template <class T1, class T2>
calculated-result-type powm1(T1 x, T2 y);

template <class T1, class T2, class Policy>
calculated-result-type powm1(T1 x, T2 y, const Policy&);

}} // namespaces
```

Returns x^y - 1.

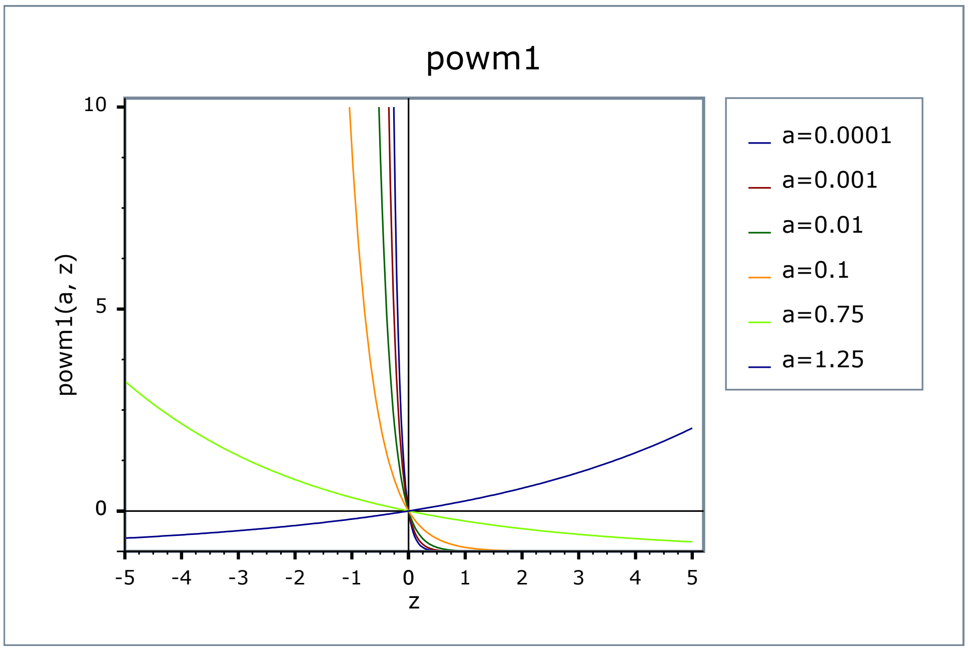
The return type of this function is computed using the *result type calculation rules* when T1 and T2 are dufferent types.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

There are two domains where this is useful: when y is very small, or when x is close to 1.

Implemented in terms of `expm1`.

The following graph illustrates the behaviour of powm1:



Accuracy

Should have approximately 2-3 epsilon accuracy.

Table 108. Error rates for powm1

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
powm1	Max = 1.06ε (Mean = 0.425ε)	Max = 2.04ε (Mean = 0.493ε)	Max = 1.88ε (Mean = 0.49ε)	Max = 1.84ε (Mean = 0.486ε)

Testing

A selection of random high precision test values calculated using NTL::RR at 1000-bit precision.

hypot

```
template <class T1, class T2>
calculated-result-type hypot(T1 x, T2 y);

template <class T1, class T2, class Policy>
calculated-result-type hypot(T1 x, T2 y, const Policy&);
```

Effects: computes

$$\mathrm{hypot}(x,\ y) \quad = \quad \sqrt{x^2 + y^2}$$

in such a way as to avoid undue underflow and overflow.

The return type of this function is computed using the *result type calculation rules* when T1 and T2 are of different types.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

When calculating

$$\mathrm{hypot}(x,\ y) \quad = \quad \sqrt{x^2 + y^2}$$

it's quite easy for the intermediate terms to either overflow or underflow, even though the result is in fact perfectly representable.

Implementation

The function is even and symmetric in x and y, so first take assume *x,y > 0* and *x > y* (we can permute the arguments if this is not the case).

Then if $x * \varepsilon >= y$ we can simply return *x*.

Otherwise the result is given by:

$$\mathrm{hypot}(x,\ y) \quad = \quad x\sqrt{1 + \left(\frac{y}{x}\right)^2}$$

Compile Time Power of a Runtime Base

The `pow` function effectively computes the compile-time integral power of a run-time base.

Synopsis

`#include <boost/math/special_functions/pow.hpp>`

```
namespace boost { namespace math {

template <int N, typename T>
calculated-result-type pow(T base);

template <int N, typename T, class Policy>
calculated-result-type pow(T base, const Policy& policy);

}}
```

Rationale and Usage

Computing the power of a number with an exponent that is known at compile time is a common need for programmers. In such cases, the usual method is to avoid the overhead implied by the `pow`, `powf` and `powl` C functions by hardcoding an expression such as:

```
// Hand-written 8th power of a 'base' variable
double result = base*base*base*base*base*base*base*base;
```

However, this kind of expression is not really readable (knowing the value of the exponent involves counting the number of occurrences of *base*), error-prone (it's easy to forget an occurrence), syntactically bulky, and non-optimal in terms of performance.

The pow function of Boost.Math helps writing this kind expression along with solving all the problems listed above:

```
// 8th power of a 'base' variable using math::pow
double result = pow<8>(base);
```

The expression is now shorter, easier to read, safer, and even faster. Indeed, `pow` will compute the expression such that only log2(N) products are made for a power of N. For instance in the example above, the resulting expression will be the same as if we had written this, with only one computation of each identical subexpression:

```
// Internal effect of pow<8>(base)
double result = ((base*base)*(base*base))*((base*base)*(base*base));
```

Only 3 different products were actually computed.

Return Type

The return type of these functions is computed using the *result type calculation rules*. For example:

- If T is a `float`, the return type is a `float`.
- If T is a `long double`, the return type is a `long double`.
- Otherwise, the return type is a `double`.

Policies

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Error Handling

Two cases of errors can occur when using `pow`:

- In case of null base and negative exponent, an overflow_error occurs since this operation is a division by 0 (it equals to 1/0).
- In case of null base and null exponent, an indeterminate_result_error occurs since the result of this operation is indeterminate. Those errors follow the general policies of error handling in Boost.Math.

The default overflow error policy is `throw_on_error`. A call like `pow<-2>(0)` will thus throw a `std::overflow_error` exception. As shown in the link given above, other error handling policies can be used:

- `errno_on_error`: Sets `::errno` to `ERANGE` and returns `std::numeric_limits<T>::infinity()`.
- `ignore_error`: Returns `std::numeric_limits<T>::infinity()`.
- `user_error`: Returns the result of `boost::math::policies::user_overflow_error`: this function must be defined by the user.

The default indeterminate result error policy is `ignore_error`, which for this function returns 1 since it's the most commonly chosen result for a power of 0. Here again, other error handling policies can be used:

- `throw_on_error`: Throws `std::domain_error`
- `errno_on_error`: Sets `::errno` to `EDOM` and returns 1.
- `user_error`: Returns the result of `boost::math::policies::user_indeterminate_result_error`: this function must be defined by the user.

Here is an example of error handling customization where we want to specify the result that has to be returned in case of error. We will thus use the `user_error` policy, by passing as second argument an instance of an overflow_error policy templated with `user_error`:

```
// First we open the boost::math::policies namespace and define the `user_overflow_error`
// by making it return the value we want in case of error (-1 here)

namespace boost { namespace math { namespace policies {
template <class T>
T user_overflow_error(const char*, const char*, const T&)
{ return -1; }
}}}


// Then we invoke pow and indicate that we want to use the user_error policy
using boost::math::policies;
double result = pow<-5>(base, policy<overflow_error<user_error> >());

// We can now test the returned value and treat the special case if needed:
if (result == -1)
{
    // there was an error, do something...
}
```

Another way is to redefine the default `overflow_error` policy by using the BOOST_MATH_OVERFLOW_ERROR_POLICY macro. Once the `user_overflow_error` function is defined as above, we can achieve the same result like this:

```
// Redefine the default error_overflow policy
#define BOOST_MATH_OVERFLOW_ERROR_POLICY user_error
#include <boost/math/special_functions/pow.hpp>

// From this point, passing a policy in argument is no longer needed, a call like this one
// will return -1 in case of error:

double result = pow<-5>(base);
```

Acknowledgements

Bruno Lalande submitted this addition to Boost.Math.

Thanks to Joaquín López Muñoz and Scott McMurray for their help in improving the implementation.

References

D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1981

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions Overview

The Sinus Cardinal family of functions (indexed by the family of indices `a > 0`) is defined by

$$\operatorname{sinc}_a(x) = \frac{\sin\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$$

it sees heavy use in signal processing tasks.

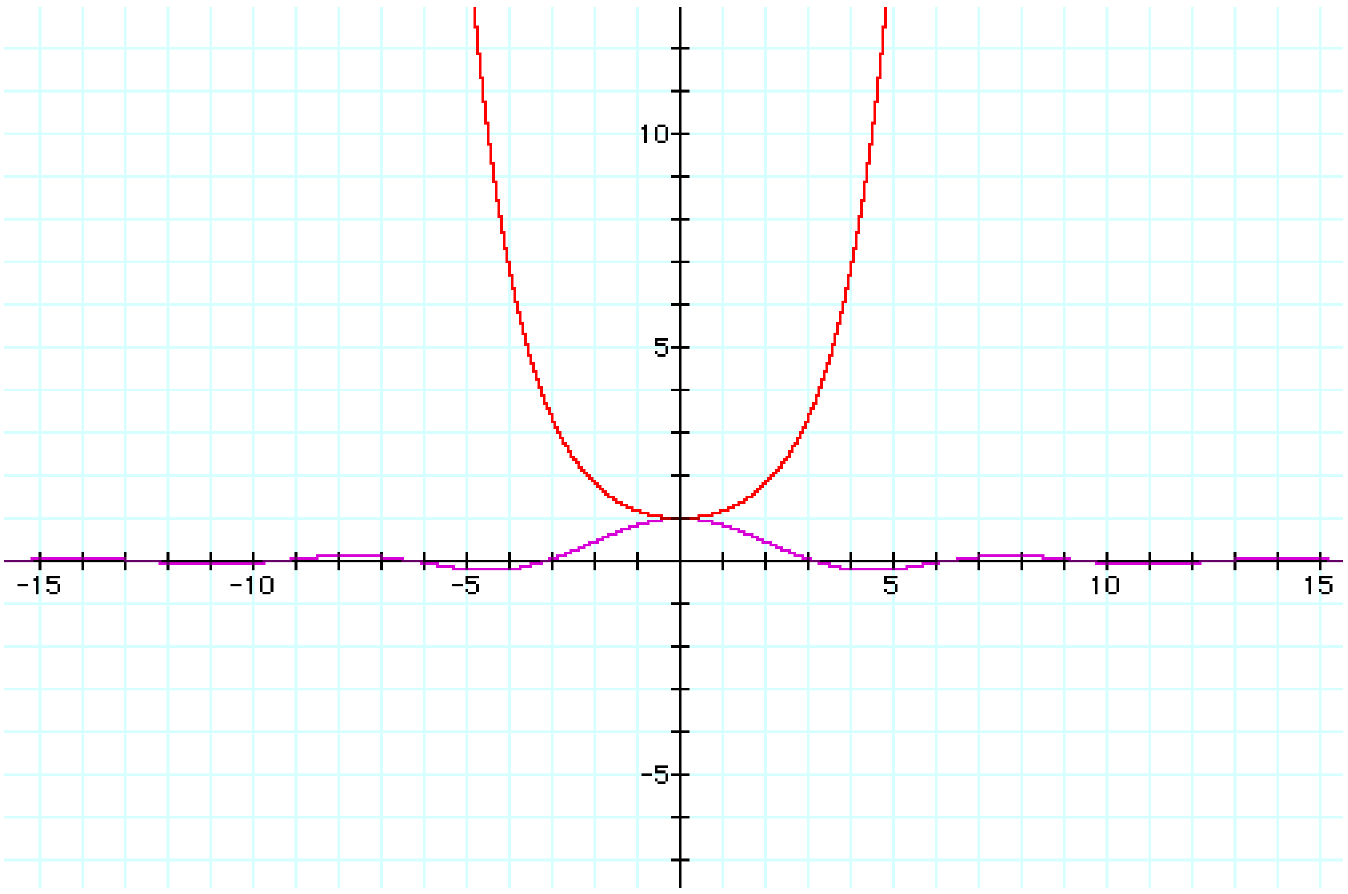
By analogy, the Hyperbolic Sinus Cardinal family of functions (also indexed by the family of indices `a > 0`) is defined by

$$\operatorname{sinhc}_a(x) = \frac{\sinh\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$$

These two families of functions are composed of entire functions.

These functions (sinc_pi and sinhc_pi) are needed by our implementation of quaternions and octonions.

Sinus Cardinal of index pi (purple) and Hyperbolic Sinus Cardinal of index pi (red) on R



sinc_pi

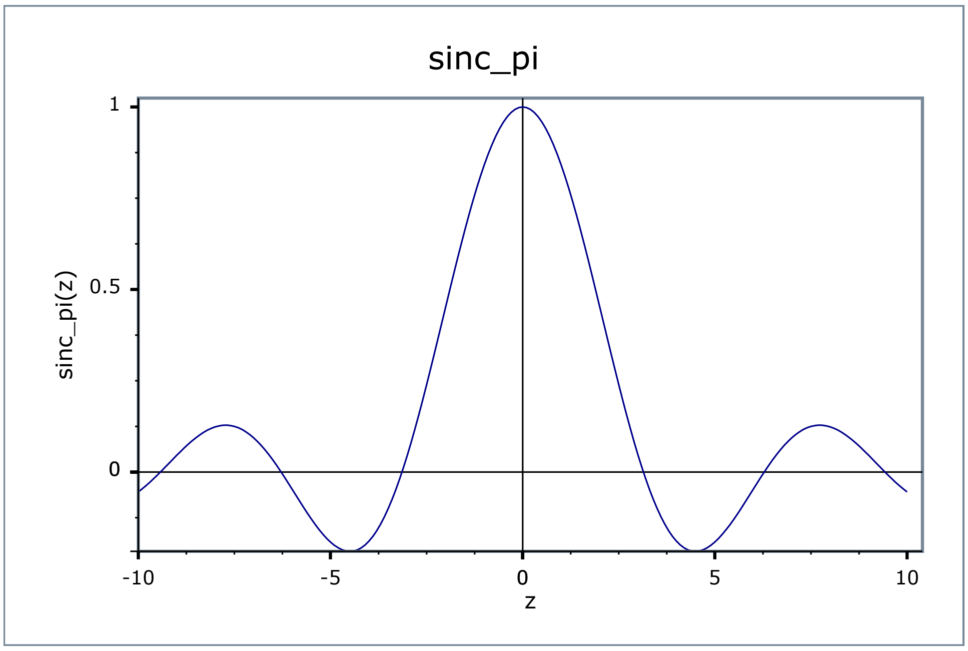
```
#include <boost/math/special_functions/sinc.hpp>
```

```
template<class T>
calculated-result-type sinc_pi(const T x);

template<class T, class Policy>
calculated-result-type sinc_pi(const T x, const Policy&);

template<class T, template<typename> class U>
U<T> sinc_pi(const U<T> x);

template<class T, template<typename> class U, class Policy>
U<T> sinc_pi(const U<T> x, const Policy&);
```

Computes the Sinus Cardinal of x:

```
sinc_pi(x) = sin(x) / x
```

The second form is for complex numbers, quaternions, octonions etc. Taylor series are used at the origin to ensure accuracy.



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

sinhc_pi

```
#include <boost/math/special_functions/sinhc.hpp>
```

```
template<class T>
calculated-result-type sinhc_pi(const T x);

template<class T, class Policy>
calculated-result-type sinhc_pi(const T x, const Policy&);

template<typename T, template<typename> class U>
U<T> sinhc_pi(const U<T> x);

template<class T, template<typename> class U, class Policy>
U<T> sinhc_pi(const U<T> x, const Policy&);
```

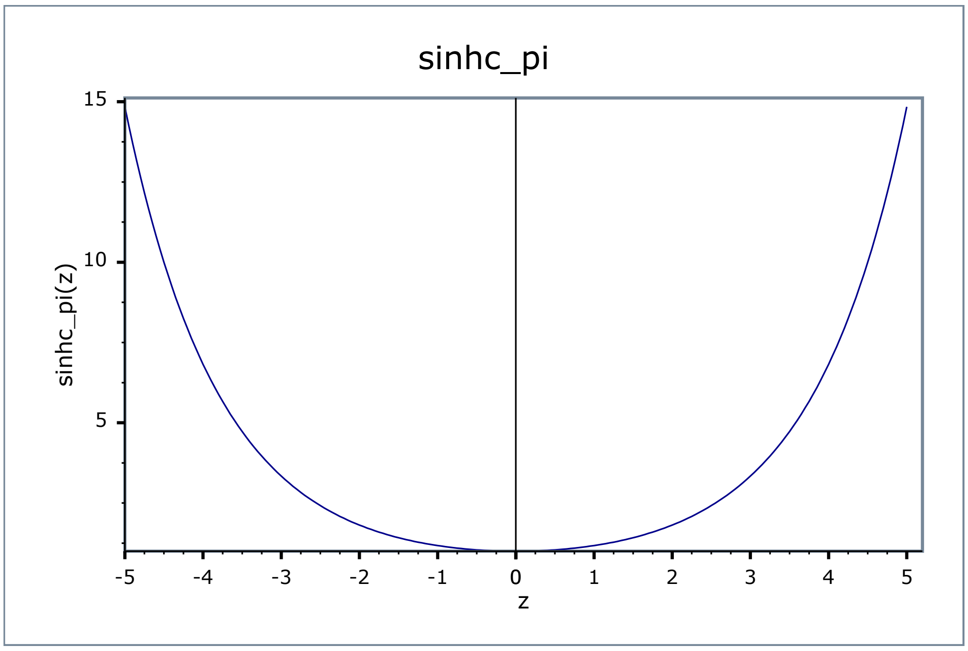
Computes http://mathworld.wolfram.com/SinhcFunction.html the Hyperbolic Sinus Cardinal of x:

```
sinhc_pi(x) = sinh(x) / x
```

The second form is for complex numbers, quaternions, octonions etc. Taylor series are used at the origin to ensure accuracy.

The return type of the first form is computed using the *result type calculation rules* when T is an integer type.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.



Inverse Hyperbolic Functions

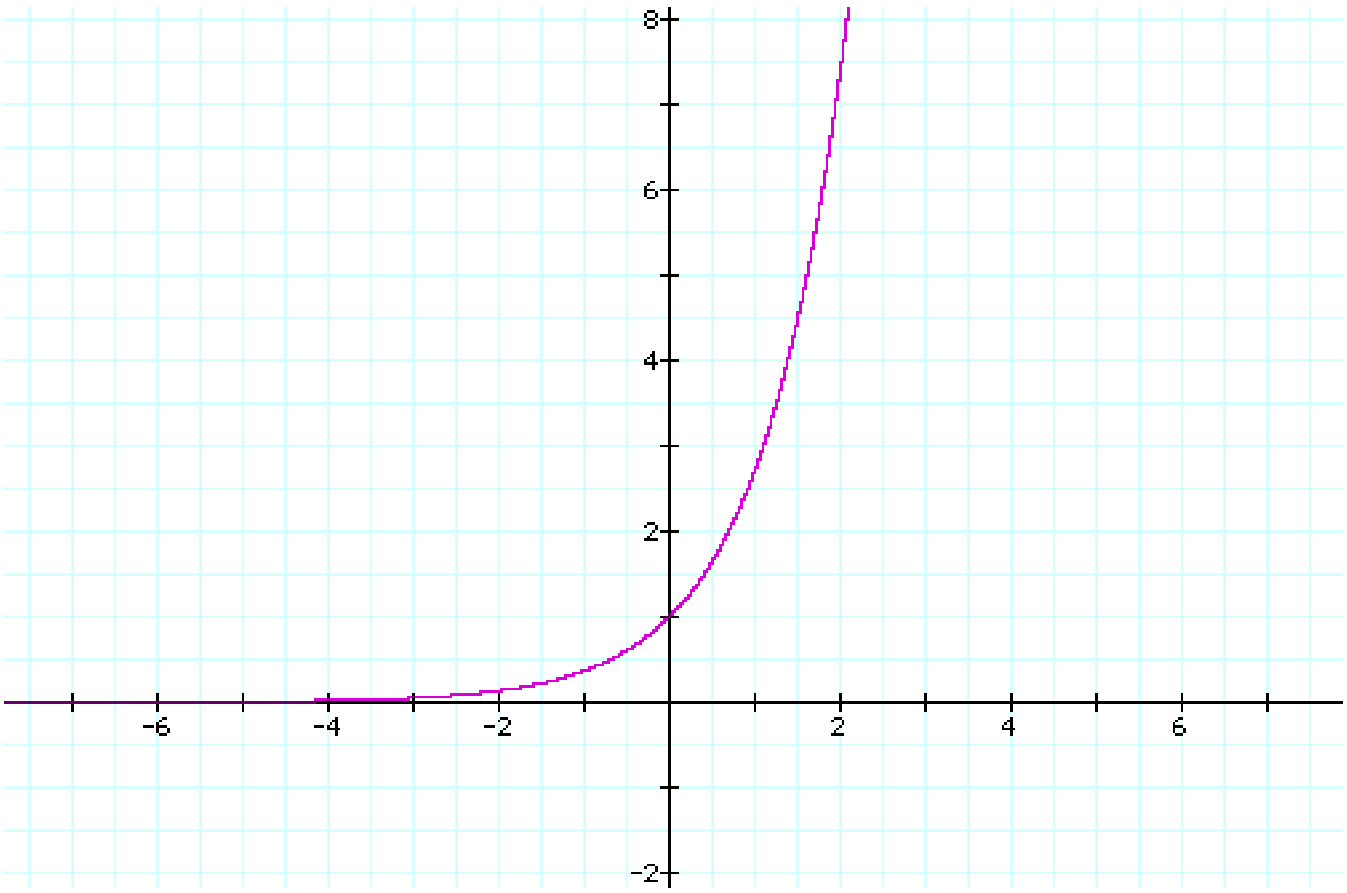
Inverse Hyperbolic Functions Overview

The exponential funtion is defined, for all objects for which this makes sense, as the power series

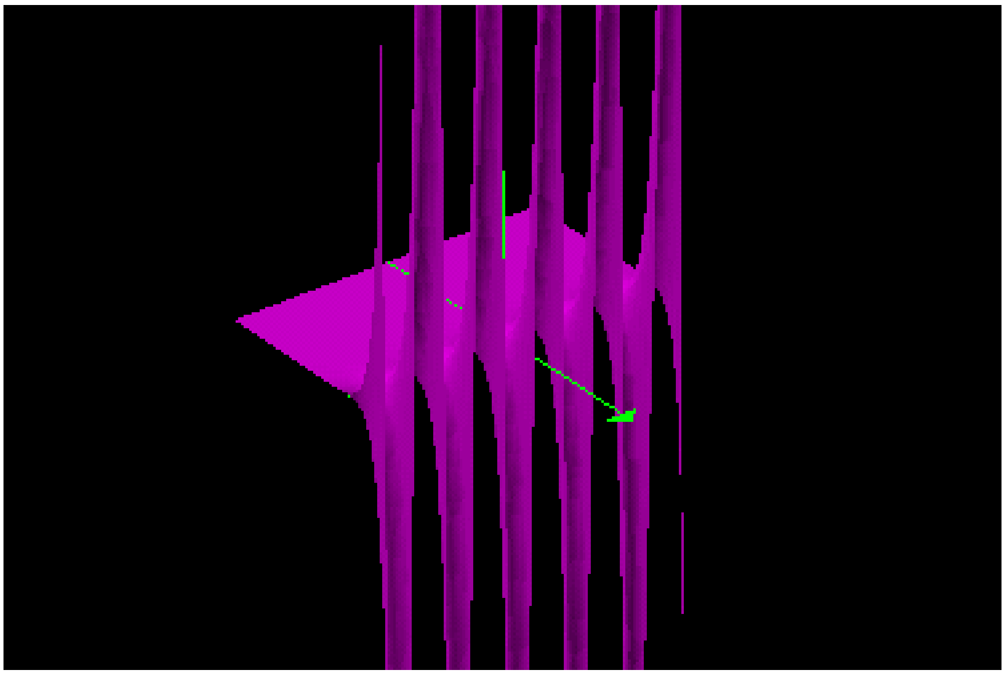
$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

, with `n! = 1x2x3x4x5...xn` (and `0! = 1` by definition) being the factorial of `n`. In particular, the exponential function is well defined for real numbers, complex number, quaternions, octonions, and matrices of complex numbers, among others.

Graph of exp on R



Real and Imaginary parts of exp on C



The hyperbolic functions are defined as power series which can be computed (for reals, complex, quaternions and octonions) as:

Hyperbolic cosine:

$$\cosh(x) = \frac{\exp(x) + \exp(-x)}{2}$$

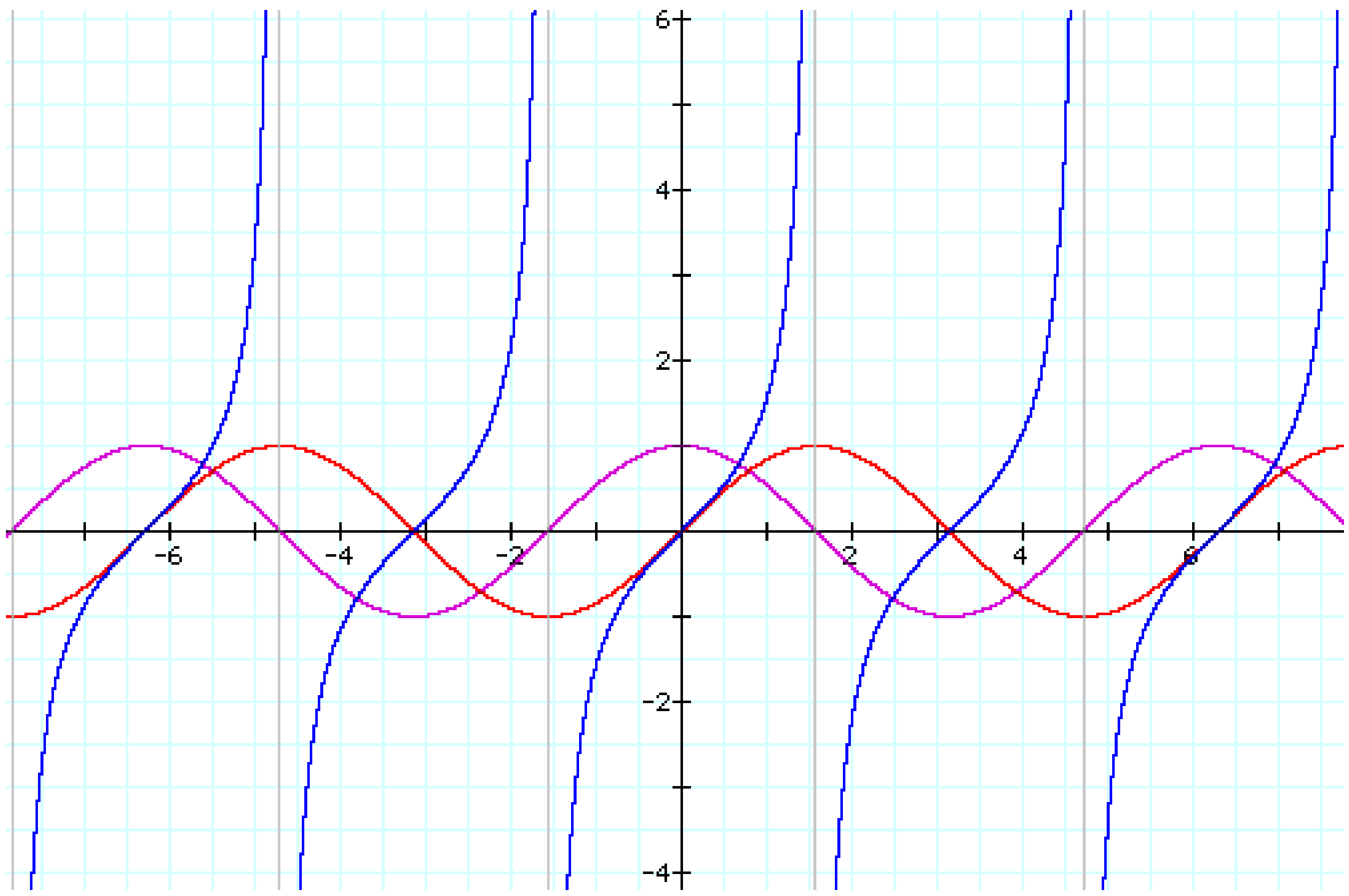
Hyperbolic sine:

$$\sinh(x) = \frac{\exp(x) - \exp(-x)}{2}$$

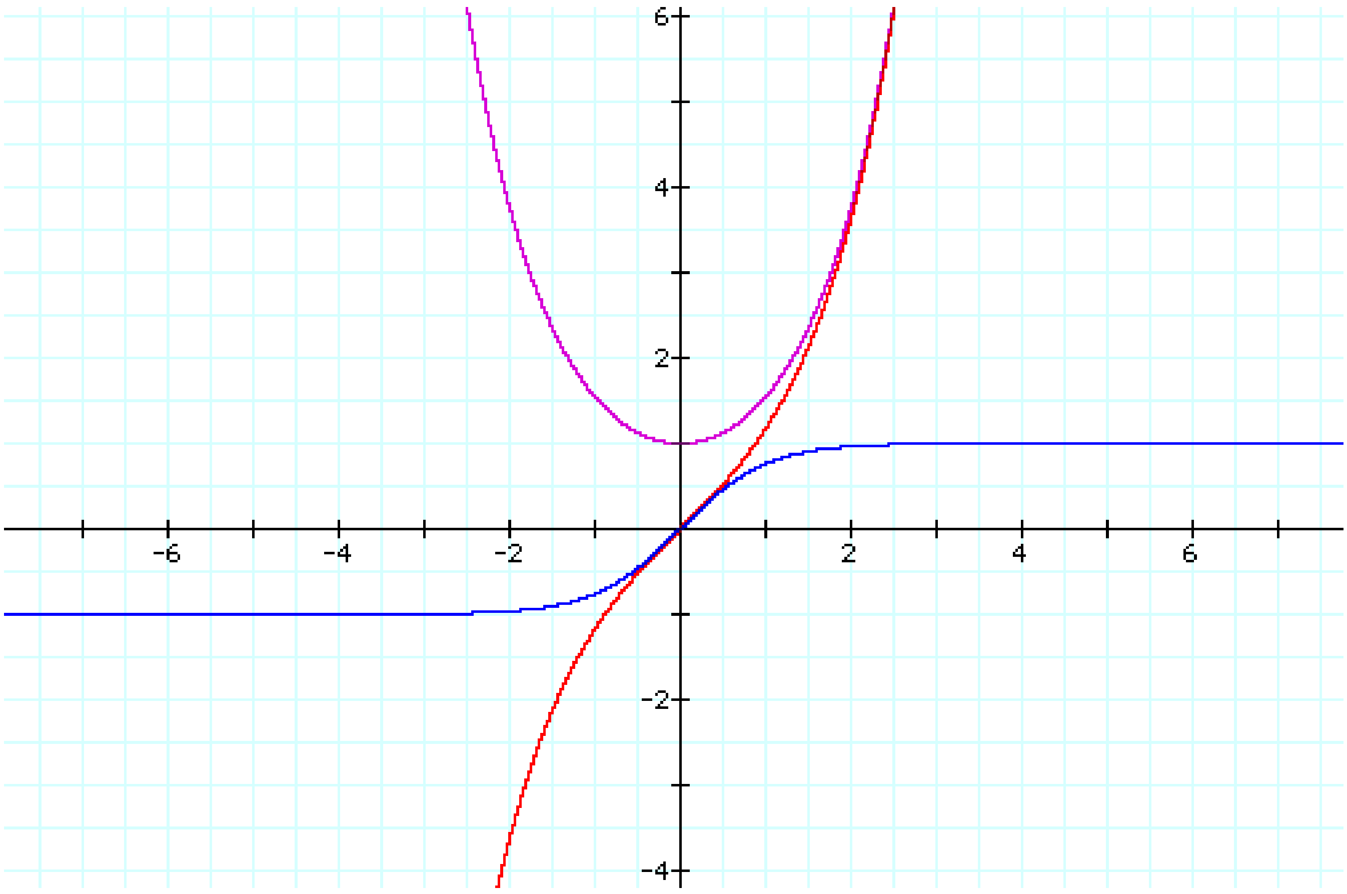
Hyperbolic tangent:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

Trigonometric functions on R (cos: purple; sin: red; tan: blue)



Hyperbolic functions on r (cosh: purple; sinh: red; tanh: blue)



The hyperbolic sine is one to one on the set of real numbers, with range the full set of reals, while the hyperbolic tangent is also one to one on the set of real numbers but with range `[0;+∞[`, and therefore both have inverses. The hyperbolic cosine is one to one from `]-∞;+1[` onto `]-∞;-1[` (and from `]+1;+∞[` onto `]-∞;-1[`); the inverse function we use here is defined on `]-∞;-1[` with range `]-∞;+1[`.

The inverse of the hyperbolic tangent is called the Argument hyperbolic tangent, and can be computed as

$$\operatorname{atanh}(x) \quad = \quad \frac{\log\left(\frac{1+x}{1-x}\right)}{2}$$

.

The inverse of the hyperbolic sine is called the Argument hyperbolic sine, and can be computed (for `[-1;-1+ε[`) as

$$\operatorname{asinh}(x) \quad = \quad \log\left(x+\sqrt{x^2+1}\right)$$

.

The inverse of the hyperbolic cosine is called the Argument hyperbolic cosine, and can be computed as

$$\operatorname{acosh}(x) \quad = \quad \log\left(x+\sqrt{x^2-1}\right)$$

.

acosh

```
#include <boost/math/special_functions/acosh.hpp>
```

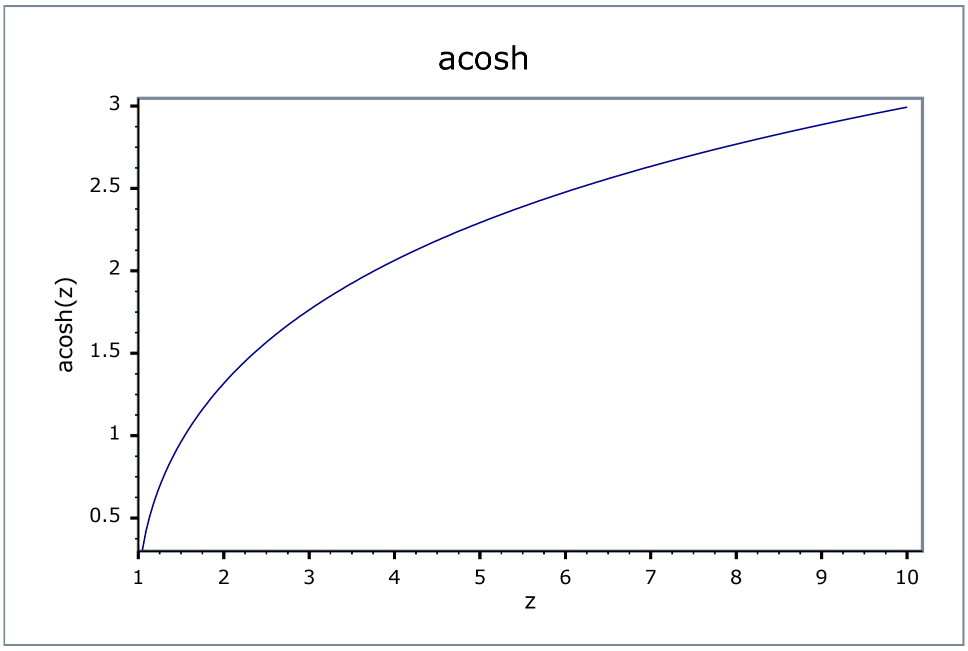
```
template<class T>
calculated-result-type acosh(const T x);

template<class T, class Policy>
calculated-result-type acosh(const T x, const Policy&);
```

Computes the reciprocal of (the restriction to the range of `[0;+∞[`) the hyperbolic cosine function, at x. Values returned are positive.

If x is in the range `]-∞;+1[` then returns the result of domain_error.

The return type of this function is computed using the *result type calculation rules*: the return type is `double` when T is an integer type, and T otherwise.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.



Accuracy

Generally accuracy is to within 1 or 2 epsilon across all supported platforms.

Testing

This function is tested using a combination of random test values designed to give full function coverage computed at high precision using the "naive" formula:

$$\operatorname{acosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

along with a selection of sanity check values computed using functions.wolfram.com to at least 50 decimal digits.

Implementation

For sufficiently large x, we can use the approximation:

$$\operatorname{acosh}(x) \approx \ln(2x) \quad ; \quad x > \frac{1}{\sqrt{\varepsilon}}$$

For x sufficiently close to 1 we can use the approximation:

$$\operatorname{acosh}(x) \approx \sqrt{2y}\left(1 - \frac{y}{12} + \frac{3y^2}{160}\right) \quad ; \quad y = \mathrm{x} - 1 \quad \wedge \quad y < \sqrt{\varepsilon}$$

Otherwise for x close to 1 we can use the following rearrangement of the primary definition to preserve accuracy:

$$\operatorname{acosh}(x) = \operatorname{log1p}\left(y + \sqrt{y^2 + 2y}\right) \quad ; \quad y = x - 1$$

Otherwise the primary definition is used:

$$\operatorname{acosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

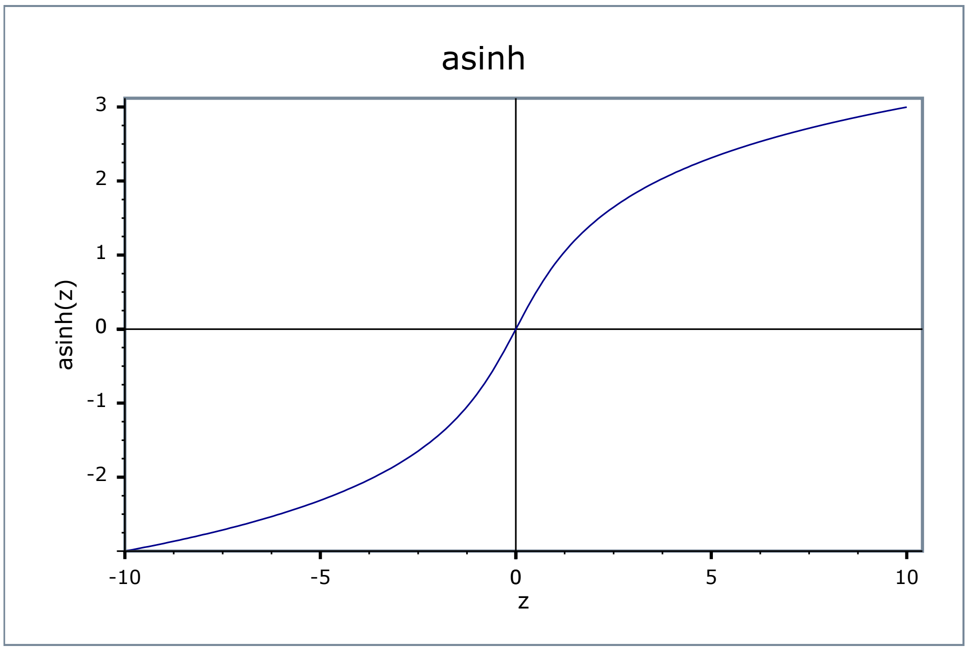
asinh

```
#include <boost/math/special_functions/asinh.hpp>
```

```
template<class T>
calculated-result-type asinh(const T x);

template<class T, class Policy>
calculated-result-type asinh(const T x, const Policy&);
```

Computes the reciprocal of the hyperbolic sine function.

The return type of this function is computed using the *result type calculation rules*: the return type is `double` when T is an integer type, and T otherwise.



The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Accuracy

Generally accuracy is to within 1 or 2 epsilon across all supported platforms.

Testing

This function is tested using a combination of random test values designed to give full function coverage computed at high precision using the "naive" formula:

$$\operatorname{asinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

along with a selection of sanity check values computed using functions.wolfram.com to at least 50 decimal digits.

Implementation

For sufficiently large x we can use the approximation:

$$\operatorname{asinh}(x) \approx \ln(2x) + \frac{1}{4x^2} \quad ; \quad x > \frac{1}{\sqrt{\varepsilon}}$$

While for very small x we can use the approximation:

$$\operatorname{asinh}(x) \approx x - \frac{x^3}{6} \quad ; \quad x < \sqrt{\varepsilon}$$

For 0.5 > x > ε the following rearrangement of the primary definition is used:

$$\operatorname{asinh}(x) = \operatorname{log1p}\left(x + \operatorname{sqrtp1m1}\left(x^2\right)\right)$$

Otherwise evalution is via the primary definition:

$$\operatorname{asinh}(x) = \operatorname{log1p}\left(x + \operatorname{sqrtp1m1}\left(x^2\right)\right)$$

atanh

```
#include <boost/math/special_functions/atanh.hpp>
```

```
template<class T>
calculated-result-type atanh(const T x);

template<class T, class Policy>
calculated-result-type atanh(const T x, const Policy&);
```

Computes the reciprocal of the hyperbolic tangent function, at x.

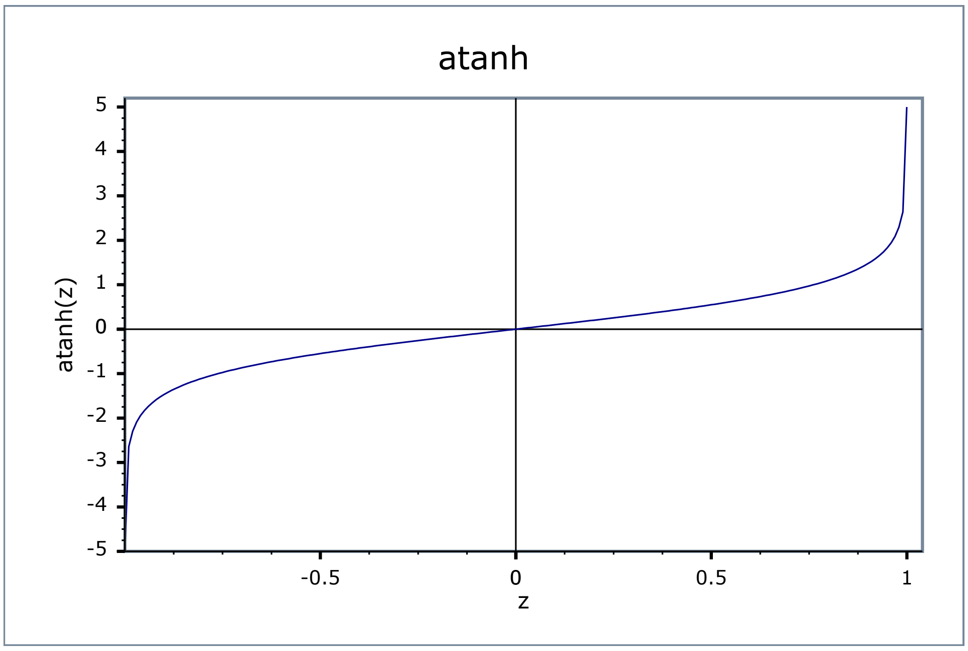
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

If x is in the range `]-∞;-1[` or in the range `]+1;+∞[` then returns the result of domain_error.

If x is in the range `[-1;-1+ε[`, then the result of -overflow_error is returned, with ε denoting numeric_limits<T>::epsilon().

If x is in the range `]+1-ε;+1]`, then the result of overflow_error is returned, with ε denoting numeric_limits<T>::epsilon().

The return type of this function is computed using the *result type calculation rules*: the return type is `double` when T is an integer type, and T otherwise.



Accuracy

Generally accuracy is to within 1 or 2 epsilon across all supported platforms.

Testing

This function is tested using a combination of random test values designed to give full function coverage computed at high precision using the "naive" formula:

$$\operatorname{atanh}(x) = \frac{\ln\left(\frac{1+x}{1-x}\right)}{2}$$

along with a selection of sanity check values computed using functions.wolfram.com to at least 50 decimal digits.

Implementation

For sufficiently small x we can use the approximation:

$$\operatorname{atanh}(x) \approx x + \frac{x^3}{3} \quad ; \quad x < \sqrt{\varepsilon}$$

Otherwise the primary definition:

$$\operatorname{atanh}(x) = \frac{\ln\left(\frac{1+x}{1-x}\right)}{2}$$

or its equivalent form:

$$\operatorname{atanh}(x) = \frac{\operatorname{log1p}(x) - \operatorname{log1p}(-x)}{2}$$

is used.

Owen's T function

Synopsis

```
#include <boost/math/special_functions/owens_t.hpp>
```

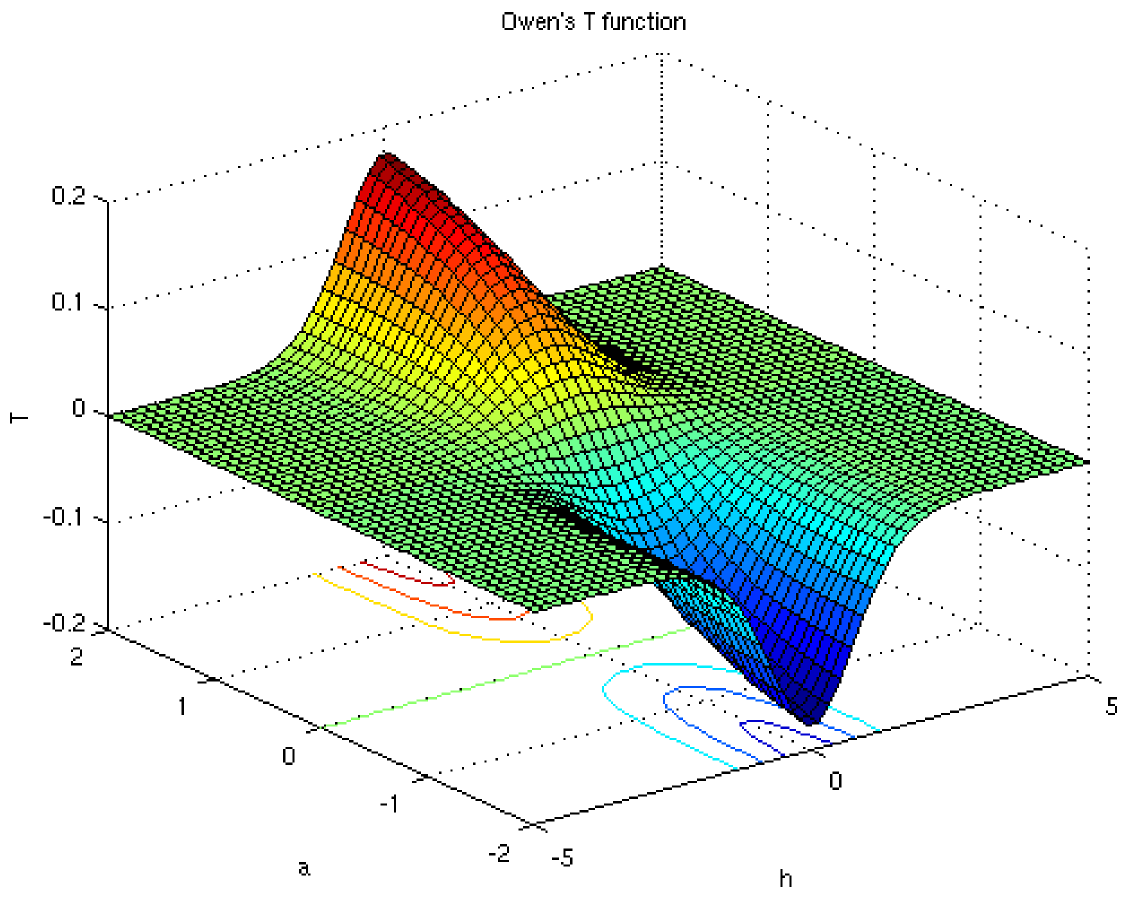
```
namespace boost{ namespace math{

template <class T>
calculated-result-type owens_t(T h, T a);

template <class T, class Policy>
calculated-result-type owens_t(T h, T a, const Policy&);

}} // namespaces
```

Description

Returns the Owens_t function of *h* and *a*.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

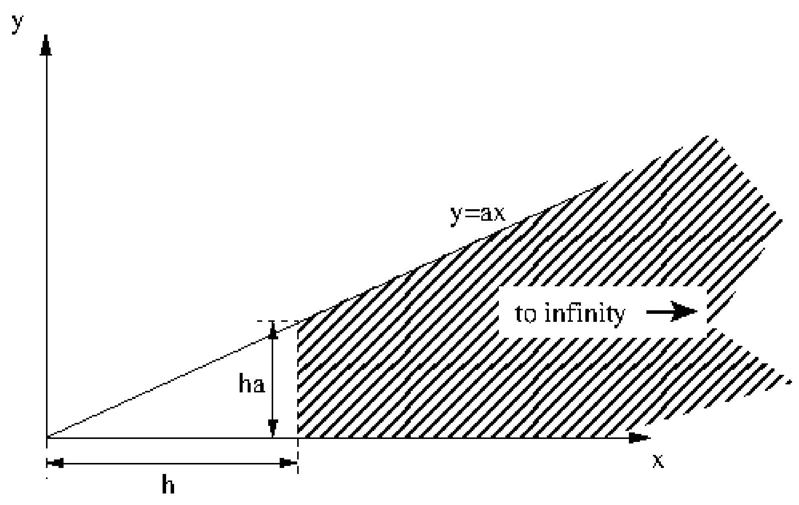
$$T(h, a) = \frac{1}{(2\pi)}\int_0^a \frac{\exp\left\{-\frac{1}{2}h^2\left(1+x^2\right)\right\}}{\left(1+x^2\right)}dx \quad ; \quad (-\infty < h, a < +\infty)$$



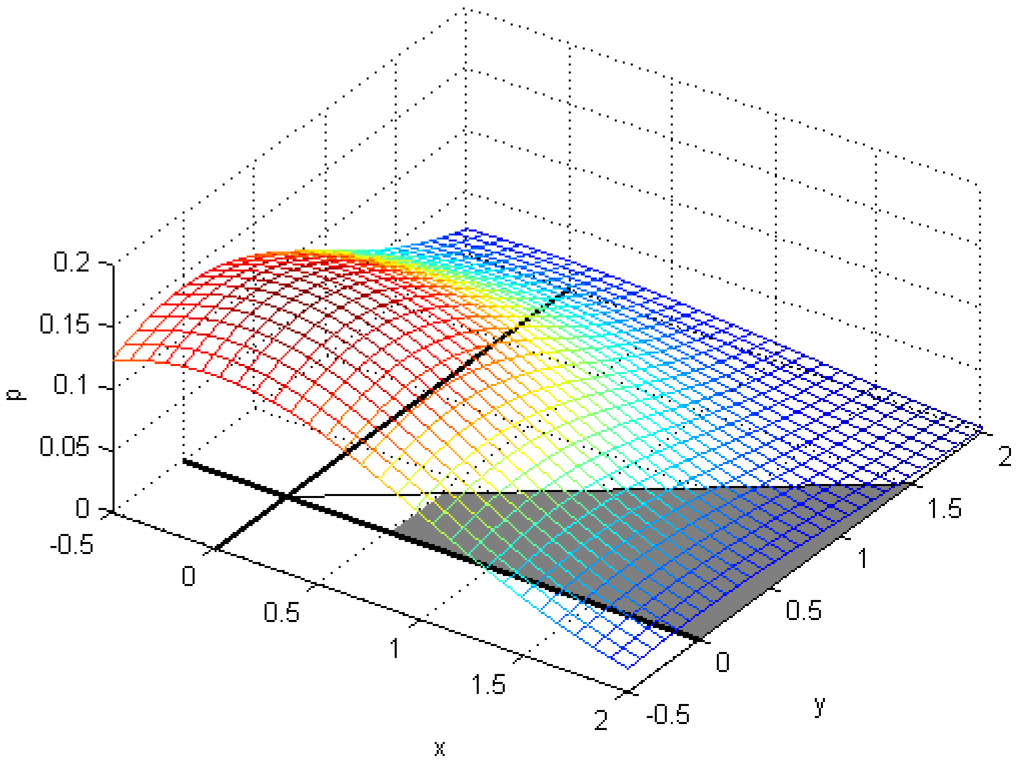
The function `owens_t(h, a)` gives the probability of the event *(X > h and 0 < Y < a * X)*, where *X* and *Y* are independent standard normal random variables.

For h and a > 0, T(h,a), gives the volume of an uncorrelated bivariate normal distribution with zero means and unit variances over the area between *y = ax* and *y = 0* and to the right of *x = h*.

That is the area shaded in the figure below (Owens 1956).



and is also illustrated by a 3D plot.



This function is used in the computation of the Skew Normal Distribution. It is also used in the computation of bivariate and multivariate normal distribution probabilities. The return type of this function is computed using the *result type calculation rules*: the result is of type `double` when T is an integer type, and type T otherwise.

Owen's original paper (page 1077) provides some additional corner cases.

> *T(h, 0) = 0*
>
> *T(0, a) = ½π arctan(a)*
>
> *T(h, 1) = ½ G(h) [1 - G(h)]*
>
> *T(h, ∞) = G(|h|)*

where G(h) is the univariate normal with zero mean and unit variance integral from -∞ to h.

Accuracy

Over the built-in types and range tested, errors are less than 10 * std::numeric_limits<RealType>::epsilon().

Table 109. Error rates for owens_t

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Owens T (medium small values)	Max = 0ε (Mean = 0ε)	Max = 3.34ε (Mean = 0.944ε)	Max = 3.34ε (Mean = 0.911ε)	Max = 4.37ε (Mean = 0.98ε)
Owens T (large and diverse values)	Max = 0ε (Mean = 0ε)	Max = 49ε (Mean = 2.16ε)	Max = 24.5ε (Mean = 1.39ε)	Max = 3.78ε (Mean = 0.621ε)

Testing

Test data was generated by Patefield and Tandy algorithms T1 and T4, and also the suggested reference routine T7.

- T1 was rejected if the result was too small compared to `atan(a)` (ie cancellation),
- T4 was rejected if there was no convergence,
- Both were rejected if they didn't agree.

Over the built-in types and range tested, errors are less than 10 std::numeric_limits<RealType>::epsilon().

However, that there was a whole domain (large *h*, small *a*) where it was not possible to generate any reliable test values (all the methods got rejected for one reason or another).

There are also two sets of sanity tests: spot values are computed using Wolfram Mathematica and The R Project for Statistical Computing.

Implementation

The function was proposed and evaluated by Donald. B. Owen, Tables for computing bivariate normal probabilities, Ann. Math. Statist., 27, 1075-1090 (1956).

The algorithms of Patefield, M. and Tandy, D. "Fast and accurate Calculation of Owen's T-Function", Journal of Statistical Software, 5 (5), 1 - 25 (2000) are adapted for C++ with arbitrary RealType.

The Patefield-Tandy algorithm provides six methods of evalualution (T1 to T6); the best method is selected according to the values of *a* and *h*. See the original paper and the source in owens_t.hpp for details.

The Patefield-Tandy algorithm is accurate to approximately 20 decimal places, so for types with greater precision we use:

- A modified version of T1 which folds the calculation of *atan(h)* into the T1 series (to avoid subtracting two values similar in magnitude), and then accelerates the resulting alternating series using method 1 from H. Cohen, F. Rodriguez Villegas, D. Zagier, "Convergence acceleration of alternating series", Bonn, (1991). The result is valid everywhere, but doesn't always converge, or may become too divergent in the first few terms to sum accurately. This is used for $ah < 1$.
- A modified version of T2 which is accelerated in the same manner as T1. This is used for $h > 1$.
- A version of T4 only when both T1 and T2 have failed to produce an accurate answer.
- Fallback to the Patefiled Tandy algorithm when all the above methods fail: this happens not at all for our test data at 100 decimal digits precision. However, there is a difficult area when *a* is very close to 1 and the precision increases which may cause this to happen in very exceptional circumstances.

Using the above algorithm and a 100-decimal digit type, results accurate to 80 decimal places were obtained in the difficult area where *a* is close to 1, and greater than 95 decimal places elsewhere.

TR1 and C99 external "C" Functions

C99 and TR1 C Functions Overview

Many of the special functions included in this library are also a part of the either the C99 Standard ISO/IEC 9899:1999 or the Technical Report on C++ Library Extensions. Therefore this library includes a thin wrapper header `boost/math/tr1.hpp` that provides compatibility with these two standards.

There are various pros and cons to using the library in this way:

Pros:

- The header to include is lightweight (i.e. fast to compile).
- The functions have extern "C" linkage, and so are usable from other languages (not just C and C++).
- C99 and C++ TR1 Standard compatibility.

Cons:

- You will need to compile and link to the external Boost.Math libraries.
- Limited to support for the types, `float`, `double` and `long double`.
- Error handling is handled via setting ::errno and returning NaN's and infinities: this may be less flexible than an C++ exception based approach.

Note

The separate libraries are required **only** if you choose to use boost/math/tr1.hpp rather than some other Boost.Math header, the rest of Boost.Math remains header-only.

The separate libraries required in order to use tr1.hpp can be compiled using bjam from within the libs/math/build directory, or from the Boost root directory using the usual Boost-wide install procedure. Alternatively the source files are located in libs/math/src and each have the same name as the function they implement. The various libraries are named as follows:

Name	Type	Functions
boost_math_c99f-<suffix>	float	C99 Functions
boost_math_c99-<suffix>	double	C99 Functions
boost_math_c99l-<suffix>	long double	C99 Functions
boost_math_tr1f-<suffix>	float	TR1 Functions
boost_math_tr1-<suffix>	double	TR1 Functions
boost_math_tr1l-<suffix>	long double	TR1 Functions

Where `<suffix>` encodes the compiler and build options used to build the libraries: for example "libboost_math_tr1-vc80-mt-gd.lib" would be the statically linked TR1 library to use with Visual C++ 8.0, in multithreading debug mode, with the DLL VC++ runtime, where as "boost_math_tr1-vc80-mt.lib" would be import library for the TR1 DLL to be used with Visual C++ 8.0 with the release multithreaded DLL VC++ runtime. Refer to the getting started guide for a full explanation of the `<suffix>` meanings.

Note

Visual C++ users will typically have the correct library variant to link against selected for them by boost/math/tr1.hpp based on your compiler settings.

Users will need to define BOOST_MATH_TR1_DYN_LINK when building their code if they want to link against the DLL versions of these libraries rather than the static versions.

Users can disable auto-linking by defining BOOST_MATH_TR1_NO_LIB when building: this is typically only used when linking against a customised build of the libraries.

Note

Linux and Unix users will generally only have one variant of these libraries installed, and can generally just link against -lboost_math_tr1 etc.

Usage Recomendations

This library now presents the user with a choice:

- To include the header only versions of the functions and have an easier time linking, but a longer compile time.
- To include the TR1 headers and link against an external library.

Which option you choose depends largely on how you prefer to work and how your system is set up.

For example a casual user who just needs the acosh function, would probably be better off including `<boost/math/special_functions/acosh.hpp>` and using `boost::math::acosh(x)` in their code.

However, for large scale software development where compile times are significant, and where the Boost libraries are already built and installed on the system, then including `<boost/math/tr1.hpp>` and using `boost::math::tr1::acosh(x)` will speed up compile times, reduce object files sizes (since there are no templates being instantiated any more), and also speed up debugging runtimes - since the externally compiled libraries can be compiler optimised, rather than built using full settings - the difference in performance between release and debug builds can be as much as 20 times, so for complex applications this can be a big win.

Supported C99 Functions

See also the quick reference guide for these functions.

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

typedef unspecified float_t;
typedef unspecified double_t;

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

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

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

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

double round(double x);
float roundf(float x);
long double roundl(long double x);

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

double trunc(double x);
float truncf(float x);
long double truncl(long double x);

}}}} // namespaces
```

Supported TR1 Functions

See also the quick reference guide for these functions.

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

// [5.2.1.1] 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);

// [5.2.1.2] 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);

// [5.2.1.3] beta function:
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// [5.2.1.4] (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);

// [5.2.1.5] (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);

// [5.2.1.6] (complete) elliptic integral of the third kind:
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);

// [5.2.1.8] 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);

// [5.2.1.9] 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);

// [5.2.1.10] 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);

// [5.2.1.11] 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);

// [5.2.1.12] (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);

// [5.2.1.13] (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);
// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// [5.2.1.15] exponential integral:
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// [5.2.1.16] Hermite polynomials:
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// [5.2.1.18] Laguerre polynomials:
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// [5.2.1.19] Legendre polynomials:
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// [5.2.1.20] Riemann zeta function:
double riemann_zeta(double);
float riemann_zetaf(float);
long double riemann_zetal(long double);

// [5.2.1.21] 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);

// [5.2.1.22] 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);

// [5.2.1.23] 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);

}}}} // namespaces
```

In addition sufficient additional overloads of the `double` versions of the above functions are provided, so that calling the function with any mixture of `float`, `double`, `long double`, or *integer* arguments is supported, with the return type determined by the *result type calculation rules*.

Currently Unsupported C99 Functions

```
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);

double log2(double x);
float log2f(float x);
long double log2l(long double x);

double logb(double x);
float logbf(float x);
long double logbl(long double x);

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);

double nan(const char *str);
float nanf(const char *str);
long double nanl(const char *str);

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

double remquo(double x, double y, int *pquo);
float remquof(float x, float y, int *pquo);
long double remquol(long double x, long double y, int *pquo);

double rint(double x);
float rintf(float x);
long double rintl(long double x);

double scalbln(double x, long ex);
```

```
float scalblnf(float x, long ex);
long double scalblnl(long double x, long ex);

double scalbn(double x, int ex);
float scalbnf(float x, int ex);
long double scalbnl(long double x, int ex);
```

Currently Unsupported TR1 Functions

```
// [5.2.1.7] 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);

// [5.2.1.17] 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);
```

C99 C Functions

Supported C99 Functions

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

typedef unspecified float_t;
typedef unspecified double_t;

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

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

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

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

double round(double x);
float roundf(float x);
long double roundl(long double x);

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

double trunc(double x);
float truncf(float x);
long double truncl(long double x);

}}}} // namespaces
```

In addition sufficient additional overloads of the `double` versions of the above functions are provided, so that calling the function with any mixture of `float`, `double`, `long double`, or *integer* arguments is supported, with the return type determined by the *result type calculation rules*.

For example:

```
acoshf(2.0f);  // float version, returns float.
acosh(2.0f);   // also calls the float version and returns float.
acosh(2.0);    // double version, returns double.
acoshl(2.0L);  // long double version, returns a long double.
acosh(2.0L);   // also calls the long double version.
acosh(2);      // integer argument is treated as a double, returns double.
```

Quick Reference

More detailed descriptions of these functions are available in the C99 standard.

```
typedef unspecified float_t;
typedef unspecified double_t;
```

In this implementation `float_t` is the same as type `float`, and `double_t` the same as type `double` unless the preprocessor symbol FLT_EVAL_METHOD is defined, in which case these are set as follows:

FLT_EVAL_METHOD	float_t	double_t
0	float	double
1	double	double
2	long double	long double

```
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Returns the inverse hyperbolic cosine of *x*.

See also acosh for the full template (header only) version of this function.

```
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Returns the inverse hyperbolic sine of *x*.

See also asinh for the full template (header only) version of this function.

```
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Returns the inverse hyperbolic tangent of *x*.

See also atanh for the full template (header only) version of this function.

```
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Returns the cubed root of *x*.

See also cbrt for the full template (header only) version of this function.

```
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Returns a value with the magnitude of *x* and the sign of *y*.

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

Returns the error function of *x*:

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

See also erf for the full template (header only) version of this function.

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

Returns the complementary error function of *x* `1-erf(x)` without the loss of precision implied by the subtraction.

See also erfc for the full template (header only) version of this function.

```
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Returns `exp(x)-1` without the loss of precision implied by the subtraction.

See also expm1 for the full template (header only) version of this function.

```
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Returns the larger (most positive) of *x* and *y*.

```
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Returns the smaller (most negative) of *x* and *y*.

```
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Returns `sqrt(x*x + y*y)` without the danger of numeric overflow implied by that formulation.

See also hypot for the full template (header only) version of this function.

```
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Returns the log of the gamma function of *x*.

$$\text{lgamma}(z) = \ln|\Gamma(z)|$$

See also lgamma for the full template (header only) version of this function.

```
long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);
```

Returns the value *x* rounded to the nearest integer as a `long long`: equivalent to `floor(x + 0.5)`

See also round for the full template (header only) version of this function.

```
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Returns the `log(x+1)` without the loss of precision implied by that formulation.

See also log1p for the full template (header only) version of this function.

```
long lround(double x);
long lroundf(float x);
long lroundl(long double x);
```

Returns the value *x* rounded to the nearest integer as a `long`: equivalent to `floor(x + 0.5)`

See also round for the full template (header only) version of this function.

```
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Returns the next representable floating point number after *x* in the direction of *y*, or *x* if `x == y`.

```
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

As `nextafter`, but with *y* always expressed as a `long double`.

```
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Returns the value *x* rounded to the nearest integer: equivalent to `floor(x + 0.5)`

See also round for the full template (header only) version of this function.

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

Returns the gamma function of *x*:

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

See also tgamma for the full template (header only) version of this function.

```
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Returns *x* truncated to the nearest integer.

See also trunc for the full template (header only) version of this function.

See also C99 ISO Standard

TR1 C Functions Quick Reference

Supported TR1 Functions

```
namespace boost{ namespace math{ namespace tr1{ extern "C"{

// [5.2.1.1] 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);

// [5.2.1.2] 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);

// [5.2.1.3] beta function:
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// [5.2.1.4] (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);

// [5.2.1.5] (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);

// [5.2.1.6] (complete) elliptic integral of the third kind:
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);

// [5.2.1.8] 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);

// [5.2.1.9] 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);

// [5.2.1.10] 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);

// [5.2.1.11] 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);

// [5.2.1.12] (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);
// [5.2.1.13] (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);

// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// [5.2.1.15] exponential integral:
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// [5.2.1.16] Hermite polynomials:
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// [5.2.1.18] Laguerre polynomials:
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// [5.2.1.19] Legendre polynomials:
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// [5.2.1.20] Riemann zeta function:
double riemann_zeta(double);
float riemann_zetaf(float);
long double riemann_zetal(long double);

// [5.2.1.21] 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);

// [5.2.1.22] 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);

// [5.2.1.23] 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);

}}}} // namespaces
```

In addition sufficient additional overloads of the `double` versions of the above functions are provided, so that calling the function with any mixture of `float`, `double`, `long double`, or *integer* arguments is supported, with the return type determined by the *result type calculation rules*.

For example:

```
expintf(2.0f);  // float version, returns float.
expint(2.0f);   // also calls the float version and returns float.
expint(2.0);    // double version, returns double.
expintl(2.0L);  // long double version, returns a long double.
expint(2.0L);   // also calls the long double version.
expint(2);      // integer argument is treated as a double, returns double.
```

Quick Reference

```
// [5.2.1.1] 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);
```

The assoc_laguerre functions return:

$$\text{laguerre}\,(n,\, m,\, x) \quad = \quad L_n^m(x) \quad = \quad (-1)^m \frac{d^m}{dx^m} L_{n+m}(x)$$

See also laguerre for the full template (header only) version of this function.

```
// [5.2.1.2] 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);
```

The assoc_legendre functions return:

$$\text{legendre\_p}\,(l,\, m,\, x) \quad = \quad P_l^m(x) \quad = \quad (1 - x^2)^{\frac{m}{2}} \frac{d^m P_l(x)}{dx^m}$$

See also legendre_p for the full template (header only) version of this function.

```
// [5.2.1.3] beta function:
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);
```

Returns the beta function of *x* and *y*:

$$\text{beta}\,(a,\, b) = B(a,\, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

See also beta for the full template (header only) version of this function.

```
// [5.2.1.4] (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);
```

Returns the complete elliptic integral of the first kind of *k*:

$$K(k) = F\left(\frac{\pi}{2},\, k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}$$

See also ellint_1 for the full template (header only) version of this function.

```
// [5.2.1.5] (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);
```

Returns the complete elliptic integral of the second kind of *k*:

$$E(k) = E\left(\frac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \sqrt{1 - k^2 \sin^2\theta}\, d\theta$$

See also ellint_2 for the full template (header only) version of this function.

```
// [5.2.1.6] (complete) elliptic integral of the third kind:
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
```

Returns the complete elliptic integral of the third kind of *k* and *nu*:

$$\prod(n, k) = \prod\left(n, \frac{\pi}{2}, k\right) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{(1 - n\sin^2\theta)\sqrt{1 - k^2\sin^2\theta}}$$

See also ellint_3 for the full template (header only) version of this function.

```
// [5.2.1.8] 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);
```

Returns the modified bessel function of the first kind of *nu* and *x*:

$$I_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\,\Gamma(v + k + 1)}$$

See also cyl_bessel_i for the full template (header only) version of this function.

```
// [5.2.1.9] 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);
```

Returns the bessel function of the first kind of *nu* and *x*:

$$J_v(z) = \left(\frac{1}{2}z\right)^v \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\,\Gamma(v + k + 1)}$$

See also cyl_bessel_j for the full template (header only) version of this function.

```
// [5.2.1.10] 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);
```

Returns the modified bessel function of the second kind of *nu* and *x*:

$$K_\nu(z) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

See also cyl_bessel_k for the full template (header only) version of this function.

```
// [5.2.1.11] 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);
```

Returns the bessel function of the second kind (Neumann function) of *nu* and *x*:

$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

See also cyl_neumann for the full template (header only) version of this function.

```
// [5.2.1.12] (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);
```

Returns the incomplete elliptic integral of the first kind of *k* and *phi*:

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

See also ellint_1 for the full template (header only) version of this function.

```
// [5.2.1.13] (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);
```

Returns the incomplete elliptic integral of the second kind of *k* and *phi*:

$$E(\varphi, k) = \int_0^\varphi \sqrt{1 - k^2\sin^2\theta}\, d\theta$$

See also ellint_2 for the full template (header only) version of this function.

```
// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);
```

Returns the incomplete elliptic integral of the third kind of *k*, *nu* and *phi*:

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

See also ellint_3 for the full template (header only) version of this function.

```
// [5.2.1.15] exponential integral:
double expint(double x);
float expintf(float x);
long double expintl(long double x);
```

Returns the exponential integral Ei of *x*:

$$Ei(x) = -E_1(-x) = \int_{-x}^{\infty} \frac{e^{-t}dt}{t}$$

See also expint for the full template (header only) version of this function.

```
// [5.2.1.16] Hermite polynomials:
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
```

Returns the n'th Hermite polynomial of *x*:

$$\text{hermite}\,(x) \quad = \quad H_n(x) \quad = \quad (-1)^n e^{x^2} \frac{d^2}{dx^2} e^{-x^2}$$

See also hermite for the full template (header only) version of this function.

```
// [5.2.1.18] Laguerre polynomials:
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);
```

Returns the n'th Laguerre polynomial of *x*:

$$\text{laguerre}\,(n, x) \quad = \quad L_n(x) \quad = \quad \frac{e^x}{n!} \frac{d^n}{dx^n}(x^n e^{-x})$$

See also laguerre for the full template (header only) version of this function.

```
// [5.2.1.19] Legendre polynomials:
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);
```

Returns the l'th Legendre polynomial of *x*:

$$\text{legendre\_p}\,(l, x) \quad = \quad P_l(x) \quad = \quad \frac{1}{2^l l!} \frac{d^l}{dx^l}(x^2 - 1)^l \quad ; \quad \left| x \right| \le 1$$

See also legendre_p for the full template (header only) version of this function.

```
// [5.2.1.20] Riemann zeta function:
double riemann_zeta(double);
float riemann_zetaf(float);
long double riemann_zetal(long double);
```

Returns the Riemann Zeta function of *x*:

$$\zeta(s) = \sum_{k=1}^{\infty} \frac{1}{k^s}$$

See also zeta for the full template (header only) version of this function.

```
// [5.2.1.21] 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);
```

Returns the spherical Bessel function of the first kind of *x* $j_n(x)$:

$$j_n(z) = \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z)$$

$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z)$$

See also sph_bessel for the full template (header only) version of this function.

```
// [5.2.1.22] 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);
```

Returns the spherical associated Legendre function of *l*, *m* and *theta*:

$$\text{sph\_legendre}(l, m, \theta) = Y_l^m(\theta, 0) = (-1)^m \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} P_l^m(\cos\theta)$$

See also spherical_harmonic for the full template (header only) version of this function.

```
// [5.2.1.23] 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);
```

Returns the spherical Neumann function of *x* $y_n(x)$:

$$j_n(z) = \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z)$$

$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z)$$

See also sph_bessel for the full template (header only) version of this function.

Currently Unsupported TR1 Functions

```
// [5.2.1.7] 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);

// [5.2.1.17] 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);
```

Note

These two functions are not implemented as they are not believed to be numerically stable.

Root Finding & Minimization Algorithms

Several tools are provided to aid finding minima and roots of functions.

Some root-finding without derivatives methods are bisection, bracket and solve, including use of TOMS 748 algorithm.

For root-finding with derivatives the methods of Newton-Raphson iteration, Halley, and Schröder are implemented.

For locating minima of a function, a Brent minima finding example is provided.

There are several fully-worked root-finding examples, including:

- root-finding without derivatives
- root-finding with 1st derivatives
- root-finding with 1st and 2nd derivatives

Root Finding Without Derivatives

Synopsis

```
#include <boost/math/tools/roots.hpp>
```

```
namespace boost { namespace math {
namespace tools { // Note namespace boost::math::tools.
// Bisection
template <class F, class T, class Tol>
std::pair<T, T>
   bisect(
      F f,
      T min,
      T max,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol>
std::pair<T, T>
   bisect(
      F f,
      T min,
      T max,
      Tol tol);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   bisect(
      F f,
      T min,
      T max,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);

// Bracket and Solve Root
template <class F, class T, class Tol>
std::pair<T, T>
   bracket_and_solve_root(
      F f,
      const T& guess,
      const T& factor,
      bool rising,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   bracket_and_solve_root(
      F f,
      const T& guess,
      const T& factor,
      bool rising,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);

 // TOMS 748 algorithm
template <class F, class T, class Tol>
std::pair<T, T>
```

```
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);

template <class F, class T, class Tol>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      const T& fa,
      const T& fb,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      const T& fa,
      const T& fb,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);

// Termination conditions:
template <class T>
struct eps_tolerance;

struct equal_floor;
struct equal_ceil;
struct equal_nearest_integer;

}}} // boost::math::tools namespaces
```

Description

These functions solve the root of some function *f(x) - without the need for any derivatives of* f(x).

The `bracket_and_solve_root` functions use TOMS 748 algorithm by Alefeld, Potra and Shi that is asymptotically the most efficient known, and has been shown to be optimal for a certain classes of smooth functions. Variants with and without Policies are provided.

Alternatively, bisect is a simple bisection routine which can be useful in its own right in some situations, or alternatively for narrowing down the range containing the root, prior to calling a more advanced algorithm.

All the algorithms in this section reduce the diameter of the enclosing interval with the same asymptotic efficiency with which they locate the root. This is in contrast to the derivative based methods which may *never* significantly reduce the enclosing interval, even though they rapidly approach the root. This is also in contrast to some other derivative-free methods (for example, Brent's method described at Brent-Dekker) which only reduces the enclosing interval on the final step. Therefore these methods return a `std::pair` containing the enclosing interval found, and accept a function object specifying the termination condition.

Three function objects are provided for ready-made termination conditions:

- *eps_tolerance* causes termination when the relative error in the enclosing interval is below a certain threshold.
- *equal_floor* and *equal_ceil* are useful for certain statistical applications where the result is known to be an integer.
- Other user-defined termination conditions are likely to be used only rarely, but may be useful in some specific circumstances.

Bisection

```
template <class F, class T, class Tol>
std::pair<T, T>
   bisect(  // Unlimited iterations.
      F f,
      T min,
      T max,
      Tol tol);

template <class F, class T, class Tol>
std::pair<T, T>
   bisect(  // Limited iterations.
      F f,
      T min,
      T max,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   bisect( // Specified policy.
      F f,
      T min,
      T max,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);
```

These functions locate the root using bisection.

`bisect` function arguments are:

f	A unary functor (or C++ lambda) which is the function *f(x)* whose root is to be found.
min	The left bracket of the interval known to contain the root.
max	The right bracket of the interval known to contain the root. It is a precondition that *min < max* and *f(min)*f(max) <= 0*, the function raises an evaluation_error if these preconditions are violated. The action taken on error is controlled by the Policy template argument: the default behavior is to throw a *boost::math::evaluation_error*. If the Policy is changed to not throw then it returns *std::pair<T>(min, min)*.
tol	A binary functor (or C++ lambda) that specifies the termination condition: the function will return the current brackets enclosing the root when *tol(min, max)* becomes true. See also predefined termination functors.

max_iter	The maximum number of invocations of *f(x)* to make while searching for the root. On exit, this is updated to the actual number of invocations performed.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Returns: a pair of values *r* that bracket the root so that:

```
f(r.first) * f(r.second) <= 0
```

and either

```
tol(r.first, r.second) == true
```

or

```
max_iter >= m
```

where *m* is the initial value of *max_iter* passed to the function.

In other words, it's up to the caller to verify whether termination occurred as a result of exceeding *max_iter* function invocations (easily done by checking the updated value of *max_iter* when the function returns), rather than because the termination condition *tol* was satisfied.

Bracket and Solve Root

```
template <class F, class T, class Tol>
std::pair<T, T>
   bracket_and_solve_root(
      F f,
      const T& guess,
      const T& factor,
      bool rising,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   bracket_and_solve_root(
      F f,
      const T& guess,
      const T& factor,
      bool rising,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);
```

`bracket_and_solve_root` is a convenience function that calls TOMS 748 algorithm internally to find the root of *f(x)*. It is generally much easier to use this function rather than TOMS 748 algorithm, since it does the hard work of bracketing the root for you. It's bracketing routines are quite robust and will usually be more foolproof than home-grown routines, unless the function can be analysed to yield tight brackets.

Note that this routine can only be used when:

- *f(x)* is monotonic in the half of the real axis containing *guess*.
- The value of the inital guess must have the same sign as the root: the function will *never cross the origin* when searching for the root.

- The location of the root should be known at least approximately, if the location of the root differs by many orders of magnitude from *guess* then many iterations will be needed to bracket the root in spite of the special heuristics used to guard against this very situation. A typical example would be setting the initial guess to 0.1, when the root is at 1e-300.

The `bracket_and_solve_root` parameters are:

f	A unary functor (or C++ lambda) that is the function whose root is to be solved. *f(x)* must be uniformly increasing or decreasing on *x*.
guess	An initial approximation to the root.
factor	A scaling factor that is used to bracket the root: the value *guess* is multiplied (or divided as appropriate) by *factor* until two values are found that bracket the root. A value such as 2 is a typical choice for *factor*. In addition *factor* will be multiplied by 2 every 32 iterations: this is to guard against a really very bad initial guess, typically these occur when it's known the result is very large or small, but not the exact order of magnitude.
rising	Set to *true* if *f(x)* is rising on *x* and *false* if *f(x)* is falling on *x*. This value is used along with the result of *f(guess)* to determine if *guess* is above or below the root.
tol	A binary functor (or C++ lambda) that determines the termination condition for the search for the root. *tol* is passed the current brackets at each step, when it returns true then the current brackets are returned as the pair result. See also predefined termination functors.
max_iter	The maximum number of function invocations to perform in the search for the root. On exit is set to the actual number of invocations performed.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Returns: a pair of values *r* that bracket the root so that:

```
f(r.first) * f(r.second) <= 0
```

and either

```
tol(r.first, r.second) == true
```

or

```
max_iter >= m
```

where *m* is the initial value of *max_iter* passed to the function.

In other words, it's up to the caller to verify whether termination occurred as a result of exceeding *max_iter* function invocations (easily done by checking the value of *max_iter* when the function returns), rather than because the termination condition *tol* was satisfied.

Algorithm TOMS 748: Alefeld, Potra and Shi: Enclosing zeros of continuous functions

```
template <class F, class T, class Tol>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);

template <class F, class T, class Tol>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      const T& fa,
      const T& fb,
      Tol tol,
      boost::uintmax_t& max_iter);

template <class F, class T, class Tol, class Policy>
std::pair<T, T>
   toms748_solve(
      F f,
      const T& a,
      const T& b,
      const T& fa,
      const T& fb,
      Tol tol,
      boost::uintmax_t& max_iter,
      const Policy&);
```

These functions implement TOMS Algorithm 748: it uses a mixture of cubic, quadratic and linear (secant) interpolation to locate the root of *f(x)*. The two pairs of functions differ only by whether values for *f(a)* and *f(b)* are already available.

Generally speaking it is easier (and often more efficient) to use bracket and solve rather than trying to bracket the root yourself as this function requires.

This function is provided rather than Brent's method as it is known to be more effient in many cases (it is asymptotically the most efficient known, and has been shown to be optimal for a certain classes of smooth functions). It also has the useful property of decreasing the bracket size with each step, unlike Brent's method which only shrinks the enclosing interval in the final step. This makes it particularly useful when you need a result where the ends of the interval round to the same integer: as often happens in statistical applications for example. In this situation the function is able to exit after a much smaller number of iterations than would otherwise be possible.

The TOMS 748 algorithm parameters are:

f	A unary functor (or C++ lambda) that is the function whose root is to be solved. f(x) need not be uniformly increasing or decreasing on *x* and may have multiple roots. However, the bounds given must bracket a single root.
a	The lower bound for the initial bracket of the root.
b	The upper bound for the initial bracket of the root. It is a precondition that *a* < *b* and that *a* and *b* bracket the root to find so that *f(a) * f(b) < 0*.
fa	Optional: the value of *f(a)*.
fb	Optional: the value of *f(b)*.
tol	A binary functor (or C++ lambda) that determines the termination condition for the search for the root. *tol* is passed the current brackets at each step, when it returns true, then the current brackets are returned as the result. See also predefined termination functors.
max_iter	The maximum number of function invocations to perform in the search for the root. On exit, *max_iter* is set to actual number of function invocations used.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

`toms748_solve` returns: a pair of values *r* that bracket the root so that:

```
f(r.first) * f(r.second) <= 0
```

and either

```
tol(r.first, r.second) == true
```

or

```
max_iter >= m
```

where *m* is the initial value of *max_iter* passed to the function.

In other words, it's up to the caller to verify whether termination occurred as a result of exceeding *max_iter* function invocations (easily done by checking the updated value of *max_iter* against its previous value passed as parameter), rather than because the termination condition *tol* was satisfied.

Brent-Decker Algorithm

The Brent-Dekker algorithm, although very well know, is not provided by this library as TOMS 748 algorithm or its slightly easier to use variant bracket and solve are superior and provide equivalent functionality.

Termination Condition Functors

```
template <class T>
struct eps_tolerance
{
   eps_tolerance();
   eps_tolerance(int bits);
   bool operator()(const T& a, const T& b)const;
};
```

`eps_tolerance` is the usual termination condition used with these root finding functions. Its `operator()` will return true when the relative distance between *a* and *b* is less than four times the machine epsilon for T, or 2^{1-bits}, whichever is the larger. In other words, you set *bits* to the number of bits of precision you want in the result. The minimal tolerance of *four times the machine epsilon of type T* is required to ensure that we get back a bracketing interval, since this must clearly be at greater than one epsilon in size. While in theory a maximum distance of twice machine epsilon is possible to achieve, in practice this results in a great deal of "thrashing" given that the function whose root is being found can only ever be accurate to 1 epsilon at best.

```
struct equal_floor
{
   equal_floor();
   template <class T> bool operator()(const T& a, const T& b)const;
};
```

This termination condition is used when you want to find an integer result that is the *floor* of the true root. It will terminate as soon as both ends of the interval have the same *floor*.

```
struct equal_ceil
{
   equal_ceil();
   template <class T> bool operator()(const T& a, const T& b)const;
};
```

This termination condition is used when you want to find an integer result that is the *ceil* of the true root. It will terminate as soon as both ends of the interval have the same *ceil*.

```
struct equal_nearest_integer
{
   equal_nearest_integer();
   template <class T> bool operator()(const T& a, const T& b)const;
};
```

This termination condition is used when you want to find an integer result that is the *closest* to the true root. It will terminate as soon as both ends of the interval round to the same nearest integer.

Implementation

The implementation of the bisection algorithm is extremely straightforward and not detailed here.

TOMS Algorithm 748: enclosing zeros of continuous functions is described in detail in:

Algorithm 748: Enclosing Zeros of Continuous Functions, G. E. Alefeld, F. A. Potra and Yixun Shi, ACM Transactions on Mathematica1 Software, Vol. 21. No. 3. September 1995. Pages 327-344.

The implementation here is a faithful translation of this paper into C++.

Root Finding With Derivatives: Newton-Raphson, Halley & Schröder

Synopsis

```
#include <boost/math/tools/roots.hpp>
```

```
namespace boost { namespace math {
namespace tools { // Note namespace boost::math::tools.
// Newton-Raphson
template <class F, class T>
T newton_raphson_iterate(F f, T guess, T min, T max, int digits);

template <class F, class T>
T newton_raphson_iterate(F f, T guess, T min, T max, int digits, boost::uintmax_t& max_iter);

// Halley
template <class F, class T>
T halley_iterate(F f, T guess, T min, T max, int digits);

template <class F, class T>
T halley_iterate(F f, T guess, T min, T max, int digits, boost::uintmax_t& max_iter);

// Schr''&#xf6;'''der
template <class F, class T>
T schroder_iterate(F f, T guess, T min, T max, int digits);

template <class F, class T>
T schroder_iterate(F f, T guess, T min, T max, int digits, boost::uintmax_t& max_iter);

template <class F, class Complex>
Complex complex_newton(F f, Complex guess, int max_iterations = std::numeric_limits<typename Com↲
plex::value_type>::digits);

template<class T>
auto quadratic_roots(T const & a, T const & b, T const & c);

}}} // namespaces boost::math::tools.
```

Description

These functions all perform iterative root-finding **using derivatives**:

- `newton_raphson_iterate` performs second-order Newton-Raphson iteration.
- `halley_iterate` and `schroder_iterate` perform third-order Halley and Schröder iteration.
- `complex_newton` performs Newton's method on complex-analytic functions.
- `solve_quadratic` solves quadratic equations using various tricks to keep catastrophic cancellation from occurring in computation of the discriminant.

Parameters of the real-valued root finding functions

F f	Type F must be a callable function object (or C++ lambda) that accepts one parameter and returns a std::pair, std::tuple, boost::tuple or boost::fusion::tuple:

	For second-order iterative method (Newton Raphson) the `tuple` should have **two** elements containing the evaluation of the function and its first derivative. For the third-order methods (Halley and Schröder) the `tuple` should have **three** elements containing the evaluation of the function and its first and second derivatives.
T guess	The initial starting value. A good guess is crucial to quick convergence!
T min	The minimum possible value for the result, this is used as an initial lower bracket.
T max	The maximum possible value for the result, this is used as an initial upper bracket.
int digits	The desired number of binary digits precision.
uintmax_t& max_iter	An optional maximum number of iterations to perform. On exit, this is updated to the actual number of iterations performed.

When using these functions you should note that:

- Default `max_iter = (std::numeric_limits<boost::uintmax_t>::max)()` is effectively 'iterate for ever'.
- They may be very sensitive to the initial guess, typically they converge very rapidly if the initial guess has two or three decimal digits correct. However convergence can be no better than bisect, or in some rare cases, even worse than bisect if the initial guess is a long way from the correct value and the derivatives are close to zero.
- These functions include special cases to handle zero first (and second where appropriate) derivatives, and fall back to bisect in this case. However, it is helpful if functor F is defined to return an arbitrarily small value *of the correct sign* rather than zero.
- If the derivative at the current best guess for the result is infinite (or very close to being infinite) then these functions may terminate prematurely. A large first derivative leads to a very small next step, triggering the termination condition. Derivative based iteration may not be appropriate in such cases.
- If the function is 'Really Well Behaved' (is monotonic and has only one root) the bracket bounds *min* and *max* may as well be set to the widest limits like zero and `numeric_limits<T>::max()`.
- But if the function more complex and may have more than one root or a pole, the choice of bounds is protection against jumping out to seek the 'wrong' root.
- These functions fall back to bisect if the next computed step would take the next value out of bounds. The bounds are updated after each step to ensure this leads to convergence. However, a good initial guess backed up by asymptotically-tight bounds will improve performance no end - rather than relying on bisection.
- The value of *digits* is crucial to good performance of these functions, if it is set too high then at best you will get one extra (unnecessary) iteration, and at worst the last few steps will proceed by bisection. Remember that the returned value can never be more accurate than *f(x)* can be evaluated, and that if *f(x)* suffers from cancellation errors as it tends to zero then the computed steps will be effectively random. The value of *digits* should be set so that iteration terminates before this point: remember that for second and third order methods the number of correct digits in the result is increasing quite substantially with each iteration, *digits* should be set by experiment so that the final iteration just takes the next value into the zone where *f(x)* becomes inaccurate. A good starting point for *digits* would be 0.6*D for Newton and 0.4*D for Halley or Shröder iteration, where D is `std::numeric_limits<T>::digits`.
- If you need some diagnostic output to see what is going on, you can `#define BOOST_MATH_INSTRUMENT` before the `#include <boost/math/tools/roots.hpp>`, and also ensure that display of all the significant digits with `cout.precision(std::numeric_limits<double>::digits10)`: or even possibly significant digits with `cout.precision(std::numeric_limits<double>::max_digits10)`: but be warned, this may produce copious output!
- Finally: you may well be able to do better than these functions by hand-coding the heuristics used so that they are tailored to a specific function. You may also be able to compute the ratio of derivatives used by these methods more efficiently than computing the derivatives themselves. As ever, algebraic simplification can be a big win.

Newton Raphson Method

Given an initial guess *x0* the subsequent values are computed using:

$$x_{N+1} = x_N - \frac{f(x)}{f'(x)}$$

Out of bounds steps revert to bisection of the current bounds.

Under ideal conditions, the number of correct digits doubles with each iteration.

Halley's Method

Given an initial guess *x0* the subsequent values are computed using:

$$x_{N+1} = x_N - \frac{2f(x)f'(x)}{2(f'(x))^2 - f(x)f''(x)}$$

Over-compensation by the second derivative (one which would proceed in the wrong direction) causes the method to revert to a Newton-Raphson step.

Out of bounds steps revert to bisection of the current bounds.

Under ideal conditions, the number of correct digits trebles with each iteration.

Schröder's Method

Given an initial guess x0 the subsequent values are computed using:

$$x_{N+1} = x_N - \frac{f(x)}{f'(x)} - \frac{f''(x)(f(x))^2}{2(f'(x))^3}$$

Over-compensation by the second derivative (one which would proceed in the wrong direction) causes the method to revert to a Newton-Raphson step. Likewise a Newton step is used whenever that Newton step would change the next value by more than 10%.

Out of bounds steps revert to bisection of the current bounds.

Under ideal conditions, the number of correct digits trebles with each iteration.

This is Schröder's general result (equation 18 from Stewart, G. W. "On Infinitely Many Algorithms for Solving Equations." English translation of Schröder's original paper. College Park, MD: University of Maryland, Institute for Advanced Computer Studies, Department of Computer Science, 1993.)

This method guarantees at least quadratic convergence (the same as Newton's method), and is known to work well in the presence of multiple roots: something that neither Newton nor Halley can do.

The complex Newton method works slightly differently than the rest of the methods: Since there is no way to bracket roots in the complex plane, the `min` and `max` arguments are not accepted. Failure to reach a root is communicated by returning `nan`s. Remember that if a function has many roots, then which root the complex Newton's method converges to is essentially impossible to predict a priori; see the Newton's fractal for more information.

Finally, the derivative of *f* must be continuous at the root or else non-roots can be found; see here for an example.

An example usage of `complex_newton` is given in `examples/daubechies_coefficients.cpp`.

Quadratics

To solve a quadratic $ax^2 + bx + c = 0$, we may use

```
auto [x0, x1] = boost::math::tools::quadratic_roots(a, b, c);
```

If the roots are real, they are arranged so that `x0` ≤ `x1`. If the roots are complex and the inputs are real, `x0` and `x1` are both `std::numeric_limits<Real>::quiet_NaN()`. In this case we must cast `a`, `b` and `c` to a complex type to extract the complex roots. If `a`, `b` and `c` are integral, then the roots are of type double. The routine is much faster if the fused-multiply-add instruction is available on your architecture. If the fma is not available, the function resorts to slow emulation. Finally, speed is improved if you compile for your particular architecture. For instance, if you compile without any architecture flags, then the `std::fma` call compiles down to `call _fma`, which dynamically chooses to emulate or execute the `vfmadd132sd` instruction based on the capabilities of the architecture. If instead, you compile with (say) `-march=native` then no dynamic choice is made: The `vfmadd132sd` instruction is always executed if available and emulation is used if not.

Examples

See root-finding examples.

Examples of Root-Finding (with and without derivatives)

The examples demonstrate how to use the various tools for root finding.

We start with the simple cube root function `cbrt` (C++ standard function name cbrt) showing root finding without derivatives.

We then show how use of derivatives can improve the speed of convergence.

(But these examples are only a demonstration and do not try to make the ultimate improvements of an 'industrial-strength' implementation, for example, of `boost::math::cbrt`, mainly by using a better computed initial 'guess' at cbrt.hpp).

Then we show how a higher root (fifth root) $^5\sqrt{}$ can be computed, and in root_finding_n_example.cpp a generic method for the nth root that constructs the derivatives at compile-time.

These methods should be applicable to other functions that can be differentiated easily.

Finding the Cubed Root With and Without Derivatives

First some `#includes` that will be needed.

```
#include <boost/math/tools/roots.hpp>
//using boost::math::policies::policy;
//using boost::math::tools::newton_raphson_iterate;
//using boost::math::tools::halley_iterate; //
//using boost::math::tools::eps_tolerance; // Binary functor for specified number of bits.
//using boost::math::tools::bracket_and_solve_root;
//using boost::math::tools::toms748_solve;

#include <boost/math/special_functions/next.hpp> // For float_distance.
#include <tuple> // for std::tuple and std::make_tuple.
#include <boost/math/special_functions/cbrt.hpp> // For boost::math::cbrt.
```

Tip

For clarity, `using` statements are provided to list what functions are being used in this example: you can, of course, partly or fully qualify the names in other ways. (For your application, you may wish to extract some parts into header files, but you should never use `using` statements globally in header files).

Let's suppose we want to find the root of a number *a*, and to start, compute the cube root.

So the equation we want to solve is:

$f(x) = x^3 - a$

We will first solve this without using any information about the slope or curvature of the cube root function.

Fortunately, the cube-root function is 'Really Well Behaved' in that it is monotonic and has only one root (we leave negative values 'as an exercise for the student').

We then show how adding what we can know about this function, first just the slope or 1st derivative *f'(x)*, will speed homing in on the solution.

Lastly, we show how adding the curvature *f''(x)* too will speed convergence even more.

Cube root function without derivatives

First we define a function object (functor):

```
template <class T>
struct cbrt_functor_noderiv
{
  //  cube root of x using only function - no derivatives.
  cbrt_functor_noderiv(T const& to_find_root_of) : a(to_find_root_of)
  { /* Constructor just stores value a to find root of. */ }
  T operator()(T const& x)
  {
    T fx = x*x*x - a; // Difference (estimate x^3 - a).
    return fx;
  }
private:
  T a; // to be 'cube_rooted'.
};
```

Implementing the cube-root function itself is fairly trivial now: the hardest part is finding a good approximation to begin with. In this case we'll just divide the exponent by three. (There are better but more complex guess algorithms used in 'real life'.)

```
template <class T>
T cbrt_noderiv(T x)
{
  // return cube root of x using bracket_and_solve (no derivatives).
  using namespace std;                          // Help ADL of std functions.
  using namespace boost::math::tools;           // For bracket_and_solve_root.

  int exponent;
  frexp(x, &exponent);                          // Get exponent of z (ignore mantissa).
  T guess = ldexp(1., exponent/3);              // Rough guess is to divide the exponent by ↵
three.
  T factor = 2;                                 // How big steps to take when searching.

  const boost::uintmax_t maxit = 20;            // Limit to maximum iterations.
  boost::uintmax_t it = maxit;                 // Initally our chosen max iterations, but up↵
dated with actual.
  bool is_rising = true;                        // So if result if guess^3 is too low, then ↵
try increasing guess.
  int digits = std::numeric_limits<T>::digits;  // Maximum possible binary digits accuracy ↵
for type T.
  // Some fraction of digits is used to control how accurate to try to make the result.
  int get_digits = digits - 3;                 // We have to have a non-zero interval at each ↵
step, so
                                                // maximum accuracy is digits - 1.  But we ↵
also have to
                                                // allow for inaccuracy in f(x), otherwise ↵
the last few
                                                // iterations just thrash around.
  eps_tolerance<T> tol(get_digits);             // Set the tolerance.
  std::pair<T, T> r = bracket_and_solve_root(cbrt_functor_node↵
riv<T>(x), guess, factor, is_rising, tol, it);
  return r.first + (r.second - r.first)/2;     // Midway between brackets is our result, if ↵
necessary we could
                                                // return the result as an interval here.
}
```

Note

The final parameter specifying a maximum number of iterations is optional. However, it defaults to `boost::uintmax_t maxit = (std::numeric_limits<boost::uintmax_t>::max)();` which is `18446744073709551615` and is more than anyone would wish to wait for!

So it may be wise to chose some reasonable estimate of how many iterations may be needed, In this case the function is so well behaved that we can chose a low value of 20.

Internally when Boost.Math uses these functions, it sets the maximum iterations to `policies::get_max_root_iterations<Policy>();`.

Should we have wished we can show how many iterations were used in `bracket_and_solve_root` (this information is lost outside `cbrt_noderiv`), for example with:

```
if (it >= maxit)
{
  std::cout << "Unable to locate solution in " << maxit << " iterations:"
    " Current best guess is between " << r.first << " and " << r.second << std::endl;
}
else
{
  std::cout << "Converged after " << it << " (from maximum of " << maxit << " itera↵
tions)." << std::endl;
}
```

for output like

```
Converged after 11 (from maximum of 20 iterations).
```

This snippet from `main()` in root_finding_example.cpp shows how it can be used.

```
try
{
  double threecubed = 27.;   // Value that has an *exactly representable* integer cube root.
  double threecubedp1 = 28.; // Value whose cube root is *not* exactly representable.

  std::cout << "cbrt(28) " << boost::math::cbrt(28.) << std::endl; // boost::math:: version ↵
of cbrt.
  std::cout << "std::cbrt(28) " << std::cbrt(28.) << std::endl;    // std:: version of cbrt.
  std::cout <<" cast double " << stat↵
ic_cast<double>(3.0365889718756625194208095785056696355814539772481111) << std::endl;

  // Cube root using bracketing:
  double r = cbrt_noderiv(threecubed);
  std::cout << "cbrt_noderiv(" << threecubed << ") = " << r << std::endl;
  r = cbrt_noderiv(threecubedp1);
  std::cout << "cbrt_noderiv(" << threecubedp1 << ") = " << r << std::endl;
```

```
  cbrt_noderiv(27) = 3
  cbrt_noderiv(28) = 3.0365889718756618
```

The result of `bracket_and_solve_root` is a pair of values that could be displayed.

The number of bits separating them can be found using `float_distance(r.first, r.second)`. The distance is zero (closest representable) for $3^3 = 27$ but `float_distance(r.first, r.second) = 3` for cube root of 28 with this function. The result (avoiding overflow) is midway between these two values.

Cube root function with 1st derivative (slope)

We now solve the same problem, but using more information about the function, to show how this can speed up finding the best estimate of the root.

For the root function, the 1st differential (the slope of the tangent to a curve at any point) is known.

This algorithm is similar to this nth root algorithm.

If you need some reminders, then derivatives of elementary functions may help.

Using the rule that the derivative of x^n for positive n (actually all nonzero n) is $n\ x^{n-1}$, allows us to get the 1st differential as $3x^2$.

To see how this extra information is used to find a root, view Newton-Raphson iterations and the animation.

We define a better functor `cbrt_functor_deriv` that returns both the evaluation of the function to solve, along with its first derivative:

To '*return*' two values, we use a std::pair of floating-point values.

```
template <class T>
struct cbrt_functor_deriv
{ // Functor also returning 1st derivative.
  cbrt_functor_deriv(T const& to_find_root_of) : a(to_find_root_of)
  { // Constructor stores value a to find root of,
    // for example: calling cbrt_functor_deriv<T>(a) to use to get cube root of a.
  }
  std::pair<T, T> operator()(T const& x)
  {
    // Return both f(x) and f'(x).
    T fx = x*x*x - a;                // Difference (estimate x^3 - value).
    T dx =  3 * x*x;                 // 1st derivative = 3x^2.
    return std::make_pair(fx, dx);   // 'return' both fx and dx.
  }
private:
  T a;                               // Store value to be 'cube_rooted'.
};
```

Our cube root function is now:

```
template <class T>
T cbrt_deriv(T x)
{
  // return cube root of x using 1st derivative and Newton_Raphson.
  using namespace boost::math::tools;
  int exponent;
  frexp(x, &exponent);                                  // Get exponent of z (ignore mantissa).
  T guess = ldexp(1., exponent/3);                    // Rough guess is to divide the exponent ↵
by three.
  T min = ldexp(0.5, exponent/3);                       // Minimum possible value is half our ↵
guess.
  T max = ldexp(2., exponent/3);                       // Maximum possible value is twice our ↵
guess.
  const int digits = std::numeric_limits<T>::digits;  // Maximum possible binary digits accur↵
acy for type T.
  int get_digits = static_cast<int>(digits * 0.6);    // Accuracy doubles with each step, so ↵
stop when we have
                                                        // just over half the digits correct.
  const boost::uintmax_t maxit = 20;
  boost::uintmax_t it = maxit;
  T result = newton_raphson_iterate(cbrt_functor_deriv<T>(x), guess, min, max, get_digits, it);
  return result;
}
```

The result of `newton_raphson_iterate` function is a single value.

Tip

There is a compromise between accuracy and speed when chosing the value of `digits`. It is tempting to simply chose `std::numeric_limits<T>::digits`, but this may mean some inefficient and unnecessary iterations as the function thrashes around trying to locate the last bit. In theory, since the precision doubles with each step it is sufficient to stop when half the bits are correct: as the last step will have doubled that to full precision. Of course the function has no way to tell if that is actually the case unless it does one more step to be sure. In practice setting the precision to slightly more than `std::numeric_limits<T>::digits / 2` is a good choice.

Note that it is up to the caller of the function to check the iteration count after the call to see if iteration stoped as a result of running out of iterations rather than meeting the required precision.

Using the test data in /test/test_cbrt.cpp this found the cube root exact to the last digit in every case, and in no more than 6 iterations at double precision. However, you will note that a high precision was used in this example, exactly what was warned against earlier on in these docs! In this particular case it is possible to compute *f(x)* exactly and without undue cancellation error, so a high limit is not too much of an issue.

However, reducing the limit to `std::numeric_limits<T>::digits * 2 / 3` gave full precision in all but one of the test cases (and that one was out by just one bit). The maximum number of iterations remained 6, but in most cases was reduced by one.

Note also that the above code omits a probable optimization by computing z^2 and reusing it, omits error handling, and does not handle negative values of z correctly. (These are left as the customary exercise for the reader!)

The `boost::math::cbrt` function also includes these and other improvements: most importantly it uses a much better initial guess which reduces the iteration count to just 1 in almost all cases.

Cube root with 1st & 2nd derivative (slope & curvature)

Next we define yet another even better functor `cbrt_functor_2deriv` that returns both the evaluation of the function to solve, along with its first **and second** derivative:

$$f''(x) = 6x$$

using information about both slope and curvature to speed convergence.

To *'return'* three values, we use a `tuple` of three floating-point values:

```
template <class T>
struct cbrt_functor_2deriv
{
  // Functor returning both 1st and 2nd derivatives.
  cbrt_functor_2deriv(T const& to_find_root_of) : a(to_find_root_of)
  { // Constructor stores value a to find root of, for example:
    // calling cbrt_functor_2deriv<T>(x) to get cube root of x,
  }
  std::tuple<T, T, T> operator()(T const& x)
  {
    // Return both f(x) and f'(x) and f''(x).
    T fx = x*x*x - a;                     // Difference (estimate x^3 - value).
    T dx = 3 * x*x;                       // 1st derivative = 3x^2.
    T d2x = 6 * x;                        // 2nd derivative = 6x.
    return std::make_tuple(fx, dx, d2x);  // 'return' fx, dx and d2x.
  }
private:
  T a; // to be 'cube_rooted'.
};
```

Our cube root function is now:

```
template <class T>
T cbrt_2deriv(T x)
{
  // return cube root of x using 1st and 2nd derivatives and Halley.
  //using namespace std;  // Help ADL of std functions.
  using namespace boost::math::tools;
  int exponent;
  frexp(x, &exponent);                                // Get exponent of z (ignore mantissa).
  T guess = ldexp(1., exponent/3);                   // Rough guess is to divide the exponent ↵
by three.
  T min = ldexp(0.5, exponent/3);                     // Minimum possible value is half our ↵
guess.
  T max = ldexp(2., exponent/3);                     // Maximum possible value is twice our ↵
guess.
  const int digits = std::numeric_limits<T>::digits;  // Maximum possible binary digits accur↵
acy for type T.
  // digits used to control how accurate to try to make the result.
  int get_digits = static_cast<int>(digits * 0.4);    // Accuracy triples with each step, so ↵
stop when just
                                                     // over one third of the digits are cor↵
rect.
  boost::uintmax_t maxit = 20;
  T result = halley_iterate(cbrt_functor_2deriv<T>(x), guess, min, max, get_digits, maxit);
  return result;
}
```

The function `halley_iterate` also returns a single value, and the number of iterations will reveal if it met the convergence criterion set by `get_digits`.

The no-derivative method gives a result of

```
cbrt_noderiv(28) = 3.0365889718756618
```

with a 3 bits distance between the bracketed values, whereas the derivative methods both converge to a single value

```
cbrt_2deriv(28) = 3.0365889718756627
```

which we can compare with the boost::math::cbrt

```
cbrt(28)              = 3.0365889718756627
```

Note that the iterations are set to stop at just one-half of full precision, and yet, even so, not one of the test cases had a single bit wrong. What's more, the maximum number of iterations was now just 4.

Just to complete the picture, we could have called `schroder_iterate` in the last example: and in fact it makes no difference to the accuracy or number of iterations in this particular case. However, the relative performance of these two methods may vary depending upon the nature of *f(x)*, and the accuracy to which the initial guess can be computed. There appear to be no generalisations that can be made except "try them and see".

Finally, had we called `cbrt` with NTL::RR set to 1000 bit precision (about 300 decimal digits), then full precision can be obtained with just 7 iterations. To put that in perspective, an increase in precision by a factor of 20, has less than doubled the number of iterations. That just goes to emphasise that most of the iterations are used up getting the first few digits correct: after that these methods can churn out further digits with remarkable efficiency.

Or to put it another way: *nothing beats a really good initial guess!*

Full code of this example is at root_finding_example.cpp,

Using C++11 Lambda's

Since all the root finding functions accept a function-object, they can be made to work (often in a lot less code) with C++11 lambda's. Here's the much reduced code for our "toy" cube root function:

```
template <class T>
T cbrt_2deriv_lambda(T x)
{
   // return cube root of x using 1st and 2nd derivatives and Halley.
   //using namespace std;  // Help ADL of std functions.
   using namespace boost::math::tools;
   int exponent;
   frexp(x, &exponent);                                // Get exponent of z (ignore mantissa).
   T guess = ldexp(1., exponent / 3);                    // Rough guess is to divide the expo↲
nent by three.
   T min = ldexp(0.5, exponent / 3);                     // Minimum possible value is half ↲
our guess.
   T max = ldexp(2., exponent / 3);                      // Maximum possible value is twice ↲
our guess.
   const int digits = std::numeric_limits<T>::digits;  // Maximum possible binary digits accur↲
acy for type T.
   // digits used to control how accurate to try to make the result.
   int get_digits = static_cast<int>(digits * 0.4);    // Accuracy triples with each step, so ↲
stop when just
   // over one third of the digits are correct.
   boost::uintmax_t maxit = 20;
   T result = halley_iterate(
      // lambda function:
      [x](const T& g){ return std::make_tuple(g * g * g - x, 3 * g * g, 6 * g); },
      guess, min, max, get_digits, maxit);
   return result;
}
```

Full code of this example is at root_finding_example.cpp,

Computing the Fifth Root

Let's now suppose we want to find the **fifth root** of a number *a*.

The equation we want to solve is :

$f(x) = x^5 - a$

If your differentiation is a little rusty (or you are faced with an function whose complexity makes differentiation daunting), then you can get help, for example, from the invaluable WolframAlpha site.

For example, entering the commmand: `differentiate x ^ 5`

or the Wolfram Language command: `D[x ^ 5, x]`

gives the output: `d/dx(x ^ 5) = 5 x ^ 4`

and to get the second differential, enter: `second differentiate x ^ 5`

or the Wolfram Language command: `D[x ^ 5, { x, 2 }]`

to get the output: `d ^ 2 / dx ^ 2(x ^ 5) = 20 x ^ 3`

To get a reference value, we can enter: `fifth root 3126`

or: `N[3126 ^ (1 / 5), 50]`

to get a result with a precision of 50 decimal digits:

5.0003199590478625588206333405631053401128722314376

(We could also get a reference value using multiprecision root).

The 1st and 2nd derivatives of x^5 are:

$f'(x) = 5x^4$

$f''(x) = 20x^3$

Using these expressions for the derivatives, the functor is:

```
template <class T>
struct fifth_functor_2deriv
{
  // Functor returning both 1st and 2nd derivatives.
  fifth_functor_2deriv(T const& to_find_root_of) : a(to_find_root_of)
  { /* Constructor stores value a to find root of, for example: */ }

  std::tuple<T, T, T> operator()(T const& x)
  {
    // Return both f(x) and f'(x) and f''(x).
    T fx = boost::math::pow<5>(x) - a;    // Difference (estimate x^3 - value).
    T dx = 5 * boost::math::pow<4>(x);    // 1st derivative = 5x^4.
    T d2x = 20 * boost::math::pow<3>(x);  // 2nd derivative = 20 x^3
    return std::make_tuple(fx, dx, d2x);  // 'return' fx, dx and d2x.
  }
private:
  T a;                                    // to be 'fifth_rooted'.
}; // struct fifth_functor_2deriv
```

Our fifth-root function is now:

```
template <class T>
T fifth_2deriv(T x)
{
  // return fifth root of x using 1st and 2nd derivatives and Halley.
  using namespace std;                  // Help ADL of std functions.
  using namespace boost::math::tools;   // for halley_iterate.

  int exponent;
  frexp(x, &exponent);                  // Get exponent of z (ignore mantissa).
  T guess = ldexp(1., exponent / 5);    // Rough guess is to divide the exponent by five.
  T min = ldexp(0.5, exponent / 5);     // Minimum possible value is half our guess.
  T max = ldexp(2., exponent / 5);      // Maximum possible value is twice our guess.
  // Stop when slightly more than one of the digits are correct:
  const int digits = static_cast<int>(std::numeric_limits<T>::digits * 0.4);
  const boost::uintmax_t maxit = 50;
  boost::uintmax_t it = maxit;
  T result = halley_iterate(fifth_functor_2deriv<T>(x), guess, min, max, digits, it);
  return result;
}
```

Full code of this example is at root_finding_example.cpp and root_finding_n_example.cpp.

Root-finding using Boost.Multiprecision

The apocryphally astute reader might, by now, be asking "How do we know if this computes the 'right' answer?".

For most values, there is, sadly, no 'right' answer. This is because values can only rarely be *exactly represented* by C++ floating-point types. What we do want is the 'best' representation - one that is the nearest representable value. (For more about how numbers are represented see Floating point).

Of course, we might start with finding an external reference source like Wolfram Alpha, as above, but this is not always possible.

Another way to reassure is to compute 'reference' values at higher precision with which to compare the results of our iterative computations using built-in like `double`. They should agree within the tolerance that was set.

The result of `static_casting` to `double` from a higher-precision type like `cpp_bin_float_50` is guaranteed to be the **nearest representable** `double` value.

For example, the cube root functions in our example for `cbrt(28.)` compute

`std::cbrt<double>(28.) = 3.0365889718756627`

WolframAlpha says `3.036588971875662519420809578505669635581453977248111123242141...`

`static_cast<double>(3.03658897187566251942080957850) = 3.0365889718756627`

This example `cbrt(28.) = 3.0365889718756627`

Tip

To ensure that all potentially significant decimal digits are displayed use `std::numeric_limits<T>::max_digits10` (or if not available on older platforms or compilers use `2+std::numeric_limits<double>::digits*3010/10000`).

Ideally, values should agree to `std::numeric-limits<T>::digits10` decimal digits.

This also means that a 'reference' value to be **input** or `static_cast` should have at least `max_digits10` decimal digits (17 for 64-bit `double`).

If we wish to compute **higher-precision values** then, on some platforms, we may be able to use `long double` with a higher precision than `double` to compare with the very common `double` and/or a more efficient built-in quad floating-point type like `__float128`.

Almost all platforms can easily use Boost.Multiprecision, for example, cpp_dec_float or a binary type cpp_bin_float types, to compute values at very much higher precision.

Note

With multiprecision types, it is debatable whether to use the type `T` for computing the initial guesses. Type `double` is like to be accurate enough for the method used in these examples. This would limit the exponent range of possible values to that of `double`. There is also the cost of conversion to and from type `T` to consider. In these examples, `double` is used via `typedef double guess_type`.

Since the functors and functions used above are templated on the value type, we can very simply use them with any of the Boost.Multiprecision types. As a reminder, here's our toy cube root function using 2 derivatives and C++11 lambda functions to find the root:

```
template <class T>
T cbrt_2deriv_lambda(T x)
{
   // return cube root of x using 1st and 2nd derivatives and Halley.
   //using namespace std;  // Help ADL of std functions.
   using namespace boost::math::tools;
   int exponent;
   frexp(x, &exponent);                                  // Get exponent of z (ignore mantissa).
   T guess = ldexp(1., exponent / 3);                    // Rough guess is to divide the expo↵
nent by three.
   T min = ldexp(0.5, exponent / 3);                     // Minimum possible value is half ↵
our guess.
   T max = ldexp(2., exponent / 3);                      // Maximum possible value is twice ↵
our guess.
  const int digits = std::numeric_limits<T>::digits;  // Maximum possible binary digits accur↵
acy for type T.
   // digits used to control how accurate to try to make the result.
   int get_digits = static_cast<int>(digits * 0.4);    // Accuracy triples with each step, so ↵
stop when just
   // over one third of the digits are correct.
   boost::uintmax_t maxit = 20;
   T result = halley_iterate(
      // lambda function:
      [x](const T& g){ return std::make_tuple(g * g * g - x, 3 * g * g, 6 * g); },
      guess, min, max, get_digits, maxit);
   return result;
}
```

Some examples below are 50 decimal digit decimal and binary types (and on some platforms a much faster `float128` or `quad_float` type) that we can use with these includes:

```
#include <boost/multiprecision/cpp_bin_float.hpp> // For cpp_bin_float_50.
#include <boost/multiprecision/cpp_dec_float.hpp> // For cpp_dec_float_50.
#ifndef _MSC_VER  // float128 is not yet supported by Microsoft compiler at 2013.
#  include <boost/multiprecision/float128.hpp> // Requires libquadmath.
#endif
```

Some using statements simplify their use:

```
  using boost::multiprecision::cpp_dec_float_50; // decimal.
  using boost::multiprecision::cpp_bin_float_50; // binary.
#ifndef _MSC_VER  // Not supported by Microsoft compiler.
  using boost::multiprecision::float128;
#endif
```

They can be used thus:

```
std::cout.precision(std::numeric_limits<cpp_dec_float_50>::digits10);

cpp_dec_float_50 two = 2; //
cpp_dec_float_50  r = cbrt_2deriv(two);
std::cout << "cbrt(" << two << ") = " << r << std::endl;

r = cbrt_2deriv(2.); // Passing a double, so ADL will compute a double precision result.
std::cout << "cbrt(" << two << ") = " << r << std::endl;
// cbrt(2) = 1.2599210498948731906665443602832965552806854248047 'wrong' from digits 17 onwards!
r = cbrt_2deriv(static_cast<cpp_dec_float_50>(2.)); // Passing a cpp_dec_float_50,
// so will compute a cpp_dec_float_50 precision result.
std::cout << "cbrt(" << two << ") = " << r << std::endl;
r = cbrt_2deriv<cpp_dec_float_50>(2.); // Explictly a cpp_dec_float_50, so will compute a ↵
cpp_dec_float_50 precision result.
std::cout << "cbrt(" << two << ") = " << r << std::endl;
// cpp_dec_float_50 1.2599210498948731647672106072782283505702514647015
```

A reference value computed by Wolfram Alpha is

```
N[2^(1/3), 50]  1.2599210498948731647672106072782283505702514647015
```

which agrees exactly.

To **show** values to their full precision, it is necessary to adjust the `std::ostream` `precision` to suit the type, for example:

```
template <typename T>
T show_cube_root(T value)
{ // Demonstrate by printing the root using all definitely significant digits.
  std::cout.precision(std::numeric_limits<T>::digits10);
  T r = cbrt_2deriv(value);
  std::cout << "value = " << value << ", cube root =" << r << std::endl;
  return r;
}
```

```
show_cube_root(2.);
show_cube_root(2.L);
show_cube_root(two);
```

which outputs:

```
cbrt(2) = 1.2599210498948731647672106072782283505702514647015

value = 2, cube root =1.25992104989487
value = 2, cube root =1.25992104989487
value = 2, cube root =1.2599210498948731647672106072782283505702514647015
```

Tip

Be **very careful** about the floating-point type `T` that is passed to the root-finding function. Carelessly passing a integer by writing `cpp_dec_float_50 r = cbrt_2deriv(2);` or `show_cube_root(2);` will provoke many warnings and compile errors.

Even `show_cube_root(2.F);` will produce warnings because `typedef double guess_type` defines the type used to compute the guess and bracket values as `double`.

Even more treacherous is passing a `double` as in `cpp_dec_float_50 r = cbrt_2deriv(2.);` which silently gives the 'wrong' result, computing a `double` result and **then** converting to `cpp_dec_float_50`! All digits beyond `max_digits10` will be incorrect. Making the `cbrt` type explicit with `cbrt_2deriv<cpp_dec_float_50>(2.);` will give you the desired 50 decimal digit precision result.

Full code of this example is at root_finding_multiprecision_example.cpp.

Generalizing to Compute the nth root

If desired, we can now further generalize to compute the *n*th root by computing the derivatives **at compile-time** using the rules for differentiation and `boost::math::pow<N>` where template parameter `N` is an integer and a compile time constant. Our functor and function now have an additional template parameter `N`, for the root required.

Note

Since the powers and derivatives are fixed at compile time, the resulting code is as efficient as as if hand-coded as the cube and fifth-root examples above. A good compiler should also optimise any repeated multiplications.

Our *n*th root functor is

```
template <int N, class T = double>
struct nth_functor_2deriv
{ // Functor returning both 1st and 2nd derivatives.
  BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");
  BOOST_STATIC_ASSERT_MSG((N > 0) == true, "root N must be > 0!");

  nth_functor_2deriv(T const& to_find_root_of) : a(to_find_root_of)
  { /* Constructor stores value a to find root of, for example: */ }

  // using boost::math::tuple; // to return three values.
  std::tuple<T, T, T> operator()(T const& x)
  {
    // Return f(x), f'(x) and f''(x).
    using boost::math::pow;
    T fx = pow<N>(x) - a;                     // Difference (estimate x^n - a).
    T dx = N * pow<N - 1>(x);                 // 1st derivative f'(x).
    T d2x = N * (N - 1) * pow<N - 2 >(x);     // 2nd derivative f''(x).

    return std::make_tuple(fx, dx, d2x);     // 'return' fx, dx and d2x.
  }
private:
  T a;                                        // to be 'nth_rooted'.
};
```

and our *n*th root function is

```
template <int N, class T = double>
T nth_2deriv(T x)
{ // return nth root of x using 1st and 2nd derivatives and Halley.

  using namespace std;  // Help ADL of std functions.
  using namespace boost::math::tools; // For halley_iterate.

  BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");
  BOOST_STATIC_ASSERT_MSG((N > 0) == true, "root N must be > 0!");
  BOOST_STATIC_ASSERT_MSG((N > 1000) == false, "root N is too big!");

  typedef double guess_type; // double may restrict (exponent) range for a multiprecision T?

  int exponent;
  frexp(static_cast<guess_type>(x), &exponent);                // Get exponent of z (ignore ↵
mantissa).
  T guess = ldexp(static_cast<guess_type>(1.), exponent / N);   // Rough guess is to divide ↵
the exponent by n.
  T min = ldexp(static_cast<guess_type>(1.) / 2, exponent / N); // Minimum possible value is ↵
half our guess.
  T max = ldexp(static_cast<guess_type>(2.), exponent / N);     // Maximum possible value is ↵
twice our guess.

  int digits = std::numeric_limits<T>::digits * 0.4;           // Accuracy triples with each ↵
step, so stop when
                                                                // slightly more than one ↵
third of the digits are correct.
  const boost::uintmax_t maxit = 20;
  boost::uintmax_t it = maxit;
  T result = halley_iterate(nth_functor_2deriv<N, T>(x), guess, min, max, digits, it);
  return result;
}
```

```
    show_nth_root<5, double>(2.);
    show_nth_root<5, long double>(2.);
#ifndef _MSC_VER  // float128 is not supported by Microsoft compiler 2013.
    show_nth_root<5, float128>(2);
#endif
    show_nth_root<5, cpp_dec_float_50>(2); // dec
    show_nth_root<5, cpp_bin_float_50>(2); // bin
```

produces an output similar to this

```
 Using MSVC 2013

nth Root finding Example.
Type double value = 2, 5th root = 1.14869835499704
Type long double value = 2, 5th root = 1.14869835499704
Type class boost::multiprecision::number<class boost::multipreci↵
sion::backends::cpp_dec_float<50,int,void>,1> value = 2,
  5th root = 1.14869835499703500679862694677792758944385088909780
Type class boost::multiprecision::number<class boost::multipreci↵
sion::backends::cpp_bin_float<50,10,void,int,0,0>,0> value = 2,
  5th root = 1.14869835499703500679862694677792758944385088909780
```

Tip

Take care with the type passed to the function. It is best to pass a `double` or greater-precision floating-point type.

Passing an integer value, for example, `nth_2deriv<5>(2)` will be rejected, while `nth_2deriv<5, double>(2)` converts the integer to `double`.

Avoid passing a `float` value that will provoke warnings (actually spurious) from the compiler about potential loss of data, as noted above.

Warning

Asking for unreasonable roots, for example, `show_nth_root<1000000>(2.);` may lead to Loss of significance like `Type double value = 2, 1000000th root = 1.00000069314783`. Use of the the `pow` function is more sensible for this unusual need.

Full code of this example is at root_finding_n_example.cpp.

A More complex example - Inverting the Elliptic Integrals

The arc length of an ellipse with radii *a* and *b* is given by:

```
L(a, b) = 4aE(k)
```

with:

$$k = \sqrt{(1 - b^2/a^2)}$$

where *E(k)* is the complete elliptic integral of the second kind - see ellint_2.

Let's suppose we know the arc length and one radii, we can then calculate the other radius by inverting the formula above. We'll begin by encoding the above formula into a functor that our root-finding algorithms can call.

Note that while not completely obvious from the formula above, the function is completely symmetrical in the two radii - which can be interchanged at will - in this case we need to make sure that `a >= b` so that we don't accidentally take the square root of a negative number:

```
template <typename T = double>
struct elliptic_root_functor_noderiv
{ //  Nth root of x using only function - no derivatives.
   elliptic_root_functor_noderiv(T const& arc, T const& radius) : m_arc(arc), m_radius(radius)
   { // Constructor just stores value a to find root of.
   }
   T operator()(T const& x)
   {
      using std::sqrt;
     // return the difference between required arc-length, and the calculated arc-length for an
      // ellipse with radii m_radius and x:
      T a = (std::max)(m_radius, x);
      T b = (std::min)(m_radius, x);
      T k = sqrt(1 - b * b / (a * a));
      return 4 * a * boost::math::ellint_2(k) - m_arc;
   }
private:
   T m_arc;     // length of arc.
   T m_radius;  // one of the two radii of the ellipse
}; // template <class T> struct elliptic_root_functor_noderiv
```

We'll also need a decent estimate to start searching from, the approximation:

```
L(a, b) ≈ 4√(a² + b²)
```

Is easily inverted to give us what we need, which using derivative-free root finding leads to the algorithm:

```
template <class T = double>
T elliptic_root_noderiv(T radius, T arc)
{ // return the other radius of an ellipse, given one radii and the arc-length
   using namespace std;  // Help ADL of std functions.
   using namespace boost::math::tools; // For bracket_and_solve_root.

   T guess = sqrt(arc * arc / 16 - radius * radius);
   T factor = 1.2;                     // How big steps to take when searching.

   const boost::uintmax_t maxit = 50;  // Limit to maximum iterations.
  boost::uintmax_t it = maxit;       // Initally our chosen max iterations, but updated with ↵
actual.
  bool is_rising = true;             // arc-length increases if one radii increases, so func↵
tion is rising
   // Define a termination condition, stop when nearly all digits are correct, but allow for
   // the fact that we are returning a range, and must have some inaccuracy in the elliptic ↵
integral:
   eps_tolerance<T> tol(std::numeric_limits<T>::digits - 2);
   // Call bracket_and_solve_root to find the solution, note that this is a rising function:
   std::pair<T, T> r = bracket_and_solve_root(elliptic_root_functor_noderiv<T>(arc, radi↵
us), guess, factor, is_rising, tol, it);
   // Result is midway between the endpoints of the range:
   return r.first + (r.second - r.first) / 2;
} // template <class T> T elliptic_root_noderiv(T x)
```

This function generally finds the root within 8-10 iterations, so given that the runtime is completely dominated by the cost of calling the ellliptic integral it would be nice to reduce that count somewhat. We'll try to do that by using a derivative-based method; the derivatives of this function are rather hard to work out by hand, but fortunately Wolfram Alpha can do the grunt work for us to give:

```
d/da L(a, b) = 4(a²E(k) - b²K(k)) / (a² - b²)
```

Note that now we have **two** elliptic integral calls to get the derivative, so our functor will be at least twice as expensive to call as the derivative-free one above: we'll have to reduce the iteration count quite substantially to make a difference!

Here's the revised functor:

```
template <class T = double>
struct elliptic_root_functor_1deriv
{ // Functor also returning 1st derviative.
   BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");

   elliptic_root_functor_1deriv(T const& arc, T const& radius) : m_arc(arc), m_radius(radius)
   { // Constructor just stores value a to find root of.
   }
   std::pair<T, T> operator()(T const& x)
   {
      using std::sqrt;
     // Return the difference between required arc-length, and the calculated arc-length for an
      // ellipse with radii m_radius and x, plus it's derivative.
     // See http://www.wolframalpha.com/input/?i=d%2Fda+[4+*+a+*+EllipticE%281+-+b^2%2Fa^2%29]
      // We require two elliptic integral calls, but from these we can calculate both
      // the function and it's derivative:
      T a = (std::max)(m_radius, x);
      T b = (std::min)(m_radius, x);
      T a2 = a * a;
      T b2 = b * b;
      T k = sqrt(1 - b2 / a2);
      T Ek = boost::math::ellint_2(k);
      T Kk = boost::math::ellint_1(k);
      T fx = 4 * a * Ek - m_arc;
      T dfx = 4 * (a2 * Ek - b2 * Kk) / (a2 - b2);
      return std::make_pair(fx, dfx);
   }
private:
   T m_arc;     // length of arc.
   T m_radius;  // one of the two radii of the ellipse
};  // struct elliptic_root__functor_1deriv
```

The root-finding code is now almost the same as before, but we'll make use of Newton-iteration to get the result:

```
template <class T = double>
T elliptic_root_1deriv(T radius, T arc)
{
   using namespace std;  // Help ADL of std functions.
   using namespace boost::math::tools; // For newton_raphson_iterate.

   BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");

   T guess = sqrt(arc * arc / 16 - radius * radius);
   T min = 0;   // Minimum possible value is zero.
   T max = arc; // Maximum possible value is the arc length.

   // Accuracy doubles at each step, so stop when just over half of the digits are
   // correct, and rely on that step to polish off the remainder:
   int get_digits = static_cast<int>(std::numeric_limits<T>::digits * 0.6);
   const boost::uintmax_t maxit = 20;
   boost::uintmax_t it = maxit;
   T result = newton_raphson_iterate(elliptic_root_functor_1deriv<T>(arc, radi↵
us), guess, min, max, get_digits, it);
   return result;
} // T elliptic_root_1_deriv  Newton-Raphson
```

The number of iterations required for `double` precision is now usually around 4 - so we've slightly more than halved the number of iterations, but made the functor twice as expensive to call!

Interestingly though, the second derivative requires no more expensive elliptic integral calls than the first does, in other words it comes essentially "for free", in which case we might as well make use of it and use Halley-iteration. This is quite a typical situation when inverting special-functions. Here's the revised functor:

```
template <class T = double>
struct elliptic_root_functor_2deriv
{ // Functor returning both 1st and 2nd derivatives.
   BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");

   elliptic_root_functor_2deriv(T const& arc, T const& radius) : m_arc(arc), m_radius(radius) {}
   std::tuple<T, T, T> operator()(T const& x)
   {
      using std::sqrt;
     // Return the difference between required arc-length, and the calculated arc-length for an
      // ellipse with radii m_radius and x, plus it's derivative.
      // See http://www.wolframalpha.com/input/?i=d^2%2Fda^2+[4+*+a+*+EllipticE%281+-
+b^2%2Fa^2%29]
      // for the second derivative.
      T a = (std::max)(m_radius, x);
      T b = (std::min)(m_radius, x);
      T a2 = a * a;
      T b2 = b * b;
      T k = sqrt(1 - b2 / a2);
      T Ek = boost::math::ellint_2(k);
      T Kk = boost::math::ellint_1(k);
      T fx = 4 * a * Ek - m_arc;
      T dfx = 4 * (a2 * Ek - b2 * Kk) / (a2 - b2);
      T dfx2 = 4 * b2 * ((a2 + b2) * Kk - 2 * a2 * Ek) / (a * (a2 - b2) * (a2 - b2));
      return std::make_tuple(fx, dfx, dfx2);
   }
private:
   T m_arc;     // length of arc.
   T m_radius;  // one of the two radii of the ellipse
};
```

The actual root-finding code is almost the same as before, except we can use Halley, rather than Newton iteration:

```
template <class T = double>
T elliptic_root_2deriv(T radius, T arc)
{
   using namespace std;                // Help ADL of std functions.
   using namespace boost::math::tools; // For halley_iterate.

   BOOST_STATIC_ASSERT_MSG(boost::is_integral<T>::value == false, "Only floating-point type ↵
types can be used!");

   T guess = sqrt(arc * arc / 16 - radius * radius);
   T min = 0;                                    // Minimum possible value is zero.
   T max = arc;                                 // radius can't be larger than the arc length.

   // Accuracy triples at each step, so stop when just over one-third of the digits
   // are correct, and the last iteration will polish off the remaining digits:
   int get_digits = static_cast<int>(std::numeric_limits<T>::digits * 0.4);
   const boost::uintmax_t maxit = 20;
   boost::uintmax_t it = maxit;
   T result = halley_iterate(elliptic_root_functor_2deriv<T>(arc, radi↵
us), guess, min, max, get_digits, it);
   return result;
} // nth_2deriv Halley
```

While this function uses only slightly fewer iterations (typically around 3) to find the root, compared to the original derivative-free method, we've moved from 8-10 elliptic integral calls to 6.

Full code of this example is at root_elliptic_finding.cpp.

The Effect of a Poor Initial Guess

It's instructive to take our "toy" example algorithms, and use deliberately bad initial guesses to see how the various root finding algorithms fair. We'll start with the cubed root, and using the cube root of 500 as the test case:

Initial Guess=	**-500% (≈1.323)**	**-100% (≈3.97)**	**-50% (≈3.96)**	**-20% (≈6.35)**	**-10% (≈7.14)**	**-5% (≈7.54)**	**5% (≈8.33)**	**10% (≈8.73)**	**20% (≈9.52)**	**50% (≈11.91)**	**100% (≈15.87)**	**500 (≈47.6)**
bracket_and_solve_root	12	8	8	10	11	11	11	11	11	11	7	13
newton_iterate	12	7	7	5	5	4	4	5	5	6	7	9
halley_iterate	7	4	4	3	3	3	3	3	3	4	4	6
schroder_iterate	11	6	6	4	3	3	3	3	4	5	5	8

As you can see `bracket_and_solve_root` is relatively insensitive to starting location - as long as you don't start many orders of magnitude away from the root it will take roughly the same number of steps to bracket the root and solve it. On the other hand the derivative-based methods are slow to start, but once they have some digits correct they increase precision exceptionally fast: they are therefore quite sensitive to the initial starting location.

The next table shows the number of iterations required to find the second radius of an ellipse with first radius 50 and arc-length 500:

Initial Guess=	**-500% (≈20.6)**	**-100% (≈61.81)**	**-50% (≈61.81)**	**-20% (≈98.9)**	**-10% (≈111.3)**	**-5% (≈117.4)**	**5% (≈129.8)**	**10% (≈136)**	**20% (≈148.3)**	**50% (≈185.4)**	**100% (≈247.2)**	**500 (≈741.7)**
bracket_and_solve_root	11	5	5	8	8	7	7	8	9	8	6	10
newton_iterate	4	4	4	3	3	3	3	3	3	4	4	4
halley_iterate	4	3	3	3	3	2	2	3	3	3	3	3
schroder_iterate	4	3	3	3	3	2	2	3	3	3	3	3

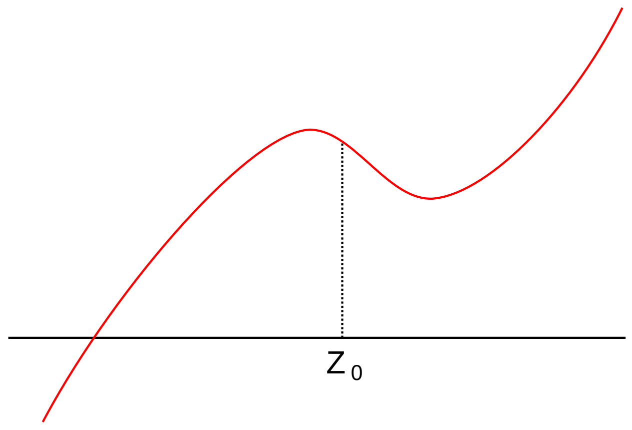
Interestingly this function is much more resistant to a poor initial guess when using derivatives.

Examples Where Root Finding Goes Wrong

There are many reasons why root root finding can fail, here are just a few of the more common examples:

Local Minima

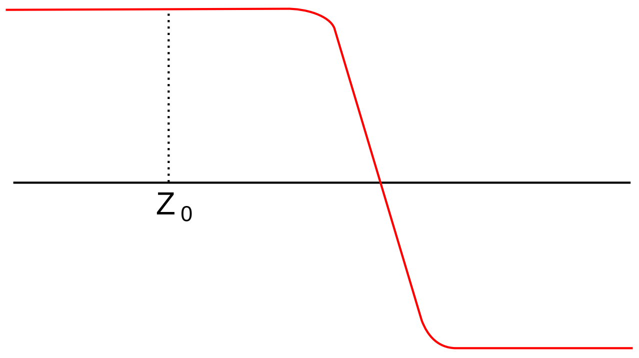
If you start in the wrong place, such as z_0 here:



Then almost any root-finding algorithm will descend into a local minima rather than find the root.

Flatlining

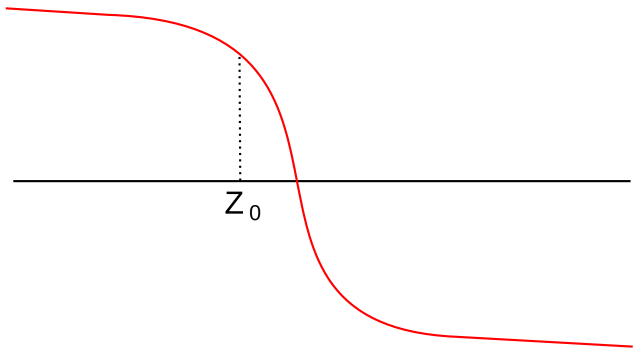
In this example, we're starting from a location (z_0) where the first derivative is essentially zero:



In this situation the next iteration will shoot off to infinity (assuming we're using derivatives that is). Our code guards against this by insisting that the root is always bracketed, and then never stepping outside those bounds. In a case like this, no root finding algorithm can do better than bisecting until the root is found.

Note that there is no scale on the graph, we have seen examples of this situation occur in practice *even when several decimal places of the initial guess z_0 are correct.*

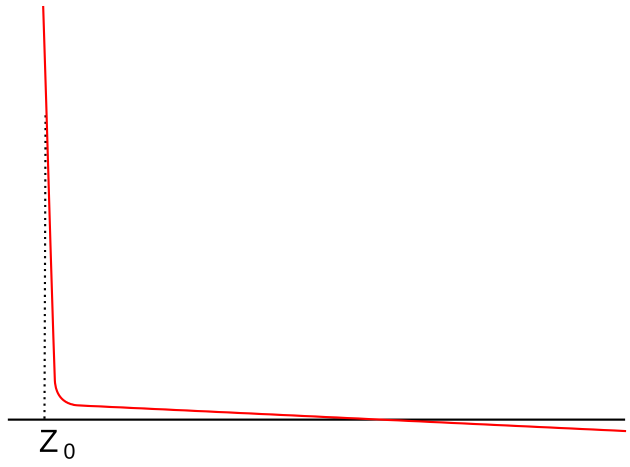
This is really a special case of a more common situation where root finding with derivatives is *divergent*. Consider starting at z_0 in this case:



An initial Newton step would take you further from the root than you started, as will all subsequent steps.

Micro-stepping / Non-convergence

Consider starting at z_0 in this situation:



The first derivative is essentially infinite, and the second close to zero (and so offers no correction if we use it), as a result we take a very small first step. In the worst case situation, the first step is so small - perhaps even so small that subtracting from z_0 has no effect at the current working precision - that our algorithm will assume we are at the root already and terminate. Otherwise we will take lot's of very small steps which never converge on the root: our algorithms will protect against that by reverting to bisection.

An example of this situation would be trying to find the root of e^{-1/z^2} - this function has a single root at $z = 0$, but for $z_0 < 0$ neither Newton nor Halley steps will ever converge on the root, and for $z_0 > 0$ the steps are actually divergent.

Locating Function Minima using Brent's algorithm

Synopsis

```
#include <boost/math/tools/minima.hpp>
```

```
template <class F, class T>
std::pair<T, T> brent_find_minima(F f, T min, T max, int bits);

template <class F, class T>
std::pair<T, T> brent_find_minima(F f, T min, T max, int bits, boost::uintmax_t& max_iter);
```

Description

These two functions locate the minima of the continuous function *f* using Brent's method: specifically it uses quadratic interpolation to locate the minima, or if that fails, falls back to a golden-section search.

Parameters

f	The function to minimise: a function object (or C++ lambda) that should be smooth over the range *[min, max]*, with no maxima occurring in that interval.
min	The lower endpoint of the range in which to search for the minima.
max	The upper endpoint of the range in which to search for the minima.
bits	The number of bits precision to which the minima should be found. Note that in principle, the minima can not be located to greater accuracy than the square root of machine epsilon (for 64-bit double, sqrt(1e-16)≅1e-8), therefore the value of *bits* will be ignored if it's greater than half the number of bits in the mantissa of T.
max_iter	The maximum number of iterations to use in the algorithm, if not provided the algorithm will just keep on going until the minima is found.

Returns:

A `std::pair` of type T containing the value of the abscissa (x) at the minima and the value of *f(x)* at the minima.

Tip

Defining BOOST_MATH_INSTRUMENT will show some parameters, for example:

```
Type T is double
bits = 24, maximum 26
tolerance = 1.19209289550781e-007
seeking minimum in range min-4 to 1.33333333333333
maximum iterations 18446744073709551615
10 iterations.
```

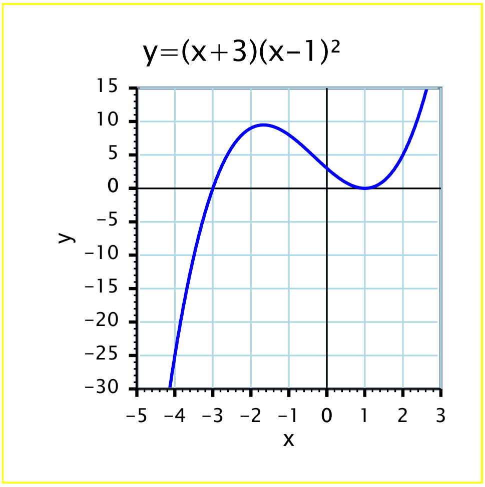
Brent Minimisation Example

As a demonstration, we replicate this Wikipedia example minimising the function *y= (x+3)(x-1)*^{2}.

It is obvious from the equation and the plot that there is a minimum at exactly unity (x = 1) and the value of the function at one is exactly zero (y = 0).

Tip

This observation shows that an analytical or Closed-form expression solution always beats brute-force hands-down for both speed and precision.



First an include is needed:

```
#include <boost/math/tools/minima.hpp>
```

This function is encoded in C++ as function object (functor) using `double` precision thus:

```
struct funcdouble
{
  double operator()(double const& x)
  {
    return (x + 3) * (x - 1) * (x - 1); // (x + 3)(x - 1)^2
  }
};
```

The Brent function is conveniently accessed through a `using` statement (noting sub-namespace `::tools`).

```
using boost::math::tools::brent_find_minima;
```

The search minimum and maximum are chosen as -4 to 4/3 (as in the Wikipedia example).

Tip

S A Stage (reference 6) reports that the Brent algorithm is *slow to start, but fast to converge*, so choosing a tight min-max range is good.

For simplicity, we set the precision parameter `bits` to `std::numeric_limits<double>::digits`, which is effectively the maximum possible `std::numeric_limits<double>::digits/2`.

Nor do we provide a value for maximum iterations parameter `max_iter`, (probably unwisely), so the function will iterate until it finds a minimum.

```
const int double_bits = std::numeric_limits<double>::digits;
std::pair<double, double> r = brent_find_minima(funcdouble(), -4., 4. / 3, double_bits);

std::streamsize precision_1 = std::cout.precision(std::numeric_limits<double>::digits10);
// Show all double precision decimal digits and trailing zeros.
std::cout << "x at minimum = " << r.first
  << ", f(" << r.first << ") = " << r.second << std::endl;
```

The resulting std::pair contains the minimum close to one, and the minimum value close to zero.

```
x at minimum = 1.00000000112345,
f(1.00000000112345) = 5.04852568272458e-018
```

The differences from the expected *one* and *zero* are less than the uncertainty, for `double` 1.5e-008, calculated from `sqrt(std::numeric_limits<double>::epsilon())`.

We can use this uncertainty to check that the two values are close-enough to those expected,

```
using boost::math::fpc::close_at_tolerance;
using boost::math::fpc::is_small;

std::cout << "x = " << r.first << ", f(x) = " << r.second << std::endl;
std::cout << std::boolalpha << "x == 1 (compared to uncertainty "
  << uncertainty << ") is " << is_close(1., r.first, uncertainty) << std::endl; // true
std::cout << std::boolalpha << "f(x) == 0 (compared to uncertainty "
  << uncertainty << ") is " << is_close(0., r.second, uncertainty) << std::endl; // true
```

that outputs

```
x == 1 (compared to uncertainty 1.49011611938477e-08) is true
f(x) == 0 (compared to uncertainty 1.49011611938477e-08) is true
```

Note

The function `close_at_tolerance` is ineffective for testing if a value is small or zero (because it may divide by small or zero and cause overflow). Function `is_small` does this job.

It is possible to make this comparison more generally with a templated function, `is_close`, first checking `is_small` before checking `close_at_tolerance`, returning `true` when small or close, for example:

```
//! Compare if value got is close to expected,
//! checking first if expected is very small
//! (to avoid divide by tiny or zero during comparison)
//! before comparing expect with value got.

template <class T>
bool is_close(T expect, T got, T tolerance)
{
  using boost::math::fpc::close_at_tolerance;
  using boost::math::fpc::is_small;
  using boost::math::fpc::FPC_STRONG;

  if (is_small<T>(expect, tolerance))
  {
    return is_small<T>(got, tolerance);
  }

  return close_at_tolerance<T>(tolerance, FPC_STRONG) (expect, got);
} // bool is_close(T expect, T got, T tolerance)
```

In practical applications, we might want to know how many iterations, and maybe to limit iterations (in case the function proves ill-behaved), and/or perhaps to trade some loss of precision for speed, for example:

```
const boost::uintmax_t maxit = 20;
boost::uintmax_t it = maxit;
std::pair<double, double> r = brent_find_minima(funcdouble(), -4., 4. / 3, bits, it);
std::cout << "x at minimum = " << r.first << ", f(" << r.first << ") = " << r.second
  << " after " << it << " iterations. " << std::endl;
```

limits to a maximum of 20 iterations (a reasonable estimate for this example function, even for much higher precision shown later).

The parameter `it` is updated to return the actual number of iterations (so it may be useful to also keep a record of the limit set in `const maxit`).

It is neat to avoid showing insignificant digits by computing the number of decimal digits to display.

```
std::streamsize prec = static_cast<int>(2 + sqrt((double)bits));  // Number of significant ↲
decimal digits.
std::streamsize precision_3 = std::cout.precision(prec); // Save and set new precision.
std::cout << "Showing " << bits << " bits "
  "precision with " << prec
  << " decimal digits from tolerance " << sqrt(std::numeric_limits<double>::epsilon())
  << std::endl;

std::cout << "x at minimum = " << r.first
  << ", f(" << r.first << ") = " << r.second
  << " after " << it << " iterations. " << std::endl;
std::cout.precision(precision_3); // Restore.
```

```
Showing 53 bits precision with 9 decimal digits from tolerance 1.49011611938477e-008
x at minimum = 1, f(1) = 5.04852568e-018
```

We can also half the number of precision bits from 52 to 26:

```
const int bits_div_2 = std::numeric_limits<double>::digits / 2; // Half digits precision (ef↵
fective maximum).
double epsilon_2 = boost::math::pow<-(std::numeric_limits<double>::digits/2 - 1), double>(2);
std::streamsize prec = static_cast<int>(2 + sqrt((double)bits_div_2));  // Number of signific↵
ant decimal digits.

std::cout << "Showing " << bits_div_2 << " bits precision with " << prec
  << " decimal digits from tolerance " << sqrt(epsilon_2)
  << std::endl;
std::streamsize precision_4 = std::cout.precision(prec); // Save.
const boost::uintmax_t maxit = 20;
boost::uintmax_t it_4 = maxit;
std::pair<double, double> r = brent_find_minima(funcdouble(), -4., 4. / 3, bits_div_2, it_4);
std::cout << "x at minimum = " << r.first << ", f(" << r.first << ") = ↵
" << r.second << std::endl;
std::cout << it_4 << " iterations. " << std::endl;
std::cout.precision(precision_4); // Restore.
```

showing no change in the result and no change in the number of iterations, as expected.

It is only if we reduce the precision to a quarter, specifying only 13 precision bits

```
const int bits_div_4 = std::numeric_limits<double>::digits / 4; // Quarter precision.
double epsilon_4 = boost::math::pow<-(std::numeric_limits<double>::digits / 4 - 1), double>(2);
std::streamsize prec = static_cast<int>(2 + sqrt((double)bits_div_4));  // Number of signific↵
ant decimal digits.
std::cout << "Showing " << bits_div_4 << " bits precision with " << prec
  << " decimal digits from tolerance " << sqrt(epsilon_4)
  << std::endl;
std::streamsize precision_5 = std::cout.precision(prec); // Save & set.
const boost::uintmax_t maxit = 20;

boost::uintmax_t it_5 = maxit;
std::pair<double, double> r = brent_find_minima(funcdouble(), -4., 4. / 3, bits_div_4, it_5);
std::cout << "x at minimum = " << r.first << ", f(" << r.first << ") = " << r.second
<< ", after " << it_5 << " iterations. " << std::endl;
std::cout.precision(precision_5); // Restore.
```

that we reduce the number of iterations from 10 to 7 that the result slightly differs from *one* and *zero*.

```
Showing 13 bits precision with 9 decimal digits from tolerance 0.015625
x at minimum = 0.9999776, f(0.9999776) = 2.0069572e-009 after 7 iterations.
```

Templating on floating-point type

If we want to switch the floating-point type, then the functor must be revised. Since the functor is stateless, the easiest option is to simply make `operator()` a template member function:

```
struct func
{
  template <class T>
  T operator()(T const& x)
  {
    return (x + 3) * (x - 1) * (x - 1); // (x + 3)(x - 1)^2
  }
};
```

The `brent_find_minima` function can now be used in template form, for example using `long double`:

```
std::streamsize precision_t1 = std::cout.precision(std::numeric_limits<long double>::di↵
gits10); // Save & set.
long double bracket_min = -4.;
long double bracket_max = 4. / 3;
const int bits = std::numeric_limits<long double>::digits;
const boost::uintmax_t maxit = 20;
boost::uintmax_t it = maxit;

std::pair<long double, long double> r = brent_find_minima(func(), bracket_min, brack↵
et_max, bits, it);
std::cout << "x at minimum = " << r.first << ", f(" << r.first << ") = " << r.second
  << ", after " << it << " iterations. " << std::endl;
std::cout.precision(precision_t1);  // Restore.
```

The form shown uses the floating-point type `long double` by deduction, but it is also possible to be more explicit, for example:

```
std::pair<long double, long double> r = brent_find_minima<func, long double>
(func(), bracket_min, bracket_max, bits, it);
```

In order to show the use of multiprecision below (or other user-defined types), it may be convenient to write a templated function to use this:

```
//! Example template function to find and show minima.
//! \tparam T floating-point or fixed_point type.
template <class T>
void show_minima()
{
  using boost::math::tools::brent_find_minima;
  using std::sqrt;
  try
  { // Always use try'n'catch blocks with Boost.Math to ensure you get any error messages.

    int bits = std::numeric_limits<T>::digits/2; // Maximum is digits/2;
    std::streamsize prec = static_cast<int>(2 + sqrt((double)bits));  // Number of signific↵
ant decimal digits.
    std::streamsize precision = std::cout.precision(prec); // Save and set.

    std::cout << "\n\nFor type: " << typeid(T).name()
      << ",\n  epsilon = " << std::numeric_limits<T>::epsilon()
      // << ", precision of " << bits << " bits"
      << ",\n  the maximum theoretical precision from Brent's minimization is "
      << sqrt(std::numeric_limits<T>::epsilon())
     << "\n  Displaying to std::numeric_limits<T>::digits10 " << prec << ", significant decim↵
al digits."
      << std::endl;

    const boost::uintmax_t maxit = 20;
    boost::uintmax_t it = maxit;
    // Construct using string, not double, avoids loss of precision.
    //T bracket_min = static_cast<T>("-4");
    //T bracket_max = static_cast<T>("1.3333333333333333333333333333333333333333333333333");

    // Construction from double may cause loss of precision for multiprecision types like ↵
cpp_bin_float,
    // but brackets values are good enough for using Brent minimization.
    T bracket_min = static_cast<T>(-4);
    T bracket_max = static_cast<T>(1.3333333333333333333333333333333333333333333333333);

   std::pair<T, T> r = brent_find_minima<func, T>(func(), bracket_min, bracket_max, bits, it);

    std::cout << "  x at minimum = " << r.first << ", f(" << r.first << ") = " << r.second;
    if (it < maxit)
    {
      std::cout << ",\n  met " << bits << " bits precision" << ", after " << it << " itera↵
tions." << std::endl;
    }
    else
    {
     std::cout << ",\n  did NOT meet " << bits << " bits precision" << " after " << it << " it↵
erations!" << std::endl;
    }
    // Check that result is that expected (compared to theoretical uncertainty).
    T uncertainty = sqrt(std::numeric_limits<T>::epsilon());
   std::cout << std::boolalpha << "x == 1 (compared to uncertainty " << uncertainty << ") is "
      << is_close(static_cast<T>(1), r.first, uncertainty) << std::endl;
   std::cout << std::boolalpha << "f(x) == (0 compared to uncertainty " << uncertainty << ") is ↵
"
      << is_close(static_cast<T>(0), r.second, uncertainty) << std::endl;
    // Problems with this using multiprecision with expression template on?
    std::cout.precision(precision);  // Restore.
  }
  catch (const std::exception& e)
  { // Always useful to include try & catch blocks because default policies
    // are to throw exceptions on arguments that cause errors like underflow, overflow.
    // Lacking try & catch blocks, the program will abort without a message below,
    // which may give some helpful clues as to the cause of the exception.
    std::cout <<
      "\n""Message from thrown exception was:\n   " << e.what() << std::endl;
  }
} // void show_minima()
```

Note

the prudent addition of `try'n'catch` blocks within the function. This ensures that any malfunction will provide a Boost.Math error message rather than uncommunicatively calling `std::abort`.

We can use this with all built-in floating-point types, for example

```
show_minima<float>();
show_minima<double>();
show_minima<long double>();
```

Quad 128-bit precision

On platforms that provide it, a 128-bit quad type can be used. (See float128).

If this type is available, the build should define the macro BOOST_HAVE_QUADMATH:

```
#ifdef BOOST_HAVE_QUADMATH  // Defined only if GCC or Intel and have quadmath.lib or .dll lib↵
rary available.
  using boost::multiprecision::float128;
#endif
```

and can be (conditionally) used this:

```
#ifdef BOOST_HAVE_QUADMATH  // Defined only if GCC or Intel and have quadmath.lib or .dll lib↵
rary available.
  show_minima<float128>(); // Needs quadmath_snprintf, sqrtQ, fabsq that are in in quadmath ↵
library.
#endif
```

Multiprecision

If a higher precision than `double` (or `long double` if that is more precise) is required, then this is easily achieved using Boost.Multiprecision with some includes, for example:

```
#include <boost/multiprecision/cpp_dec_float.hpp> // For decimal boost::multipreci↵
sion::cpp_dec_float_50.
#include <boost/multiprecision/cpp_bin_float.hpp> // For binary boost::multipreci↵
sion::cpp_bin_float_50;
```

and some `typdefs`.

```
using boost::multiprecision::cpp_bin_float_50; // binary multiprecision typedef.
using boost::multiprecision::cpp_dec_float_50; // decimal multiprecision typedef.

// One might also need typedefs like these to switch expression templates off and on (default ↵
is on).
typedef boost::multiprecision::number<boost::multiprecision::cpp_bin_float<50>,
  boost::multiprecision::et_on>
  cpp_bin_float_50_et_on;  // et_on is default so is same as cpp_bin_float_50.

typedef boost::multiprecision::number<boost::multiprecision::cpp_bin_float<50>,
  boost::multiprecision::et_off>
  cpp_bin_float_50_et_off;

typedef boost::multiprecision::number<boost::multiprecision::cpp_dec_float<50>,
  boost::multiprecision::et_on> // et_on is default so is same as cpp_dec_float_50.
  cpp_dec_float_50_et_on;

typedef boost::multiprecision::number<boost::multiprecision::cpp_dec_float<50>,
  boost::multiprecision::et_off>
  cpp_dec_float_50_et_off;
```

Used thus

```
std::cout.precision(std::numeric_limits<cpp_bin_float_50>::digits10);
int bits = std::numeric_limits<cpp_bin_float_50>::digits / 2 - 2;
cpp_bin_float_50 bracket_min = static_cast<cpp_bin_float_50>("-4");
cpp_bin_float_50 bracket_max = stat↵
ic_cast<cpp_bin_float_50>("1.3333333333333333333333333333333333333333333333333");

std::cout << "Bracketing " << bracket_min << " to " << bracket_max << std::endl;
const boost::uintmax_t maxit = 20;
boost::uintmax_t it = maxit; // Will be updated with actual iteration count.
std::pair<cpp_bin_float_50, cpp_bin_float_50> r
  = brent_find_minima(func(), bracket_min, bracket_max, bits, it);

std::cout << "x at minimum = " << r.first << ",\n f(" << r.first << ") = " << r.second
// x at minimum = 1, f(1) = 5.04853e-018
  << ", after " << it << " iterations. " << std::endl;

is_close_to(static_cast<cpp_bin_float_50>("1"), r.first, sqrt(std::numeric_lim↵
its<cpp_bin_float_50>::epsilon()));
is_close_to(static_cast<cpp_bin_float_50>("0"), r.second, sqrt(std::numeric_lim↵
its<cpp_bin_float_50>::epsilon()));
```

and with our `show_minima` function

```
show_minima<cpp_bin_float_50_et_on>(); //
```

```
For type  class boost::multiprecision::number<class boost::multipreci↵
sion::backends::cpp_bin_float<50,10,void,int,0,0>,1>,
epsilon = 5.3455294202e-51,
the maximum theoretical precision from Brent minimization is 7.311312755e-26
Displaying to std::numeric_limits<T>::digits10 11 significant decimal digits.
x at minimum = 1, f(1) = 5.6273022713e-58,
met 84 bits precision, after 14 iterations.
x == 1 (compared to uncertainty 7.311312755e-26) is true
f(x) == (0 compared to uncertainty 7.311312755e-26) is true
-4 1.3333333333333333333333333333333333333333333333333
x at minimum = 0.99999999999999999999999999998813903221565569205253,
f(0.99999999999999999999999999998813903221565569205253) =
  5.6273022712501408640665300316078046703496236636624e-58
14 iterations
```

```
For type  class boost::multiprecision::number<class boost::multipreci↵
sion::backends::cpp_bin_float<50, 10, void, int, 0, 0>, 1>,
```

Tip

One can usually rely on template argument deduction to avoid specifying the verbose multiprecision types, but great care in needed with the *type of the values* provided to avoid confusing the compiler.

Tip

Using `std::cout.precision(std::numeric_limits<T>::digits10);` or `std::cout.precision(std::numeric_limits<T>::max_digits10);` during debugging may be wise because it gives some warning if construction of multiprecision values involves unintended conversion from `double` by showing trailing zero or random digits after max_digits10, that is 17 for `double`, digit 18... may be just noise.

The complete example code is at brent_minimise_example.cpp.

Implementation

This is a reasonably faithful implementation of Brent's algorithm.

References

1. Brent, R.P. 1973, Algorithms for Minimization without Derivatives, (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5.

2. Numerical Recipes in C, The Art of Scientific Computing, Second Edition, William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Cambridge University Press. 1988, 1992.

3. An algorithm with guaranteed convergence for finding a zero of a function, R. P. Brent, The Computer Journal, Vol 44, 1971.

4. Brent's method in Wikipedia.

5. Z. Zhang, An Improvement to the Brent's Method, IJEA, vol. 2, pp. 2 to 26, May 31, 2011. http://www.cscjournals.org/manuscript/Journals/IJEA/volume2/Issue1/IJEA-7.pdf

6. Steven A. Stage, Comments on An Improvement to the Brent's Method (and comparison of various algorithms) http://www.cscjournals.org/manuscript/Journals/IJEA/volume4/Issue1/IJEA-33.pdf Stage concludes that Brent's algorithm is slow to start, but fast to finish convergence, and has good accuracy.

Comparison of Root Finding Algorithms

Comparison of Cube Root Finding Algorithms

In the table below, the cube root of 28 was computed for three fundamental (built-in) types floating-point types, and one Boost.Multiprecision type cpp_bin_float using 50 decimal digit precision, using four algorithms.

The 'exact' answer was computed using a 100 decimal digit type:

```
cpp_bin_float_100 full_an↵
swer ("3.036588971875662519420809578505669635581453977248111123242141654169177268411884961770250390838097895");
```

Times were measured using Boost.Timer using `class cpu_timer`.

- *Its* is the number of iterations taken to find the root.
- *Times* is the CPU time-taken in arbitrary units.
- *Norm* is a normalized time, in comparison to the quickest algorithm (with value 1.00).
- *Dis* is the distance from the nearest representation of the 'exact' root in bits. Distance from the 'exact' answer is measured by using function Boost.Math float_distance. One or two bits distance means that all results are effectively 'correct'. Zero means 'exact' - the nearest representable value for the floating-point type.

The cube-root function is a simple function, and is a contrived example for root-finding. It does allow us to investigate some of the factors controlling efficiency that may be extrapolated to more complex functions.

The program used was root_finding_algorithms.cpp. 100000 evaluations of each floating-point type and algorithm were used and the CPU times were judged from repeat runs to have an uncertainty of 10 %. Comparing MSVC for `double` and `long double` (which are identical on this patform) may give a guide to uncertainty of timing.

The requested precision was set as follows:

Function	Precision Requested
TOMS748	numeric_limits<T>::digits - 2
Newton	floor(numeric_limits<T>::digits * 0.6)
Halley	floor(numeric_limits<T>::digits * 0.4)
Schröder	floor(numeric_limits<T>::digits * 0.4)

- The C++ Standard cube root function std::cbrt is only defined for built-in or fundamental types, so cannot be used with any User-Defined floating-point types like Boost.Multiprecision. This, and that the cube function is so impeccably-behaved, allows the implementer to use many tricks to achieve a fast computation. On some platforms,`std::cbrt` appeared several times as quick as the more general `boost::math::cbrt`, on other platforms / compiler options `boost::math::cbrt` is noticeably faster. In general, the results are highly dependent on the code-generation / processor architecture selection compiler options used. One can assume that the standard library will have been compiled with options *nearly* optimal for the platform it was installed on, where as the user has more choice over the options used for Boost.Math. Pick something too general/conservative and performance suffers, while selecting options that make use of the latest instruction set opcodes speed's things up noticeably.
- Two compilers in optimise mode were compared: GCC 4.9.1 using Netbeans IDS and Microsoft Visual Studio 2013 (Update 1) on the same hardware. The number of iterations seemed consistent, but the relative run-times surprisingly different.

- `boost::math::cbrt` allows use with *any user-defined floating-point type*, conveniently Boost.Multiprecision. It too can take some advantage of the good-behaviour of the cube function, compared to the more general implementation in the nth root-finding examples. For example, it uses a polynomial approximation to generate a better guess than dividing the exponent by three, and can avoid the complex checks in Newton-Raphson iteration required to prevent the search going wildly off-track. For a known precision, it may also be possible to fix the number of iterations, allowing inlining and loop unrolling. It also algebraically simplifies the Halley steps leading to a big reduction in the number of floating point operations required compared to a "black box" implementation that calculates the derivatives seperately and then combines them in the Halley code. Typically, it was found that computation using type `double` took a few times longer when using the various root-finding algorithms directly rather than the hand coded/optimized `cbrt` routine.

- The importance of getting a good guess can be seen by the iteration count for the multiprecision case: here we "cheat" a little and use the cube-root calculated to double precision as the initial guess. The limitation of this tactic is that the range of possible (exponent) values may be less than the multiprecision type.

- For fundamental (built-in) types, there was little to choose between the three derivative methods, but for cpp_bin_float, Newton-Raphson iteration was twice as fast. Note that the cube-root is an extreme test case as the cost of calling the functor is so cheap that the runtimes are largely dominated by the complexity of the iteration code.

- Compiling with optimisation halved computation times, and any differences between algorithms became nearly negligible. The optimisation speed-up of the TOMS Algorithm 748: enclosing zeros of continuous functions was especially noticable.

- Using a multiprecision type like `cpp_bin_float_50` for a precision of 50 decimal digits took a lot longer, as expected because most computation uses software rather than 64-bit floating-point hardware. Speeds are often more than 50 times slower.

- Using `cpp_bin_float_50`, TOMS Algorithm 748: enclosing zeros of continuous functions was much slower showing the benefit of using derivatives. Newton-Raphson iteration was found to be twice as quick as either of the second-derivative methods: this is an extreme case though, the function and its derivatives are so cheap to compute that we're really measuring the complexity of the boilerplate root-finding code.

- For multiprecision types only one or two extra *iterations* are needed to get the remaining 35 digits, whatever the algorithm used. (The time taken was of course much greater for these types).

- Using a 100 decimal-digit type only doubled the time and required only a very few more iterations, so the cost of extra precision is mainly the underlying cost of computing more digits, not in the way the algorithm works. This confirms previous observations using NTL A Library for doing Number Theory high-precision types.

Program root_finding_algorithms.cpp, Microsoft Visual C++ version 14.1, Dinkumware standard library version 650, Win32, x86

1000000 evaluations of each of 5 root_finding algorithms.

Table 110. Cube root(28) for float, double, long double and cpp_bin_float_50

	float					double					long d					cpp50				
Al-gorithm	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
cbrt	0	[illegible]	1.0	0		0	[illegible]	1.0	1		0	[illegible]	1.0	1		0	[illegible]	1.1	0	
TOMS748	8	[illegible]	6.0	-1		11	[illegible]	15.	2		11	[illegible]	9.7	2		6	[illegible]	7.6	-2	
Newton	5	[illegible]	2.6	0		6	[illegible]	3.8	0		6	[illegible]	2.0	0		2	[illegible]	1.0	0	
Halley	3	[illegible]	3.0	0		4	[illegible]	4.3	0		4	[illegible]	2.5	0		2	[illegible]	2.5	0	
Schröder	4	[illegible]	3.8	0		5	[illegible]	4.5	0		5	[illegible]	2.5	0		2	[illegible]	3.0	0	

Program root_finding_algorithms.cpp, GNU C++ version 8.2.0, GNU libstdc++ version 20180728, linux, x64
1000000 evaluations of each of 5 root_finding algorithms.

Table 111. Cube root(28) for float, double, long double and cpp_bin_float_50

	float					double					long d					cpp50				
Al-gorithm	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
cbrt	0	[illegible]	1.0	0		0	[illegible]	1.0	0		0	[illegible]	1.0	0		0	[illegible]	1.0	0	
TOMS748	8	[illegible]	7.3	-1		11	[illegible]	6.2	2		10	[illegible]	8.3	-1		6	[illegible]	6.7	-2	
Newton	5	[illegible]	4.0	0		6	[illegible]	2.2	0		6	[illegible]	2.6	0		2	[illegible]	1.0	-1	
Halley	3	[illegible]	3.7	0		4	[illegible]	2.3	0		4	[illegible]	3.3	0		2	[illegible]	2.2	0	
Schröder	4	[illegible]	4.0	0		5	[illegible]	2.3	0		5	[illegible]	4.0	0		2	[illegible]	2.7	0	

Comparison of Nth-root Finding Algorithms

A second example compares four generalized nth-root finding algorithms for various n-th roots (5, 7 and 13) of a single value 28.0, for four floating-point types, `float`, `double`, `long double` and a Boost.Multiprecision type `cpp_bin_float_50`. In each case the target accuracy was set using our "recomended" accuracy limits (or at least limits that make a good starting point - which is likely to give close to full accuracy without resorting to unnecessary iterations).

Function	**Precision Requested**
TOMS748	numeric_limits<T>::digits - 2
Newton	floor(numeric_limits<T>::digits * 0.6)
Halley	floor(numeric_limits<T>::digits * 0.4)
Schröder	floor(numeric_limits<T>::digits * 0.4)

Tests used Microsoft Visual Studio 2013 (Update 1) and GCC 4.9.1 using source code root_n_finding_algorithms.cpp.

The timing uncertainty (especially using MSVC) is at least 5% of normalized time 'Norm'.

To pick out the 'best' and 'worst' algorithms are highlighted in blue and red. More than one result can be 'best' when normalized times are indistinguishable within the uncertainty.

Program ..\example\root_n_finding_algorithms.cpp, Microsoft Visual C++ version 14.1, Dinkumware standard library version 650, Win32 Compiled in optimise mode., _X86_SSE2

Fraction of full accuracy 1

Table 112. 5th root(28) for float, double, long double and cpp_bin_float_50 types, using _X86_SSE2

	float					**double**					**long d**					**cpp50**				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	7	457	2.00	0		11	860	3.54	1		11	806	3.02	1		12	23865	8.11	0	
Newton	3	228	1.00	0		4	243	1.00	-1		4	298	1.12	-1		6	2968	1.00	0	
Halley	2	250	1.10	0		3	268	1.10	0		3	267	1.00	0		4	5812	1.89	0	
Schröder	2	256	1.12	0		3	271	1.12	-1		3	270	1.01	-1		4	6406	2.20	0	

Table 113. 7th root(28) for float, double, long double and cpp_bin_float_50 types, using _X86_SSE2

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	825	3.06	1		15	1145	4.06	2		15	1159	4.17	2		14	29581	8.12	0	
Newton	5	270	1.00	0		6	282	1.00	0		6	278	1.00	0		8	3646	1.00	0	
Halley	4	303	1.12	0		5	329	1.17	0		5	335	1.21	0		6	7831	2.15	0	
Schröder	5	340	1.26	0		6	432	1.53	0		6	367	1.32	0		7	8516	2.34	0	

Table 114. 11th root(28) for float, double, long double and cpp_bin_float_50 types, using _X86_SSE2

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	714	3.16	-2		14	909	4.19	2		14	793	3.69	2		17	21178	9.28	2	
Newton	6	226	1.00	0		7	217	1.00	0		7	215	1.00	0		9	2282	1.00	0	
Halley	4	262	1.16	-1		5	260	1.20	0		5	260	1.21	0		6	4081	1.79	0	
Schröder	6	332	1.47	0		7	314	1.45	0		7	310	1.44	0		8	6737	2.95	0	

Program root_n_finding_algorithms.cpp, Microsoft Visual C++ version 12.0, Dinkumware standard library version 610, Win32 Compiled in optimise mode., _X64_AVX

Fraction of full accuracy 1

Table 115. 5th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_AVX

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	7	239	1.50	0		11	451	2.53	1		11	439	2.49	1		12	9032	7.51	0	
Newton	3	159	1.00	0		4	178	1.00	-1		4	176	1.00	-1		6	1231	1.00	0	
Halley	2	168	1.06	0		3	203	1.14	0		3	198	1.13	0		4	2937	1.74	0	
Schröder	2	173	1.09	0		3	206	1.16	-1		3	203	1.15	-1		4	2650	2.18	0	

Table 116. 7th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_AVX

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	385	2.19	1		15	635	3.13	2		15	621	3.17	2		14	1483	6.81	0	
Newton	5	176	1.00	0		6	203	1.00	0		6	196	1.00	0		8	1675	1.00	0	
Halley	4	209	1.19	0		5	254	1.25	0		5	246	1.26	0		6	3243	1.92	0	
Schröder	5	223	1.27	0		6	273	1.34	0		6	275	1.40	0		7	4556	2.68	0	

Table 117. 11th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_AVX

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	467	2.42	-2		14	648	3.06	2		14	640	2.99	2		17	1000	8.85	2	
Newton	6	193	1.00	0		7	212	1.00	0		7	214	1.00	0		9	1928	1.00	0	
Halley	4	209	1.08	-1		5	256	1.21	0		5	250	1.17	0		6	3256	1.70	0	
Schröder	6	248	1.28	0		7	306	1.44	0		7	298	1.39	0		8	5347	2.78	0	

Program ..\example\root_n_finding_algorithms.cpp, GNU C++ version 7.1.0, GNU libstdc++ version 20170502, Win32 Compiled in optimise mode., _X64_SSE2

Fraction of full accuracy 1

Table 118. 5th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_SSE2

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	7	206	2.24	0		11	460	4.04	1		9	554	4.40	0		12	5766	8.39	0	
Newton	3	92	1.00	0		4	114	1.00	-1		5	126	1.00	0		6	6875	1.00	0	
Halley	2	106	1.15	0		3	134	1.18	0		3	178	1.41	0		4	1250	1.82	0	
Schröder	2	126	1.37	0		3	143	1.25	-1		3	198	1.57	0		4	1532	2.23	0	

Table 119. 7th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_SSE2

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	345	2.09	1		15	615	3.14	2		13	875	3.98	0		14	7093	7.32	0	
Newton	5	165	1.00	0		6	196	1.00	0		7	220	1.00	0		8	9687	1.00	0	
Halley	4	193	1.17	0		5	239	1.22	0		5	298	1.35	0		6	1906	1.97	0	
Schröder	5	217	1.32	0		6	270	1.38	0		6	367	1.67	0		7	2343	2.82	0	

Table 120. 11th root(28) for float, double, long double and cpp_bin_float_50 types, using _X64_SSE2

	float					**double**					**long d**					**cpp50**				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	12	412	2.15	-2		14	646	2.96	2		14	1054	4.22	1		17	10787	9.53	2	
Newton	6	192	1.00	0		7	218	1.00	0		7	250	1.00	0		9	11260	1.00	0	
Halley	4	200	1.04	-1		5	243	1.11	0		5	345	1.38	0		6	19687	1.75	0	
Schröder	6	254	1.32	0		7	321	1.47	0		7	471	1.88	0		8	33281	2.96	0	

Some tentative conclusions can be drawn from this limited exercise.

- Perhaps surprisingly, there is little difference between the various algorithms for fundamental (built-in) types floating-point types. Using the first derivatives (Newton-Raphson iteration) is usually the best, but while the improvement over the no-derivative TOMS Algorithm 748: enclosing zeros of continuous functions is considerable in number of iterations, but little in execution time. This reflects the fact that the function we are finding the root for is trivial to evaluate, so runtimetimes are dominated by the time taken by the boilerplate code in each method.
- The extra cost of evaluating the second derivatives (Halley or Schröder) is usually too much for any net benefit: as with the cube root, these functors are so cheap to evaluate that the runtime is largely dominated by the complexity of the root finding method.
- For a Boost.Multiprecision floating-point type, the Newton-Raphson iteration is a clear winner with a several-fold gain over TOMS Algorithm 748: enclosing zeros of continuous functions, and again no improvement from the second-derivative algorithms.
- The run-time of 50 decimal-digit Boost.Multiprecision is about 30-fold greater than `double`.
- The column 'dis' showing the number of bits distance from the correct result. The Newton-Raphson algorithm shows a bit or two better accuracy than TOMS Algorithm 748: enclosing zeros of continuous functions.
- The goodness of the 'guess' is especially crucial for Boost.Multiprecision. Separate experiments show that evaluating the 'guess' using `double` allows convergence to the final exact result in one or two iterations. So in this contrived example, crudely dividing the exponent by N for a 'guess', it would be far better to use a `pow<double>` or , if more precise `pow<long double>`, function to estimate a 'guess'. The limitation of this tactic is that the range of possible (exponent) values may be less than the multiprecision type.
- Using floating-point extension SSE2 instructions made a modest ten-percent speedup.
- Using MSVC, there was some improvement using 64-bit, markedly for Boost.Multiprecision.
- The GCC compiler 4.9.1 using 64-bit was at least five-folder faster that 32-bit, apparently reflecting better optimization.

Clearly, your mileage **will vary**, but in summary, Newton-Raphson iteration seems the first choice of algorithm, and effort to find a good 'guess' the first speed-up target, especially for Boost.Multiprecision. And of course, compiler optimisation is crucial for speed.

Comparison of Elliptic Integral Root Finding Algoritghms

A second example compares four root finding algorithms for locating the second radius of an ellipse with first radius 28 and arc length 300, for four floating-point types, `float`, `double`, `long double` and a Boost.Multiprecision type `cpp_bin_float_50`.

Which is to say we're solving:

```
4xE(sqrt(1 - 28² / x²)) - 300 = 0
```

In each case the target accuracy was set using our "recomended" accuracy limits (or at least limits that make a good starting point - which is likely to give close to full accuracy without resorting to unnecessary iterations).

Function	Precision Requested
TOMS748	numeric_limits<T>::digits - 2
Newton	floor(numeric_limits<T>::digits * 0.6)
Halley	floor(numeric_limits<T>::digits * 0.4)
Schröder	floor(numeric_limits<T>::digits * 0.4)

Tests used Microsoft Visual Studio 2013 (Update 1) and GCC 4.9.1 using source code root_elliptic_finding.cpp.

The timing uncertainty (especially using MSVC) is at least 5% of normalized time 'Norm'.

To pick out the 'best' and 'worst' algorithms are highlighted in blue and red. More than one result can be 'best' when normalized times are indistinguishable within the uncertainty.

Program root_elliptic_finding.cpp, Microsoft Visual C++ version 14.1, Dinkumware standard library version 650,Win32 Compiled in optimise mode., _X86_SSE2

Table 121. root with radius 28 and arc length 300) for float, double, long double and cpp_bin_float_50 types, using _X86_SSE2

	float					**double**					**long d**					**cpp50**				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	6	906	2.07	0		9	1312	1.79	1		9	1281	1.75	1		11	[illegible]	1.52	-3	
Newton	3	640	1.46	-1		4	875	1.19	1		4	843	1.15	1		5	[illegible]	1.23	0	
Halley	2	437	1.00	0		3	734	1.00	3		3	734	1.00	3		4	[illegible]	1.00	0	
Schröder	3	671	1.54	-1		6	1296	1.77	1		6	1406	1.92	1		5	[illegible]	1.32	-2	

Program root_elliptic_finding.cpp, Microsoft Visual C++ version 12.0, Dinkumware standard library version 610, Win32 Compiled in optimise mode., _X64_AVX

Table 122. root with radius 28 and arc length 300) for float, double, long double and cpp_bin_float_50 types, using _X64_AVX

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	5	500	1.33	-1		9	1046	1.72	1		9	1062	1.70	1		11	6937	1.54	-3	
Newton	3	484	1.29	-1		4	734	1.21	1		4	687	1.10	1		5	5462	1.20	0	
Halley	2	375	1.00	0		3	609	1.00	3		3	625	1.00	3		4	4525	1.00	0	
Schroder	3	546	1.46	-1		6	1109	1.82	1		6	1187	1.90	1		5	5602	1.24	-2	

Program root_elliptic_finding.cpp, GNU C++ version 7.1.0, GNU libstdc++ version 20170502, Win32 Compiled in optimise mode., _X64_SSE2

Table 123. root with radius 28 and arc length 300) for float, double, long double and cpp_bin_float_50 types, using _X64_SSE2

	float					double					long d					cpp50				
Algo	Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis		Its	Times	Norm	Dis	
TOMS748	5	328	1.24	-1		8	890	1.50	0		8	1234	1.61	4		11	4360	1.57	-3	
Newton	3	359	1.35	-1		4	718	1.21	1		4	843	1.10	1		5	3687	1.22	0	
Halley	2	265	1.00	0		3	593	1.00	1		3	765	1.00	7		4	3197	1.00	0	
Schroder	3	343	1.29	-1		4	812	1.37	0		4	1046	1.37	3		5	3925	1.26	-2	

Remarks:

- The function being solved is now moderately expensive to call, and twice as expensive to call when obtaining the derivative than when not. Consequently there is very little improvement in moving from a derivative free method, to Newton iteration. However, once you've calculated the first derivative the second comes almost for free, consequently the third order methods (Halley) does much the best.
- Of the two second order methods, Halley does best as would be expected: the Schroder method offers better guarantees of *quadratic* convergence, while Halley relies on a smooth function with a single root to give *cubic* convergence. It's not entirely clear why Schroder iteration often does worse than Newton.

Polynomials and Rational Functions

Polynomials

Synopsis

```
#include <boost/math/tools/polynomial.hpp>
```

```
namespace boost { namespace math {
namespace tools {

template <class T>
class polynomial
{
public:
   // typedefs:
   typedef typename std::vector<T>::value_type value_type;
   typedef typename std::vector<T>::size_type  size_type;

   // construct:
   polynomial(){}
   template <class U>
   polynomial(const U* data, unsigned order);
   template <class I>
   polynomial(I first, I last);
   template <class U>
   explicit polynomial(const U& point,
         typename boost::enable_if<boost::is_convertible<U, T> >::type* = 0);
   template <class Range>
   explicit polynomial(const Range& r,
         typename boost::enable_if<boost::math::tools::detail::is_const_iter↵
able<Range> >::type* = 0); // C++14
   polynomial(std::initializer_list<T> l); // C++11

   polynomial(std::vector<T>&& p);

   // access:
   size_type size()const;
   size_type degree()const;
   value_type& operator[](size_type i);
   const value_type& operator[](size_type i)const;
   std::vector<T> const& data() const;
   std::vector<T>& data();
   explicit operator bool() const;

   polynomial<T> prime() const;

   polynomial<T> integrate() const;

   T operator()(T z) const;



   // modify:
   void set_zero();

   // operators:
   template <class U>
   polynomial& operator +=(const U& value);
   template <class U>
   polynomial& operator -=(const U& value);
   template <class U>
   polynomial& operator *=(const U& value);
   template <class U>
   polynomial& operator /=(const U& value);
   template <class U>
   polynomial& operator %=(const U& value);
   template <class U>
   polynomial& operator +=(const polynomial<U>& value);
   template <class U>
   polynomial& operator -=(const polynomial<U>& value);
   template <class U>
   polynomial& operator *=(const polynomial<U>& value);
   template <class U>
   polynomial& operator /=(const polynomial<U>& value);
   template <class U>
   polynomial& operator %=(const polynomial<U>& value);
};

template <class T>
polynomial<T> operator + (const polynomial<T>& a, const polynomial<T>& b);
template <class T>
polynomial<T> operator - (const polynomial<T>& a, const polynomial<T>& b);
template <class T>
polynomial<T> operator * (const polynomial<T>& a, const polynomial<T>& b);
template <class T>
polynomial<T> operator / (const polynomial<T>& a, const polynomial<T>& b);
template <class T>
polynomial<T> operator % (const polynomial<T>& a, const polynomial<T>& b);

template <class T, class U>
polynomial<T> operator + (const polynomial<T>& a, const U& b);
template <class T, class U>
polynomial<T> operator - (const polynomial<T>& a, const U& b);
template <class T, class U>
polynomial<T> operator * (const polynomial<T>& a, const U& b);
template <class T, class U>
polynomial<T> operator / (const polynomial<T>& a, const U& b);
template <class T, class U>
polynomial<T> operator % (const polynomial<T>& a, const U& b);

template <class U, class T>
polynomial<T> operator + (const U& a, const polynomial<T>& b);
template <class U, class T>
polynomial<T> operator - (const U& a, const polynomial<T>& b);
template <class U, class T>
polynomial<T> operator * (const U& a, const polynomial<T>& b);

template <class T>
polynomial<T> operator - (const polynomial<T>& a);

template <class T>
polynomial<T> operator >>= (const U& a);
template <class T>
polynomial<T> operator >> (polynomial<T> const &a, const U& b);
template <class T>
polynomial<T> operator <<= (const U& a);
template <class T>
polynomial<T> operator << (polynomial<T> const &a, const U& b);

template <class T>
bool operator == (const polynomial<T> &a, const polynomial<T> &b);
template <class T>
bool operator != (const polynomial<T> &a, const polynomial<T> &b);

template <class T>
```

```
polynomial<T> pow(polynomial<T> base, int exp);

template <class charT, class traits, class T>
std::basic_ostream<charT, traits>& operator <<
   (std::basic_ostream<charT, traits>& os, const polynomial<T>& poly);

template <typename T>
std::pair< polynomial<T>, polynomial<T> >
quotient_remainder(const polynomial<T>& a, const polynomial<T>& b);

} //    namespace tools
}} //    namespace boost { namespace math
```

Description

This is a somewhat trivial class for polynomial manipulation.

See

- Polynomial and
- Donald E. Knuth, The Art of Computer Programming: Volume 2, Third edition, (1998) Chapter 4.6.1, Algorithm D: Division of polynomials over a field.

Implementation is currently of the "naive" variety, with (N^2) multiplication, for example. This class should not be used in high-performance computing environments: it is intended for the simple manipulation of small polynomials, typically generated for special function approximation.

It does has division for polynomials over a field (here floating point, complex, etc) and over a unique factorization domain (integers). Division of polynomials over a field is compatible with Euclidean GCD.

Division of polynomials over a UFD is compatible with the subresultant algorithm for GCD (implemented as subresultant_gcd), but a serious word of warning is required: the intermediate value swell of that algorithm will cause single-precision integral types to overflow very easily. So although the algorithm will work on single-precision integral types, an overload of the gcd function is only provided for polynomials with multi-precision integral types, to prevent nasty surprises. This is done somewhat crudely by disabling the overload for non-POD integral types.

Advanced manipulations: the FFT, factorisation etc are not currently provided. Submissions for these are of course welcome :-)

Polynomial Arithmetic Examples

First include the essential polynomial header (and others) to make the example:

```
#include <boost/math/tools/polynomial.hpp>
```

and some using statements are convenient:

```
using std::string;
using std::exception;
using std::cout;
using std::abs;
using std::pair;

using namespace boost::math;
using namespace boost::math::tools; // for polynomial
using boost::lexical_cast;
```

Store the coefficients in a convenient way to access them, then create some polynomials using construction from an iterator range, and finally output in a 'pretty' formula format.

Tip

Although we might conventionally write a polynomial from left to right in descending order of degree, Boost.Math stores in **ascending order of degree**.

```
Read/write for humans:    3x^3 - 4x^2 - 6x + 10
Boost polynomial storage: [ 10, -6, -4, 3 ]
```

```
boost::array<double, 4> const d3a = {{10, -6, -4, 3}};
polynomial<double> const a(d3a.begin(), d3a.end());

// With C++11 and later, you can also use initializer_list construction.
polynomial<double> const b{{-2.0, 1.0}};

// formula_format() converts from Boost storage to human notation.
cout << "a = " << formula_format(a)
<< "\nb = " << formula_format(b) << "\n\n";
```

for output:

```
a = 3x^3 - 4x^2 - 6x + 10
b = x - 2
```

```
// Now we can do arithmetic with the usual infix operators: + - * / and %.
polynomial<double> s = a + b;
cout << "a + b = " << formula_format(s) << "\n";
polynomial<double> d = a - b;
cout << "a - b = " << formula_format(d) << "\n";
polynomial<double> p = a * b;
cout << "a * b = " << formula_format(p) << "\n";
polynomial<double> q = a / b;
cout << "a / b = " << formula_format(q) << "\n";
polynomial<double> r = a % b;
cout << "a % b = " << formula_format(r) << "\n";
```

for output

```
a + b = 3x^3 - 4x^2 - 5x + 8
a - b = 3x^3 - 4x^2 - 7x + 12
a * b = 3x^4 - 10x^3 + 2x^2 + 22x - 20
a / b = 3x^2 + 2x - 2
a % b = 6
```

Division, Quotient and Remainder

Division is a special case where you can calculate two for the price of one.

Actually, quotient and remainder are always calculated together due to the nature of the algorithm: the infix operators return one result and throw the other away.

If you are doing a lot of division and want both the quotient and remainder, then you don't want to do twice the work necessary.

In that case you can call the underlying function, `quotient_remainder`, to get both results together as a pair.

```
  pair< polynomial<double>, polynomial<double> > result;
  result = quotient_remainder(a, b);
// Reassure ourselves that the result is the same.
  BOOST_ASSERT(result.first == q);
  BOOST_ASSERT(result.second == r);
```

The source code is at polynomial_arithmetic.cpp

Polynomial and Rational Function Evaluation

Synopsis

```
#include <boost/math/tools/rational.hpp>
```

```
// Polynomials:
template <std::size_t N, class T, class V>
V evaluate_polynomial(const T(&poly)[N], const V& val);

template <std::size_t N, class T, class V>
V evaluate_polynomial(const boost::array<T,N>& poly, const V& val);

template <class T, class U>
U evaluate_polynomial(const T* poly, U z, std::size_t count);

// Even polynomials:
template <std::size_t N, class T, class V>
V evaluate_even_polynomial(const T(&poly)[N], const V& z);

template <std::size_t N, class T, class V>
V evaluate_even_polynomial(const boost::array<T,N>& poly, const V& z);

template <class T, class U>
U evaluate_even_polynomial(const T* poly, U z, std::size_t count);

// Odd polynomials
template <std::size_t N, class T, class V>
V evaluate_odd_polynomial(const T(&a)[N], const V& z);

template <std::size_t N, class T, class V>
V evaluate_odd_polynomial(const boost::array<T,N>& a, const V& z);

template <class T, class U>
U evaluate_odd_polynomial(const T* poly, U z, std::size_t count);

// Rational Functions:
template <std::size_t N, class T, class V>
V evaluate_rational(const T(&a)[N], const T(&b)[N], const V& z);

template <std::size_t N, class T, class V>
V evaluate_rational(const boost::array<T,N>& a, const boost::array<T,N>& b, const V& z);

template <class T, class U, class V>
V evaluate_rational(const T* num, const U* denom, V z, unsigned count);
```

Description

Each of the functions come in three variants: a pair of overloaded functions where the order of the polynomial or rational function is evaluated at compile time, and an overload that accepts a runtime variable for the size of the coefficient array. Generally speaking, compile time evaluation of the array size results in better type safety, is less prone to programmer errors, and may result in better optimised code. The polynomial evaluation functions in particular, are specialised for various array sizes, allowing for loop unrolling, and one hopes, optimal inline expansion.

```
template <std::size_t N, class T, class V>
V evaluate_polynomial(const T(&poly)[N], const V& val);

template <std::size_t N, class T, class V>
V evaluate_polynomial(const boost::array<T,N>& poly, const V& val);

template <class T, class U>
U evaluate_polynomial(const T* poly, U z, std::size_t count);
```

Evaluates the polynomial described by the coefficients stored in *poly*.

If the size of the array is specified at runtime, then the polynomial most have order *count-1* with *count* coefficients. Otherwise it has order *N-1* with *N* coefficients.

Coefficients should be stored such that the coefficients for the x^i terms are in poly[i].

The types of the coefficients and of variable *z* may differ as long as **poly* is convertible to type *U*. This allows, for example, for the coefficient table to be a table of integers if this is appropriate.

```
template <std::size_t N, class T, class V>
V evaluate_even_polynomial(const T(&poly)[N], const V& z);

template <std::size_t N, class T, class V>
V evaluate_even_polynomial(const boost::array<T,N>& poly, const V& z);

template <class T, class U>
U evaluate_even_polynomial(const T* poly, U z, std::size_t count);
```

As above, but evaluates an even polynomial: one where all the powers of *z* are even numbers. Equivalent to calling `evaluate_polynomial(poly, z*z, count)`.

```
template <std::size_t N, class T, class V>
V evaluate_odd_polynomial(const T(&a)[N], const V& z);

template <std::size_t N, class T, class V>
V evaluate_odd_polynomial(const boost::array<T,N>& a, const V& z);

template <class T, class U>
U evaluate_odd_polynomial(const T* poly, U z, std::size_t count);
```

As above but evaluates a polynomial where all the powers are odd numbers. Equivalent to `evaluate_polynomial(poly+1, z*z, count-1) * z + poly[0]`.

```
template <std::size_t N, class T, class U, class V>
V evaluate_rational(const T(&num)[N], const U(&denom)[N], const V& z);

template <std::size_t N, class T, class U, class V>
V evaluate_rational(const boost::array<T,N>& num, const boost::array<U,N>& denom, const V& z);

template <class T, class U, class V>
V evaluate_rational(const T* num, const U* denom, V z, unsigned count);
```

Evaluates the rational function (the ratio of two polynomials) described by the coefficients stored in *num* and *demom*.

If the size of the array is specified at runtime then both polynomials most have order *count-1* with *count* coefficients. Otherwise both polynomials have order *N-1* with *N* coefficients.

Array *num* describes the numerator, and *demon* the denominator.

Coefficients should be stored such that the coefficients for the x^i terms are in num[i] and denom[i].

The types of the coefficients and of variable *v* may differ as long as **num* and **denom* are convertible to type *V*. This allows, for example, for one or both of the coefficient tables to be a table of integers if this is appropriate.

These functions are designed to safely evaluate the result, even when the value *z* is very large. As such they do not take advantage of compile time array sizes to make any optimisations. These functions are best reserved for situations where *z* may be large: if you can be sure that numerical overflow will not occur then polynomial evaluation with compile-time array sizes may offer slightly better performance.

Implementation

Polynomials are evaluated by Horners method. If the array size is known at compile time then the functions dispatch to size-specific implementations that unroll the evaluation loop.

Rational evaluation is by Horners method: with the two polynomials being evaluated in parallel to make the most of the processors floating-point pipeline. If *v* is greater than one, then the polynomials are evaluated in reverse order as polynomials in *1/v*: this avoids unnecessary numerical overflow when the coefficients are large.

Both the polynomial and rational function evaluation algorithms can be tuned using various configuration macros to provide optimal performance for a particular combination of compiler and platform. This includes support for second-order Horner's methods. The various options are documented here. However, the performance benefits to be gained from these are marginal on most current hardware, consequently it's best to run the performance test application before changing the default settings.

Interpolation

Cubic B-spline interpolation

Synopsis

```
#include <boost/math/interpolators/cubic_b_spline.hpp>
```

```
namespace boost { namespace math {

  template <class Real>
  class cubic_b_spline
  {
  public:

    template <class BidiIterator>
      cubic_b_spline(BidiIterator a, BidiIterator b, Real left_endpoint, Real step_size,
                     Real left_endpoint_derivative = std::numeric_limits<Real>::quiet_NaN(),
                    Real right_endpoint_derivative = std::numeric_limits<Real>::quiet_NaN());
      cubic_b_spline(const Real* const f, size_t length, Real left_endpoint, Real step_size,
                     Real left_endpoint_derivative = std::numeric_limits<Real>::quiet_NaN(),
                    Real right_endpoint_derivative = std::numeric_limits<Real>::quiet_NaN());

      Real operator()(Real x) const;

      Real prime(Real x) const;
  };

}} // namespaces
```

Cubic B-Spline Interpolation

The cubic B-spline class provided by boost allows fast and accurate interpolation of a function which is known at equally spaced points. The cubic B-spline interpolation is numerically stable as it uses compactly supported basis functions constructed via iterative convolution. This is to be contrasted to traditional cubic spline interpolation which is ill-conditioned as the global support of cubic polynomials causes small changes far from the evaluation point to exert a large influence on the calculated value.

There are many use cases for interpolating a function at equally spaced points. One particularly important example is solving ODEs whose coefficients depend on data determined from experiment or numerical simulation. Since most ODE steppers are adaptive, they must be able to sample the coefficients at arbitrary points; not just at the points we know the values of our function.

The first two arguments to the constructor are either:

- A pair of bidirectional iterators into the data, or
- A pointer to the data, and a length of the data array.

These are then followed by:

- The start of the functions domain
- The step size

Optionally, you may provide two additional arguments to the constructor, namely the derivative of the function at the left endpoint, and the derivative at the right endpoint. If you do not provide these arguments, they will be estimated using one-sided finite-difference formulas. An example of a valid call to the constructor is

```
std::vector<double> f{0.01, -0.02, 0.3, 0.8, 1.9, -8.78, -22.6};
double t0 = 0;
double h = 0.01;
boost::math::cubic_b_spline<double> spline(f.begin(), f.end(), t0, h);
```

The endpoints are estimated using a one-sided finite-difference formula. If you know the derivative at the endpoint, you may pass it to the constructor via

```
boost::math::cubic_b_spline<double> spline(f.begin(), f.end(), t0, h, a_prime, b_prime);
```

To evaluate the interpolant at a point, we simply use

```
double y = spline(x);
```

and to evaluate the derivative of the interpolant we use

```
double yp = spline.prime(x);
```

Be aware that the accuracy guarantees on the derivative of the spline are an order lower than the guarantees on the original function, see Numerical Analysis, Graduate Texts in Mathematics, 181, Rainer Kress for details.

Finally, note that this is an interpolator, not an extrapolator. Therefore, you should strenuously avoid evaluating the spline outside the endpoints. However, it is not an error if you do, as often you cannot control where (say) an ODE stepper will evaluate your function. As such the interpolant tries to do something reasonable when the passed a value outside the endpoints. For evaluation within one stepsize of the interval, you can assume something somewhat reasonable was returned. As you move further away from the endpoints, the interpolant decays to its average on the interval.

Complexity and Performance

The call to the constructor requires (n) operations, where n is the number of points to interpolate. Each call the the interpolant is (1) (constant time). On the author's Intel Xeon E3-1230, this takes 21ns as long as the vector is small enough to fit in cache.

Accuracy

Let h be the stepsize. If f is four-times continuously differentiable, then the interpolant is (h^4) accurate and the derivative is (h^3) accurate.

Testing

Since the interpolant obeys $s(x_j) = f(x_j)$ at all interpolation points, the tests generate random data and evaluate the interpolant at the interpolation points, validating that equality with the data holds.

In addition, constant, linear, and quadratic functions are interpolated to ensure that the interpolant behaves as expected.

Example

This example demonstrates how to use the cubic b spline interpolator for regularly spaced data.

```
#include <boost/math/interpolators/cubic_b_spline.hpp>

int main()
{
    // We begin with an array of samples:
    std::vector<double> v(500);
    // And decide on a stepsize:
    double step = 0.01;

    // Initialize the vector with a function we'd like to interpolate:
    for (size_t i = 0; i < v.size(); ++i)
    {
        v[i] = sin(i*step);
    }
    // We could define an arbitrary start time, but for now we'll just use 0:
    boost::math::cubic_b_spline<double> spline(v.data(), v.size(), 0 /* start time */, step);

    // Now we can evaluate the spline wherever we please.
    std::mt19937 gen;
    boost::random::uniform_real_distribution<double> absissa(0, v.size()*step);
    for (size_t i = 0; i < 10; ++i)
    {
        double x = absissa(gen);
        std::cout << "sin(" << x << ") = " << sin(x) << ", spline interpolation gives ↵
" << spline(x) << std::endl;
        std::cout << "cos(" << x << ") = " << cos(x) << ", spline derivative interpolation ↵
gives " << spline.prime(x) << std::endl;
    }

    // The next example is less trivial:
    // We will try to figure out when the population of the United States crossed 100 million.
    // Since the census is taken every 10 years, the data is equally spaced, so we can use ↵
the cubic b spline.
    // Data taken from https://en.wikipedia.org/wiki/United_States_Census
    // We'll start at the year 1860:
    double t0 = 1860;
    double time_step = 10;
    std::vector<double> population{31443321,  /* 1860 */
                                   39818449,  /* 1870 */
                                   50189209,  /* 1880 */
                                   62947714,  /* 1890 */
                                   76212168,  /* 1900 */
                                   92228496,  /* 1910 */
                                   106021537, /* 1920 */
                                   122775046, /* 1930 */
                                   132164569, /* 1940 */
                                   150697361, /* 1950 */
                                   179323175};/* 1960 */

    // An eyeball estimate indicates that the population crossed 100 million around 1915.
    // Let's see what interpolation says:
   boost::math::cubic_b_spline<double> p(population.data(), population.size(), t0, time_step);

    // Now create a function which has a zero at p = 100,000,000:
    auto f = [=](double t){ return p(t) - 100000000; };

   // Boost includes a bisection algorithm, which is robust, though not as fast as some others
    // we provide, but let's try that first.  We need a termination condition for it, which
   // takes the two endpoints of the range and returns either true (stop) or false (keep going),
   // we could use a predefined one such as boost::math::tools::eps_tolerance<double>, but that
   // won't stop until we have full double precision which is overkill, since we just need the
    // endpoint to yield the same month.  While we're at it, we'll keep track of the number of
    // iterations required too, though this is strictly optional:
```

```
    auto termination = [](double left, double right)
    {
       double left_month = std::round((left - std::floor(left)) * 12 + 1);
       double right_month = std::round((right - std::floor(right)) * 12 + 1);
       return (left_month == right_month) && (std::floor(left) == std::floor(right));
    };
    std::uintmax_t iterations = 1000;
    auto result =  boost::math::tools::bisect(f, 1910.0, 1920.0, termination, iterations);
   auto time = result.first;  // termination condition ensures that both endpoints yield the ↲
same result
    auto month = std::round((time - std::floor(time))*12  + 1);
    auto year = std::floor(time);
    std::cout << "The population of the United States surpassed 100 million on the ";
    std::cout << month << "th month of " << year << std::endl;
    std::cout << "Found in " << iterations << " iterations" << std::endl;

    // Since the cubic B spline offers the first derivative, we could equally have used New↲
ton iterations,
    // this takes "number of bits correct" as a termination condition - 20 should be plenty ↲
for what we need,
    // and once again, we track how many iterations are taken:

    auto f_n = [=](double t) { return std::make_pair(p(t) - 100000000, p.prime(t)); };
    iterations = 1000;
    time = boost::math::tools::newton_raphson_iterate(f_n, 1910.0, 1900.0, 2000.0, 20, itera↲
tions);
    month = std::round((time - std::floor(time))*12  + 1);
    year = std::floor(time);
    std::cout << "The population of the United States surpassed 100 million on the ";
    std::cout << month << "th month of " << year << std::endl;
    std::cout << "Found in " << iterations << " iterations" << std::endl;

}
```

```
Program output is:
```

```
sin(4.07362) = -0.802829, spline interpolation gives - 0.802829
cos(4.07362) = -0.596209, spline derivative interpolation gives - 0.596209
sin(0.677385) = 0.626758, spline interpolation gives 0.626758
cos(0.677385) = 0.779214, spline derivative interpolation gives 0.779214
sin(4.52896) = -0.983224, spline interpolation gives - 0.983224
cos(4.52896) = -0.182402, spline derivative interpolation gives - 0.182402
sin(4.17504) = -0.85907, spline interpolation gives - 0.85907
cos(4.17504) = -0.511858, spline derivative interpolation gives - 0.511858
sin(0.634934) = 0.593124, spline interpolation gives 0.593124
cos(0.634934) = 0.805111, spline derivative interpolation gives 0.805111
sin(4.84434) = -0.991307, spline interpolation gives - 0.991307
cos(4.84434) = 0.131567, spline derivative interpolation gives 0.131567
sin(4.56688) = -0.989432, spline interpolation gives - 0.989432
cos(4.56688) = -0.144997, spline derivative interpolation gives - 0.144997
sin(1.10517) = 0.893541, spline interpolation gives 0.893541
cos(1.10517) = 0.448982, spline derivative interpolation gives 0.448982
sin(3.1618) = -0.0202022, spline interpolation gives - 0.0202022
cos(3.1618) = -0.999796, spline derivative interpolation gives - 0.999796
sin(1.54084) = 0.999551, spline interpolation gives 0.999551
cos(1.54084) = 0.0299566, spline derivative interpolation gives 0.0299566
The population of the United States surpassed 100 million on the 11th month of 1915
Found in 12 iterations
The population of the United States surpassed 100 million on the 11th month of 1915
Found in 3 iterations
```

Cardinal Quadratic B-spline interpolation

Synopsis

```
#include <boost/math/interpolators/cardinal_quadratic_b_spline.hpp>
```

```
namespace boost{ namespace math{ namespace interpolators {

template <class Real>
class cardinal_quadratic_b_spline
{
public:
    // If you don't know the value of the derivative at the endpoints, leave them as nans and ↵
the routine will estimate them.
    // y[0] = y(a), y[n - 1] = y(b), step_size = (b - a)/(n -1).
    cardinal_quadratic_b_spline(const Real* const y,
                                size_t n,
                                Real t0 /* initial time, left endpoint */,
                                Real h  /*spacing, stepsize*/,
                                Real left_endpoint_derivative = std::numeric_lim↵
its<Real>::quiet_NaN(),
                                Real right_endpoint_derivative = std::numeric_lim↵
its<Real>::quiet_NaN())

    cardinal_quadratic_b_spline(std::vector<Real> const & y,
                                Real t0 /* initial time, left endpoint */,
                                Real h  /*spacing, stepsize*/,
                                Real left_endpoint_derivative = std::numeric_lim↵
its<Real>::quiet_NaN(),
                                Real right_endpoint_derivative = std::numeric_lim↵
its<Real>::quiet_NaN())

    Real operator()(Real t) const;

    Real prime(Real t) const;
};
}}}
```

Cardinal Quadratic B-Spline Interpolation

The cardinal quadratic B-spline interpolator is very nearly the same as the cubic B-spline interpolator, with the modification that the basis functions are constructed by convolving a box function with itself twice, rather than three times as is done with the cubic B-spline.

Since the basis functions are less smooth than the cubic B-spline, you will nearly always wish to use the cubic B-spline interpolator rather than this. However, this interpolator is occasionally useful for approximating functions of reduced smoothness, as hence finds a uses internally in the Boost.Math library.

It is reasonable to test this interpolator against the cubic b-spline interpolator when you are approximating functions which are two or three times continuously differentiable, but not three or four times differentiable.

Whittaker-Shannon interpolation

Synopsis

```
#include <boost/math/interpolators/whittaker_shannon.hpp>
```

```
namespace boost { namespace math { namespace interpolators {

  template <class RandomAccessContainer>
  class whittaker_shannon
  {
  public:

      using Real = RandomAccessContainer::value_type;

      whittaker_shannon(RandomAccessContainer&& v, Real left_endpoint, Real step_size);

      Real operator()(Real x) const;

      Real prime(Real x) const;
  };

}}} // namespaces
```

Whittaker-Shannon Interpolation

The Whittaker-Shannon interpolator takes equispaced data and interpolates between them via a sum of sinc functions. This interpolation is stable and infinitely smooth, but has linear complexity in the data, making it slow relative to compactly-supported b-splines. In addition, we cannot pass an infinite amount of data into the class, and must truncate the (perhaps) infinite sinc series to a finite number of terms. Since the sinc function has slow 1/x decay, the truncation of the series can incur large error. Hence this interpolator works best when operating on samples of compactly supported functions. Here is an example of interpolating a smooth "bump function":

```
auto bump = [](double x) { if (std::abs(x) >= 1) { return 0.0; } re↵
turn std::exp(-1.0/(1.0-x*x)); };

double t0 = -1;
size_t n = 2049;
double h = 2.0/(n-1.0);

std::vector<double> v(n);
for(size_t i = 0; i < n; ++i) {
    double t = t0 + i*h;
    v[i] = bump(t);
}


auto ws = whittaker_shannon(std::move(v), t0, h);

double y = ws(0.3);
```

The derivative of the interpolant can also be evaluated, but the accuracy is not as high:

```
double yp = ws.prime(0.3);
```

Complexity and Performance

The call to the constructor requires (1) operations, simply moving data into the class. Each call the the interpolant is (n), where n is the number of points to interpolate.

Barycentric Rational Interpolation

Synopsis

```
#include <boost/math/interpolators/barycentric_rational.hpp>

namespace boost{ namespace math{
    template<class Real>
    class barycentric_rational
    {
    public:
        template <class InputIterator1, class InputIterator2>
        barycentric_rational(InputIterator1 start_x, InputIterator1 end_x, InputIterat↲
or2 start_y, size_t approximation_order = 3);

        barycentric_rational(std::vector<Real>&& x, std::vector<Real>&& y, size_t order = 3);

       barycentric_rational(const Real* const x, const Real* const y, size_t n, size_t approx↲
imation_order = 3);

        Real operator()(Real x) const;

        Real prime(Real x) const;

        std::vector<Real>&& return_x();

        std::vector<Real>&& return_y();
    };

}}
```

Description

Barycentric rational interpolation is a high-accuracy interpolation method for non-uniformly spaced samples. It requires (N) time for construction, and (N) time for each evaluation. Linear time evaluation is not optimal; for instance the cubic B-spline can be evaluated in constant time. However, using the cubic B-spline requires uniformly spaced samples, which are not always available.

Use of the class requires a vector of independent variables `x[0]`, `x[1]`, ... `x[n-1]` where `x[i+1] > x[i]`, and a vector of dependent variables `y[0]`, `y[1]`, ... , `y[n-1]`. The call is trivial:

```
std::vector<double> x(500);
std::vector<double> y(500);
// populate x, y, then:
boost::math::barycentric_rational<double> interpolant(std::move(x), std::move(y));
```

This implicitly calls the constructor with approximation order 3, and hence the accuracy is (h^4). In general, if you require an approximation order *d*, then the error is (h^{d+1}). A call to the constructor with an explicit approximation order is demonstrated below

```
boost::math::barycentric_rational<double> interpolant(std::move(x), std::move(y), 5);
```

To evaluate the interpolant, simply use

```
double x = 2.3;
double y = interpolant(x);
```

and to evaluate its derivative use

```
double y = interpolant.prime(x);
```

If you no longer require the interpolant, then you can get your data back:

```
std::vector<double> xs = interpolant.return_x();
std::vector<double> ys = interpolant.return_y();
```

Be aware that once you return your data, the interpolant is **dead**.

Caveats

Although this algorithm is robust, it can surprise you. The main way this occurs is if the sample spacing at the endpoints is much larger than the spacing in the center. This is to be expected; all interpolants perform better in the opposite regime, where samples are clustered at the endpoints and somewhat uniformly spaced throughout the center.

A desirable property of any interpolator f is that for all $x_{\min} \le x \le x_{\max}$, $y_{\min} \le f(x) \le y_{\max}$. *This property does not hold for the barycentric rational interpolator.* However, unless you deliberately try to antagonize this interpolator (by, for instance, placing the final value far from all the rest), you will probably not fall victim to this problem.

The reference used for implementation of this algorithm is Barycentric rational interpolation with no poles and a high rate of interpolation, and the evaluation of the derivative is given by Some New Aspects of Rational Interpolation.

Examples

This example shows how to use barycentric rational interpolation, using Walter Kohn's classic paper "Solution of the Schrodinger Equation in Periodic Lattices with an Application to Metallic Lithium" In this paper, Kohn needs to repeatedly solve an ODE (the radial Schrodinger equation) given a potential which is only known at non-equally samples data.

If he'd only had the barycentric rational interpolant of boost::math!

References: Kohn, W., and N. Rostoker. "Solution of the Schrodinger equation in periodic lattices with an application to metallic lithium." Physical Review 94.5 (1954): 1111.

```
#include <boost/math/interpolators/barycentric_rational.hpp>

int main()
{
    // The lithium potential is given in Kohn's paper, Table I:
    std::vector<double> r(45);
    std::vector<double> mrV(45);

    // We'll skip the code for filling the above vectors with data for now...

    // Now we want to interpolate this potential at any r:
    boost::math::barycentric_rational<double> b(r.data(), mrV.data(), r.size());

    for (size_t i = 1; i < 8; ++i)
    {
        double r = i*0.5;
        std::cout <<  "(r, V) = (" << r << ", " << -b(r)/r << ")\n";
    }
}
```

This further example shows how to use the iterator based constructor, and then uses the function object in our root finding algorithms to locate the points where the potential achieves a specific value.

```
#include <boost/math/interpolators/barycentric_rational.hpp>
#include <boost/range/adaptors.hpp>
#include <boost/math/tools/roots.hpp>

int main()
{
    // The lithium potential is given in Kohn's paper, Table I,
    // we could equally use an unordered_map, a list of tuples or pairs,
    // or a 2-dimentional array equally easily:
    std::map<double, double> r;

    r[0.02] = 5.727;
    r[0.04] = 5.544;
    r[0.06] = 5.450;
    r[0.08] = 5.351;
    r[0.10] = 5.253;
    r[0.12] = 5.157;
    r[0.14] = 5.058;
    r[0.16] = 4.960;
    r[0.18] = 4.862;
    r[0.20] = 4.762;
    r[0.24] = 4.563;
    r[0.28] = 4.360;
    r[0.32] = 4.1584;
    r[0.36] = 3.9463;
    r[0.40] = 3.7360;
    r[0.44] = 3.5429;
    r[0.48] = 3.3797;
    r[0.52] = 3.2417;
    r[0.56] = 3.1209;
    r[0.60] = 3.0138;
    r[0.68] = 2.8342;
    r[0.76] = 2.6881;
    r[0.84] = 2.5662;
    r[0.92] = 2.4242;
    r[1.00] = 2.3766;
    r[1.08] = 2.3058;
    r[1.16] = 2.2458;
    r[1.24] = 2.2035;
    r[1.32] = 2.1661;
    r[1.40] = 2.1350;
    r[1.48] = 2.1090;
    r[1.64] = 2.0697;
    r[1.80] = 2.0466;
    r[1.96] = 2.0325;
    r[2.12] = 2.0288;
    r[2.28] = 2.0292;
    r[2.44] = 2.0228;
    r[2.60] = 2.0124;
    r[2.76] = 2.0065;
    r[2.92] = 2.0031;
    r[3.08] = 2.0015;
    r[3.24] = 2.0008;
    r[3.40] = 2.0004;
    r[3.56] = 2.0002;
    r[3.72] = 2.0001;

    // Let's discover the absissa that will generate a potential of exactly 3.0,
    // start by creating 2 ranges for the x and y values:
    auto x_range = boost::adaptors::keys(r);
    auto y_range = boost::adaptors::values(r);
   boost::math::barycentric_rational<double> b(x_range.begin(), x_range.end(), y_range.begin());
    //
    // We'll use a lamda expression to provide the functor to our root finder, since we want
    // the abscissa value that yields 3, not zero.  We pass the functor b by value to the
    // lambda expression since barycentric_rational is trivial to copy.
    // Here we're using simple bisection to find the root:
    boost::uintmax_t iterations = (std::numeric_limits<boost::uintmax_t>::max)();
    double abscissa_3 = boost::math::tools::bisect([=](double x) { re↵
turn b(x) - 3; }, 0.44, 1.24, boost::math::tools::eps_tolerance<double>(), iterations).first;
    std::cout << "Abscissa value that yields a potential of 3 = " << abscissa_3 << std::endl;
    std::cout << "Root was found in " << iterations << " iterations." << std::endl;
    //
    // However, we have a more efficient root finding algorithm than simple bisection:
    iterations = (std::numeric_limits<boost::uintmax_t>::max)();
    abscissa_3 = boost::math::tools::bracket_and_solve_root([=](double x) { re↵
turn b(x) - 3; }, 0.6, 1.2, false, boost::math::tools::eps_tolerance<double>(), itera↵
tions).first;
    std::cout << "Abscissa value that yields a potential of 3 = " << abscissa_3 << std::endl;
    std::cout << "Root was found in " << iterations << " iterations." << std::endl;
}
```

```
Program output is:
```

```
Abscissa value that yields a potential of 3 = 0.604728
Root was found in 54 iterations.
Abscissa value that yields a potential of 3 = 0.604728
Root was found in 10 iterations.
```

Vector-valued Barycentric Rational Interpolation

Synopsis

```
#include <boost/math/interpolators/vector_barycentric_rational.hpp>

namespace boost{ namespace math{

template<class TimeContainer, class SpaceContainer>
class vector_barycentric_rational
{
public:
    using Real = typename TimeContainer::value_type;
    using Point = typename SpaceContainer::value_type;
   vector_barycentric_rational(TimeContainer&& times, SpaceContainer&& points, size_t approx↵
imation_order = 3);

    void operator()(Point& x, Real t) const;

    Point operator()(Real t) const;

    void prime(Point& dxdt, Real t) const;

    Point prime(Real t);

    void eval_with_prime(Point& x, Point& dxdt, Real t) const;

    std::pair<Point, Point> eval_with_prime(Real t) const;
};

}}
```

Description

The *n* dimensional vector-valued barycentric rational interpolator is exactly the same as *n* scalar-valued barycentric rational interpolators. This is provided primarily for convenience and a slight improvement in efficiency over using *n* different rational interpolators and combining their results.

Use of the class requires a `Point`-type which has size known at compile time. These requirements are satisfied by (for example) `Eigen::Vector2d`s and `std::array<Real, N>` classes. The call to the constructor computes the weights:

```
using boost::math::vector_barycentric_rational;
std::vector<double> t(100);
std::vector<Eigen::Vector2d> y(100);
// initialize t and y . . .
vector_barycentric_rational<decltype(t), decltype(y)> interpolant(std::move(t), std::move(y));
```

To evaluate the interpolant, use

```
double t = 2.3;
Eigen::Vector2d y = interpolant(t);
```

If you want to populate a vector passed into the interpolant, rather than get it returned, that syntax is supported:

```
Eigen::Vector2d y;
interpolant(y, t);
```

We tested this with `Eigen::Vectors` and found no performance benefit, but other `Point`-types might not be the same.

To evaluate the derivative of the interpolant use

```
auto [y, y_prime] = interpolant.eval_with_prime(x);
```

Computation of the derivative requires evaluation, so if you can try to use both values at once.

Catmull-Rom Splines

Synopsis

```
#include <boost/math/interpolators/catmull_rom.hpp>

namespace boost{ namespace math{

    template<class Point, class RandomAccessContainer = std::vector<Point> >
    class catmull_rom
    {
    public:

        catmull_rom(RandomAccessContainer&& points, bool closed = false, Real al↵
pha = (Real) 1/ (Real) 2)

        catmull_rom(std::initializer_list<Point> l, bool closed = false, type↵
name Point::value_type alpha = (typename Point::value_type) 1/ (typename Point::value_type) 2);

        Real operator()(Real s) const;

        Real max_parameter() const;

        Real parameter_at_point(size_t i) const;

        Point prime(Real s) const;
    };

}}
```

Description

Catmull-Rom splines are a family of interpolating curves which are commonly used in computer graphics and animation. Catmull-Rom splines enjoy the following properties:

- Affine invariance: The interpolant commutes with affine transformations.
- Local support of the basis functions: This gives stability and fast evaluation.
- C^2-smoothness
- Interpolation of control points-this means the curve passes through the control points. Many curves (such as Bezier) are *approximating*-they do not pass through their control points. This makes them more difficult to use than interpolating splines.

The `catmull_rom` class provided by Boost creates a cubic Catmull-Rom spline from an array of points in any dimension. Since there are numerous ways to represent a point in *n*-dimensional space, the class attempts to be flexible by templating on the point type. The requirements on the point type are discussing in more detail below, but roughly, it must have a dereference operator defined (e.g., `p[0]` is not a syntax error), it must be able to be dereferenced up to `dimension -1`, and `p[i]` is of type `Real`, define `value_type`, and the free function `size()`. These requirements are met by `std::vector` and `std::array`. The basic usage is shown here:

```
std::vector<std::array<Real, 3>> points(4);
points[0] = {0,0,0};
points[1] = {1,0,0};
points[2] = {0,1,0};
points[3] = {0,0,1};
catmull_rom<std::array<Real, 3>> cr(std::move(points));
// Interpolate at s = 0.1:
auto point = cr(0.1);
```

The spline can be either open or *closed*, closed meaning that there is some *s > 0* such that *P(s) = P(0)*. The default is open, but this can be easily changed:

```
// closed = true
catmull_rom<std::array<Real, 3>> cr(std::move(points), true);
```

In either case, evaluating the interpolator at *s=0* returns the first point in the list.

If the curve is open, then the first and last segments may have strange behavior. The traditional solution is to prepend a carefully selected control point to the data so that the first data segment (second interpolator segment) has reasonable tangent vectors, and simply ignore the first interpolator segment. A control point is appended to the data using similar criteria. However, we recommend not going through this effort until it proves to be necessary: For most use-cases, the curve is good enough without prepending and appending control points, and responsible selection of non-data control points is difficult.

Inside `catmull_rom`, the curve is represented as closed. This is because an open Catmull-Rom curve is *implicitly closed*, but the closing point is the zero vector. There is no reason to suppose that the zero vector is a better closing point than the endpoint (or any other point, for that matter), so traditionally Catmull-Rom splines leave the segment between the first and second point undefined, as well as the segment between the second-to-last and last point. We find this property of the traditional implementation of Catmull-Rom splines annoying and confusing to the user. Hence internally, we close the curve so that the first and last segments are defined. Of course, this causes the *tangent* vectors to the first and last points to be bizarre. This is a "pick your poison" design decision- either the curve cannot interpolate in its first and last segments, or the tangents along the first and last segments are meaningless. In the vast majority of cases, this will be no problem to the user. However, if it becomes a problem, then the user should add one extra point in a position they believe is reasonable and close the curve.

Since the routine internally represents the curve as closed, a question arises: Why does the user have to specify if the curve is open or closed? The answer is that the parameterization is chosen by the routine, so it is of interest to the user to understand the values where a meaningful result is returned.

```
Real max_s = cr.max_parameter();
```

If you attempt to interpolate for `s > max_s`, an exception is thrown. If the curve is closed, then `cr(max_s) = p0`, where `p0` is the first point on the curve. If the curve is open, then `cr(max_s) = pf`, where `pf` is the final point on the curve.

The Catmull-Rom curve admits an infinite number of parameterizations. The default parameterization of the `catmull_rom` class is the so-called *centripedal* parameterization. This parameterization has been shown to be the only parameterization that does not form cusps or self-intersections within segments. However, for advanced users, other parameterizations can be chosen using the *alpha* parameter:

```
// alpha = 1 is the "chordal" parameterization.
catmull_rom<std::array<double, 3>> cr(std::move(points), false, 1.0);
```

The alpha parameter must always be in the range `[0,1]`.

Finally, the tangent vector to any point of the curve can be computed via

```
double s = 0.1;
Point tangent = cr.prime(s);
```

Since the magnitude of the tangent vector is dependent on the parameterization, it is not meaningful (unless the user chooses the chordal parameterization *alpha = 1* which parameterizes by Euclidean distance between points.) However, its direction is meaningful no matter the parameterization, so the user may wish to normalize this result.

Examples

Performance

The following performance numbers were generated for a call to the Catmull-Rom interpolation method. The number that follows the slash is the number of points passed to the interpolant. We see that evaluation of the interpolant is (*log*(*N*)).

```
Run on 2700 MHz CPU
CPU Caches:
  L1 Data 32K (x2)
  L1 Instruction 32K (x2)
  L2 Unified 262K (x2)
  L3 Unified 3145K (x1)
---------------------------------------------------------
Benchmark                                  Time           CPU
---------------------------------------------------------
BM_CatmullRom<double>/4                    20 ns         20 ns
BM_CatmullRom<double>/8                    21 ns         21 ns
BM_CatmullRom<double>/16                   23 ns         23 ns
BM_CatmullRom<double>/32                   24 ns         24 ns
BM_CatmullRom<double>/64                   27 ns         27 ns
BM_CatmullRom<double>/128                  27 ns         27 ns
BM_CatmullRom<double>/256                  30 ns         30 ns
BM_CatmullRom<double>/512                  32 ns         31 ns
BM_CatmullRom<double>/1024                 33 ns         33 ns
BM_CatmullRom<double>/2048                 34 ns         34 ns
BM_CatmullRom<double>/4096                 36 ns         36 ns
BM_CatmullRom<double>/8192                 38 ns         38 ns
BM_CatmullRom<double>/16384                39 ns         39 ns
BM_CatmullRom<double>/32768                40 ns         40 ns
BM_CatmullRom<double>/65536                45 ns         44 ns
BM_CatmullRom<double>/131072               46 ns         46 ns
BM_CatmullRom<double>/262144               50 ns         50 ns
BM_CatmullRom<double>/524288               53 ns         52 ns
BM_CatmullRom<double>/1048576              58 ns         57 ns
BM_CatmullRom<double>_BigO               2.97 lgN      2.97 lgN
BM_CatmullRom<double>_RMS                  19 %          19 %
```

Point types

We have already discussed that certain conditions on the `Point` type template argument must be obeyed. The following shows a custom point type in 3D which can be used as a template argument to Catmull-Rom:

```
template<class Real>
class mypoint3d
{
public:
    // Must define a value_type:
    typedef Real value_type;

    // Regular constructor--need not be of this form.
    mypoint3d(Real x, Real y, Real z) {m_vec[0] = x; m_vec[1] = y; m_vec[2] = z; }

    // Must define a default constructor:
    mypoint3d() {}

    // Must define array access:
    Real operator[](size_t i) const
    {
        return m_vec[i];
    }

    // Must define array element assignment:
    Real& operator[](size_t i)
    {
        return m_vec[i];
    }

private:
    std::array<Real, 3> m_vec;
};


// Must define the free function "size()":
template<class Real>
constexpr size_t size(const mypoint3d<Real>& c)
{
    return 3;
}
```

These conditions are satisfied by both `std::array` and `std::vector`, but it may nonetheless be useful to define your own point class so that (say) you can define geometric distance between them.

Caveats

The Catmull-Rom interpolator requires memory for three more points than is provided by the user. This causes the class to call a `resize()` on the input vector. If `v.capacity() >= v.size() + 3`, then no problems arise; there are no reallocs, and in practice this condition is almost always satisfied. However, if `v.capacity() < v.size() + 3`, the realloc causes a performance penalty of roughly 20%.

Generic Containers

The `Point` type may be stored in a different container than `std::vector`. For example, here is how to store the points in a Boost.uBLAS vector:

```
mypoint3d<Real> p0(0.1, 0.2, 0.3);
mypoint3d<Real> p1(0.2, 0.3, 0.4);
mypoint3d<Real> p2(0.3, 0.4, 0.5);
mypoint3d<Real> p3(0.4, 0.5, 0.6);
mypoint3d<Real> p4(0.5, 0.6, 0.7);
mypoint3d<Real> p5(0.6, 0.7, 0.8);

boost::numeric::ublas::vector<mypoint3d<Real>> u(6);
u[0] = p0;
u[1] = p1;
u[2] = p2;
u[3] = p3;
u[4] = p4;
u[5] = p5;

// Tests initializer_list:
catmull_rom<mypoint3d<Real>, decltype(u)> cat(std::move(u));
```

References

- Cem Yuksel, Scott Schaefer, and John Keyser, *Parameterization and applications of Catmull–Rom curves*, Computer-Aided Design 43 (2011) 747–755.
- Phillip J. Barry and Ronald N. Goldman, *A Recursive Evaluation Algorithm for a Class of Catmull-Rom Splines*, Computer Graphics, Volume 22, Number 4, August 1988

Quadrature and Differentiation

Trapezoidal Quadrature

Synopsis

```
#include <boost/math/quadrature/trapezoidal.hpp>
namespace boost{ namespace math{ namespace quadrature {

template<class F, class Real>
auto trapezoidal(F f, Real a, Real b,
                 Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                 size_t max_refinements = 12,
                 Real* error_estimate = nullptr,
                 Real* L1 = nullptr);

template<class F, class Real, class Policy>
auto trapezoidal(F f, Real a, Real b, Real tol, size_t max_refinements,
                 Real* error_estimate, Real* L1, const Policy& pol);

}}} // namespaces
```

Description

The functional `trapezoidal` calculates the integral of a function *f* using the surprisingly simple trapezoidal rule. If we assume only that the integrand is twice continuously differentiable, we can prove that the error of the composite trapezoidal rule is (h^2). Hence halving the interval only cuts the error by about a fourth, which in turn implies that we must evaluate the function many times before an acceptable accuracy can be achieved.

However, the trapezoidal rule has an astonishing property: If the integrand is periodic, and we integrate it over a period, then the trapezoidal rule converges faster than any power of the step size *h*. This can be seen by examination of the Euler-Maclaurin summation formula, which relates a definite integral to its trapezoidal sum and error terms proportional to the derivatives of the function at the endpoints and the Bernoulli numbers. If the derivatives at the endpoints are the same or vanish, then the error very nearly vanishes. Hence the trapezoidal rule is essentially optimal for periodic integrands.

Other classes of integrands which are integrated efficiently by this method are the $C_0^\infty(\propto)$ bump functions and bell-shaped integrals over the infinite interval. For details, see Trefethen's SIAM review.

In its simplest form, an integration can be performed by the following code

```
using boost::math::quadrature::trapezoidal;
auto f = [](double x) { return 1/(5 - 4*cos(x)); };
double I = trapezoidal(f, 0.0, boost::math::constants::two_pi<double>());
```

The integrand must accept a real number argument, but can return a complex number. This is useful for contour integrals (which are manifestly periodic) and high-order numerical differentiation of analytic functions. An example using the integral definition of the complex Bessel function is shown here:

```
auto bessel_integrand = [&n, &z](double theta)->std::complex<double>
{
    std::complex<double> z{2, 3};
    using std::cos;
    using std::sin;
    return cos(z*sin(theta) - 2*theta)/pi<double>();
};

using boost::math::quadrature::trapezoidal;
std::complex<double> Jnz = trapezoidal(bessel_integrand, 0.0, pi<Real>());
```

Other special functions which are efficiently evaluated in the complex plane by trapezoidal quadrature are modified Bessel functions and the complementary error function. Another application of complex-valued trapezoidal quadrature is computation of high-order numerical derivatives; see Lyness and Moler for details.

Since the routine is adaptive, step sizes are halved continuously until a tolerance is reached. In order to control this tolerance, simply call the routine with an additional argument

```
double I = trapezoidal(f, 0.0, two_pi<double>(), 1e-6);
```

The routine stops when successive estimates of the integral `I1` and `I0` differ by less than the tolerance multiplied by the estimated L_1 norm of the function. A good choice for the tolerance is √ε, which is the default. If the integrand is periodic, then the number of correct digits should double on each interval halving. Hence, once the integration routine has estimated that the error is √ε, then the actual error should be ~ε. If the integrand is **not** periodic, then reducing the error to √ε takes much longer, but is nonetheless possible without becoming a major performance bug.

A question arises as to what to do when successive estimates never pass below the tolerance threshold. The stepsize would be halved repeatedly, generating an exponential explosion in function evaluations. As such, you may pass an optional argument `max_refinements` which controls how many times the interval may be halved before giving up. By default, this maximum number of refinement steps is 12, which should never be hit in double precision unless the function is not periodic. However, for higher-precision types, it may be of interest to allow the algorithm to compute more refinements:

```
size_t max_refinements = 15;
long double I = trapezoidal(f, 0.0L, two_pi<long double>(), 1e-9L, max_refinements);
```

Note that the maximum allowed compute time grows exponentially with `max_refinements`. The routine will not throw an exception if the maximum refinements is achieved without the requested tolerance being attained. This is because the value calculated is more often than not still usable. However, for applications with high-reliability requirements, the error estimate should be queried. This is achieved by passing additional pointers into the routine:

```
double error_estimate;
double L1;
double I = trapezoidal(f, 0.0, two_pi<double>(), tolerance, max_refinements, &error_estim↵
ate, &L1);
if (error_estimate > tolerance*L1)
{
     double I = some_other_quadrature_method(f, 0, two_pi<double>());
}
```

The final argument is the L_1 norm of the integral. This is computed along with the integral, and is an essential component of the algorithm. First, the L_1 norm establishes a scale against which the error can be measured. Second, the L_1 norm can be used to evaluate the stability of the computation. This can be formulated in a rigorous manner by defining the **condition number of summation**. The condition number of summation is defined by

$\kappa(S_n) := \Sigma_i^n |x_i|/|\Sigma_i^n x_i|$

If this number of $\sim 10^k$, then *k* additional digits are expected to be lost in addition to digits lost due to floating point rounding error. As all numerical quadrature methods reduce to summation, their stability is controlled by the ratio ∫ |f| dt/|∫ f dt |, which is easily seen to be equivalent to condition number of summation when evaluated numerically. Hence both the error estimate and the condition number of summation should be analyzed in applications requiring very high precision and reliability.

As an example, we consider evaluation of Bessel functions by trapezoidal quadrature. The Bessel function of the first kind is defined via

$J_n(x) = 1/2\Pi \int_{-\Pi}^{\Pi} \cos(n\ t - x \sin(t))\ dt$

The integrand is periodic, so the Euler-Maclaurin summation formula guarantees exponential convergence via the trapezoidal quadrature. Without careful consideration, it seems this would be a very attractive method to compute Bessel functions. However, we see that for large *n* the integrand oscillates rapidly, taking on positive and negative values, and hence the trapezoidal sums become ill-conditioned. In double precision, *x = 17* and *n = 25* gives a sum which is so poorly conditioned that zero correct digits are obtained.

The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

References:

Trefethen, Lloyd N., Weideman, J.A.C., *The Exponentially Convergent Trapezoidal Rule*, SIAM Review, Vol. 56, No. 3, 2014.

Stoer, Josef, and Roland Bulirsch. *Introduction to numerical analysis. Vol. 12.*, Springer Science & Business Media, 2013.

Higham, Nicholas J. *Accuracy and stability of numerical algorithms.* Society for industrial and applied mathematics, 2002.

Lyness, James N., and Cleve B. Moler. *Numerical differentiation of analytic functions.* SIAM Journal on Numerical Analysis 4.2 (1967): 202-210.

Gil, Amparo, Javier Segura, and Nico M. Temme. *Computing special functions by using quadrature rules.* Numerical Algorithms 33.1-4 (2003): 265-275.

Gauss-Legendre quadrature

Synopsis

```
#include <boost/math/quadrature/gauss.hpp>
```

```
namespace boost{ namespace math{ namespace quadrature{

template <class Real, unsigned Points, class Policy = boost::math::policies::policy<> >
struct gauss
{
   static const RandomAccessContainer& abscissa();
   static const RandomAccessContainer& weights();

   template <class F>
   static auto integrate(F f, Real* pL1 = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>()))

   template <class F>
   static auto integrate(F f, Real a, Real b, Real* pL1 = nullptr)->decltype(std::declv↵
al<F>()(std::declval<Real>()))
};

}}} // namespaces
```

description

The `gauss` class template performs "one shot" non-adaptive Gauss-Legendre integration on some arbitrary function *f* using the number of evaluation points as specified by *Points*.

This is intentionally a very simple quadrature routine, it obtains no estimate of the error, and is not adaptive, but is very efficient in simple cases that involve integrating smooth "bell like" functions and functions with rapidly convergent power series.

```
static const RandomAccessContainer& abscissa();
static const RandomAccessContainer& weights();
```

These functions provide direct access to the abscissa and weights used to perform the quadrature: the return type depends on the *Points* template parameter, but is always a RandomAccessContainer type. Note that only positive (or zero) abscissa and weights are stored.

```
template <class F>
static auto integrate(F f, Real* pL1 = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>()))
```

Integrates *f* over (-1,1), and optionally sets `*pL1` to the L1 norm of the returned value: if this is substantially larger than the return value, then the sum was ill-conditioned. Note however, that no error estimate is available.

```
template <class F>
static auto integrate(F f, Real a, Real b, Real* pL1 = nullptr)->decltype(std::declv↵
al<F>()(std::declval<Real>()))
```

Integrates *f* over (a,b), and optionally sets `*pL1` to the L1 norm of the returned value: if this is substantially larger than the return value, then the sum was ill-conditioned. Note however, that no error estimate is available. This function supports both finite and infinite *a* and *b*, as long as `a < b`.

The Gaussian quadrature routine support both real and complex-valued quadrature. For example, the Lambert-W function admits the integral representation

W(z) = 1/2Π $\int_{-\Pi}^{\Pi}$ ((1- v cot(v))^2 + v^2)/(z + v csc(v) exp(-v cot(v))) dv

so it can be effectively computed via Gaussian quadrature using the following code:

```
Complex z{2, 3};
auto lw = [&z](Real v)->Complex {
  using std::cos;
  using std::sin;
  using std::exp;
  Real sinv = sin(v);
  Real cosv = cos(v);

  Real cotv = cosv/sinv;
  Real cscv = 1/sinv;
  Real t = (1-v*cotv)*(1-v*cotv) + v*v;
  Real x = v*cscv*exp(-v*cotv);
  Complex den = z + x;
  Complex num = t*(z/pi<Real>());
  Complex res = num/den;
  return res;
};

boost::math::quadrature::gauss<Real, 30> integrator;
Complex W = integrator.integrate(lw, (Real) 0, pi<Real>());
```

Choosing the number of points

Internally class `gauss` has pre-computed tables of abscissa and weights for 7, 15, 20, 25 and 30 points at up to 100-decimal digit precision. That means that using for example, `gauss<double, 30>::integrate` incurs absolutely zero set-up overhead from computing the abscissa/weight pairs. When using multiprecision types with less than 100 digits of precision, then there is a small initial one time cost, while the abscissa/weight pairs are constructed from strings.

However, for types with higher precision, or numbers of points other than those given above, the abscissa/weight pairs are computed when first needed and then cached for future use, which does incur a noticeable overhead. If this is likely to be an issue, then

- Defining BOOST_MATH_GAUSS_NO_COMPUTE_ON_DEMAND will result in a compile-time error, whenever a combination of number type and number of points is used which does not have pre-computed values.
- There is a program gauss_kronrod_constants.cpp which was used to provide the pre-computed values already in gauss.hpp. The program can be trivially modified to generate code and constants for other precisions and numbers of points.

Examples

We'll begin by integrating t^2 atan(t) over (0,1) using a 7 term Gauss-Legendre rule, and begin by defining the function to integrate as a C++ lambda expression:

```
using namespace boost::math::quadrature;

auto f = [](const double& t) { return t * t * std::atan(t); };
```

Integration is simply a matter of calling the `gauss<double, 7>::integrate` method:

```
double Q = gauss<double, 7>::integrate(f, 0, 1);
```

Which yields a value 0.2106572512 accurate to 1e-10.

For more accurate evaluations, we'll move to a multiprecision type and use a 20-point integration scheme:

```
using boost::multiprecision::cpp_bin_float_quad;

auto f2 = [](const cpp_bin_float_quad& t) { return t * t * atan(t); };

cpp_bin_float_quad Q2 = gauss<cpp_bin_float_quad, 20>::integrate(f2, 0, 1);
```

Which yields 0.2106572512258069881080923020669, which is accurate to 5e-28.

Gauss-Kronrod Quadrature

Overview

Gauss-Kronrod quadrature is an extension of Gaussian quadrature which provides an a posteriori error estimate for the integral.

The idea behind Gaussian quadrature is to choose *n* nodes and weights in such a way that polynomials of order *2n-1* are integrated exactly. However, integration of polynomials is trivial, so it is rarely done via numerical methods. Instead, transcendental and numerically defined functions are integrated via Gaussian quadrature, and the defining problem becomes how to estimate the remainder. Gaussian quadrature alone (without some form of interval splitting) cannot answer this question.

It is possible to compute a Gaussian quadrature of order *n* and another of order (say) *2n+1*, and use the difference as an error estimate. However, this is not optimal, as the zeros of the Legendre polynomials (nodes of the Gaussian quadrature) are never the same for different orders, so *3n+1* function evaluations must be performed. Kronrod considered the problem of how to interleave nodes into a Gaussian quadrature in such a way that all previous function evaluations can be reused, while increasing the order of polynomials that can be integrated exactly. This allows an a posteriori error estimate to be provided while still preserving exponential convergence. Kronrod discovered that by adding *n+1* nodes (computed from the zeros of the Legendre-Stieltjes polynomials) to a Gaussian quadrature of order *n*, he could integrate polynomials of order *3n+1*.

The integration routines provided here will perform either adaptive or non-adaptive quadrature, they should be chosen for the integration of smooth functions with no end point singularities. For difficult functions, or those with end point singularities, please refer to the double-exponential integration schemes.

```
#include <boost/math/quadrature/gauss_kronrod.hpp>

template <class Real, unsigned N, class Policy = boost::math::policies::policy<> >
class gauss_kronrod
{
   static const RandomAccessContainer& abscissa();
   static const RandomAccessContainer& weights();

   template <class F>
   static auto integrate(F f,
                         Real a, Real b,
                         unsigned max_depth = 15,
                         Real tol = tools::root_epsilon<Real>(),
                         Real* error = nullptr,
                      Real* pL1 = nullptr)->decltype(std::declval<F>()(std::declval<Real>()));
};
```

Description

```
static const RandomAccessContainer& abscissa();
static const RandomAccessContainer& weights();
```

These functions provide direct access to the abscissa and weights used to perform the quadrature: the return type depends on the *Points* template parameter, but is always a RandomAccessContainer type. Note that only positive (or zero) abscissa and weights are stored, and that they contain both the Gauss and Kronrod points.

```
template <class F>
static auto integrate(F f,
                            Real a, Real b,
                            unsigned max_depth = 15,
                            Real tol = tools::root_epsilon<Real>(),
                            Real* error = nullptr,
                            Real* pL1 = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>()));
```

Performs adaptive Gauss-Kronrod quadrature on function *f* over the range (a,b).

max_depth sets the maximum number of interval splittings permitted before stopping. Set this to zero for non-adaptive quadrature. Note that the algorithm descends the tree depth first, so only "difficult" areas of the integral result in interval splitting.

tol Sets the maximum relative error in the result, this should not be set too close to machine epsilon or the function will simply thrash and probably not return accurate results. On the other hand the default may be overly-pressimistic.

error When non-null, `*error` is set to the difference between the (N-1)/2 point Gauss approximation and the N-point Gauss-Kronrod approximation.

pL1 When non-null, `*pL1` is set to the L1 norm of the result, if there is a significant difference between this and the returned value, then the result is likely to be ill-conditioned.

Choosing the number of points

The number of points specified in the *Points* template parameter must be an odd number: giving a (N-1)/2 Gauss quadrature as the comparison for error estimation.

Internally class `gauss_kronrod` has pre-computed tables of abscissa and weights for 15, 31, 41, 51 and 61 Gauss-Kronrod points at up to 100-decimal digit precision. That means that using for example, `gauss_kronrod<double, 31>::integrate` incurs absolutely zero set-up overhead from computing the abscissa/weight pairs. When using multiprecision types with less than 100 digits of precision, then there is a small initial one time cost, while the abscissa/weight pairs are constructed from strings.

However, for types with higher precision, or numbers of points other than those given above, the abscissa/weight pairs are computed when first needed and then cached for future use, which does incur a noticeable overhead. If this is likely to be an issue, then:

- Defining BOOST_MATH_GAUSS_NO_COMPUTE_ON_DEMAND will result in a compile-time error, whenever a combination of number type and number of points is used which does not have pre-computed values.
- There is a program gauss_kronrod_constants.cpp which was used to provide the pre-computed values already in gauss_kronrod.hpp. The program can be trivially modified to generate code and constants for other precisions and numbers of points.

Complex Quadrature

The Gauss-Kronrod quadrature support integrands defined on the real line and returning complex values. In this case, the template argument is the real type, and the complex type is deduced via the return type of the function.

Examples

We'll begin by integrating exp(-t^2/2) over (0,+∞) using a 7 term Gauss rule and 15 term Kronrod rule, and begin by defining the function to integrate as a C++ lambda expression:

```
using namespace boost::math::quadrature;

auto f1 = [](double t) { return std::exp(-t*t / 2); };
```

W'll start off with a one shot (ie non-adaptive) integration, and keep track of the estimated error:

```
double error;
double Q = gauss_kronrod<double, 15>::integrate(f1, 0, std::numeric_limits<double>::infin↲
ity(), 0, 0, &error);
```

This yields Q = 1.25348207361, which has an absolute error of 1e-4 compared to the estimated error of 5e-3: this is fairly typical, with the difference between Gauss and Gauss-Kronrod schemes being much higher than the actual error. Before moving on to adaptive quadrature, lets try again with more points, in fact with the largest Gauss-Kronrod scheme we have cached (30/61):

```
Q = gauss_kronrod<double, 61>::integrate(f1, 0, std::numeric_limits<double>::infin↲
ity(), 0, 0, &error);
```

This yields an absolute error of 3e-15 against an estimate of 1e-8, which is about as good as we're going to get at double precision

However, instead of continuing with ever more points, lets switch to adaptive integration, and set the desired relative error to 1e-14 against a maximum depth of 5:

```
Q = gauss_kronrod<double, 15>::integrate(f1, 0, std::numeric_limits<double>::infinity(), 5, 1e-
14, &error);
```

This yields an actual error of zero, against an estimate of 4e-15. In fact in this case the requested tolerance was almost certainly set too low: as we've seen above, for smooth functions, the precision achieved is often double that of the estimate, so if we integrate with a tolerance of 1e-9:

```
Q = gauss_kronrod<double, 15>::integrate(f1, 0, std::numeric_limits<double>::infinity(), 5, 1e-
9, &error);
```

We still achieve 1e-15 precision, with an error estimate of 1e-10.

Caveats

For essentially all analytic integrands bounded on the domain, the error estimates provided by the routine are woefully pessimistic. In fact, in this case the error is very nearly equal to the error of the Gaussian quadrature formula of order `(N-1)/2`, whereas the Kronrod extension converges exponentially beyond the Gaussian estimate. Very sophisticated method exist to estimate the error, but all require the integrand to lie in a particular function space. A more sophisticated a posteriori error estimate for an element of a particular function space is left to the user.

These routines are deliberately kept relatively simple: when they work, they work very well and very rapidly. However, no effort has been made to make these routines work well with end-point singularities or other "difficult" integrals. In such cases please use one of the double-exponential integration schemes which are generally much more robust.

References

- Kronrod, Aleksandr Semenovish (1965), *Nodes and weights of quadrature formulas. Sixteen-place tables*, New York: Consultants Bureau
- Dirk P. Laurie, *Calculation of Gauss-Kronrod Quadrature Rules*, Mathematics of Computation, Volume 66, Number 219, 1997
- Gonnet, Pedro, *A Review of Error Estimation in Adaptive Quadrature*, https://arxiv.org/pdf/1003.4629.pdf

Double-exponential quadrature

Overview

Synopsis

```
    #include <boost/math/quadrature/tanh_sinh.hpp>
    #include <boost/math/quadrature/exp_sinh.hpp>
    #include <boost/math/quadrature/sinh_sinh.hpp>

    namespace boost{ namespace math{

    template<class Real>
    class tanh_sinh
    {
    public:
        tanh_sinh(size_t max_refinements = 15, const Real& min_comple↵
ment = tools::min_value<Real>() * 4)

        template<class F>
        auto integrate(const F f, Real a, Real b,
                       Real tolerance = tools::root_epsilon<Real>(),
                       Real* error = nullptr,
                       Real* L1 = nullptr,
                       std::size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;

        template<class F>
        auto integrate(const F f, Real
                       tolerance = tools::root_epsilon<Real>(),
                       Real* error = nullptr,
                       Real* L1 = nullptr,
                       std::size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;

    };

    template<class Real>
    class exp_sinh
    {
    public:
        exp_sinh(size_t max_refinements = 9);

        template<class F>
        auto integrate(const F f, Real a, Real b,
                       Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                       Real* error = nullptr,
                       Real* L1 = nullptr,
                       size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;
        template<class F>
        auto integrate(const F f,
                       Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                       Real* error = nullptr,
                       Real* L1 = nullptr,
                       size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;
    };

    template<class Real>
    class sinh_sinh
    {
    public:
        sinh_sinh(size_t max_refinements = 9);

        template<class F>
        auto integrate(const F f,
                       Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                       Real* error = nullptr,
                       Real* L1 = nullptr,
                       size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;
    };

}}
```

These three integration routines provide robust general purpose quadrature, each having a "native" range over which quadrature is performed. For example, the `sinh_sinh` quadrature integrates over the entire real line, the `tanh_sinh` over (-1, 1), and the `exp_sinh` over (0, ∞). The latter integrators also have auxilliary ranges which are handled via a change of variables on the function being integrated, so that the `tanh_sinh` can handle integration over (a, b), and `exp_sinh` over (a, ∞) and(-∞, b).

Like the other quadrature routines in Boost, these routines support both real and complex-valued integrands.

The `integrate` methods which do not specify a range always integrate over the native range of the method, and generally are the most efficient and produce the smallest code, on the other hand the methods which do specify the bounds of integration are the most general, and use argument transformations which are generally very robust. The following table summarizes the ranges supported by each method:

Integrator	**Native range**	**Other supported ranges**	**Comments**
tanh_sinh	(-1,1)	(a,b) (a,∞) (-∞,b) (-∞,∞)	Special care is taken for endpoints at or near zero to ensure that abscissa values are calculated without the loss of precision that would normally occur. Likewise when transforming to an infinite endpoint, the additional information which tanh_sinh has internally on abscissa values is used to ensure no loss of precision during the transformation.
exp_sinh	(0,∞)	(a,∞) (-∞,0) (-∞,b)	
sinh_sinh	(-∞,∞)		

tanh_sinh

```
template<class Real>
class tanh_sinh
{
public:
    tanh_sinh(size_t max_refinements = 15, const Real& min_comple↵
ment = tools::min_value<Real>() * 4)

    template<class F>
    auto integrate(const F f, Real a, Real b,
                   Real tolerance = tools::root_epsilon<Real>(),
                   Real* error = nullptr,
                   Real* L1 = nullptr,
                   std::size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;

    template<class F>
    auto integrate(const F f, Real
                   tolerance = tools::root_epsilon<Real>(),
                   Real* error = nullptr,
                   Real* L1 = nullptr,
                   std::size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;

};
```

The tanh-sinh quadrature routine provided by boost is a rapidly convergent numerical integration scheme for holomorphic integrands. By this we mean that the integrand is the restriction to the real line of a complex-differentiable function which is bounded on the interior of the unit disk $|z| < 1$, so that it lies within the so-called Hardy space. If your integrand obeys these conditions, it can be shown that tanh-sinh integration is optimal, in the sense that it requires the fewest function evaluations for a given accuracy of any quadrature algorithm for a random element from the Hardy space. A basic example of how to use the tanh-sinh quadrature is shown below:

```
tanh_sinh<double> integrator;
auto f = [](double x) { return 5*x + 7; };
// Integrate over native bounds of (-1,1):
double Q = integrator.integrate(f);
// Integrate over (0,1.1) instead:
Q = integrator.integrate(f, 0.0, 1.1);
```

The basic idea of tanh-sinh quadrature is that a variable transformation can cause the endpoint derivatives to decay rapidly. When the derivatives at the endpoints decay much faster than the Bernoulli numbers grow, the Euler-Maclaurin summation formula tells us that simple trapezoidal quadrature converges faster than any power of h. That means the number of correct digits of the result should roughly double with each new level of integration (halving of h), Hence the default termination condition for integration is usually set to the square root of machine epsilon. Most well-behaved integrals should converge to full machine precision with this termination condition, and in 6 or fewer levels at double precision, or 7 or fewer levels for quad precision.

One very nice property of tanh-sinh quadrature is that it can handle singularities at the endpoints of the integration domain. For instance, the following integrand, singular at both endpoints, can be efficiently evaluated to 100 binary digits:

```
auto f = [](Real x) { return log(x)*log1p(-x); };
Real Q = integrator.integrate(f, (Real) 0, (Real) 1);
```

Now onto the caveats: As stated before, the integrands must lie in a Hardy space to ensure rapid convergence. Attempting to integrate a function which is not bounded on the unit disk by tanh-sinh can lead to very slow convergence. For example, take the Runge function:

```
auto f1 = [](double t) { return 1/(1+25*t*t); };
Q = integrator.integrate(f1, (double) -1, (double) 1);
```

This function has poles at ± /5, and as such it is not bounded on the unit disk. However, the related function

```
auto f2 = [](double t) { return 1/(1+0.04*t*t); };
Q = integrator.integrate(f2, (double) -1, (double) 1);
```

has poles outside the unit disk (at ± 5), and is therefore in the Hardy space. Our benchmarks show that the second integration is performed 22x faster than the first! If you do not understand the structure of your integrand in the complex plane, do performance testing before deployment.

Like the trapezoidal quadrature, the tanh-sinh quadrature produces an estimate of the L_1 norm of the integral along with the requested integral. This is to establish a scale against which to measure the tolerance, and to provide an estimate of the condition number of the summation. This can be queried as follows:

```
tanh_sinh<double> integrator;
auto f = [](double x) { return 5*x + 7; };
double termination = std::sqrt(std::numeric_limits<double>::epsilon());
double error;
double L1;
size_t levels;
double Q = integrator.integrate(f, 0.0, 1.0, termination, &error, &L1, &levels);
double condition_number = L1/std::abs(Q);
```

If the condition number is large, the computed integral is worthless: typically one can assume that Q has lost one digit of precision when the condition number of O(10^Q). The returned error term is not the actual error in the result, but merely an a posteriori error estimate. It is the absolute difference between the last two approximations, and for well behaved integrals, the actual error should be very much smaller than this. The following table illustrates how the errors and conditioning vary for few sample integrals, in each case the termination condition was set to the square root of epsilon, and all tests were conducted in double precision:

Integral	Range	Error	Actual measured error	Levels	Condition Number	Comments
`5 * x + 7`	(0,1)	3.5e-15	0	5	1	This trivial case shows just how accurate these methods can be.
`log(x) * log(x)`	0, 1)	0	0	5	1	This is an example of an integral that Gaussian integrators fail to handle.
`exp(-x) / sqrt(x)`	(0,+∞)	8.0e-10	1.1e-15	5	1	Gaussian integrators typically fail to handle the singularities at the endpoints of this one.
`x * sin(2 * exp(2 * sin(2 * exp(2 * x))))`	(-1,1)	7.2e-16	4.9e-17	9	1.89	This is a truely horrible integral that oscillates wildly and unpredictably with some very sharp "spikes" in it's graph. The higher number of levels used reflects the difficulty of sampling the more extreme features.

Integral	**Range**	**Error**	**Actual measured error**	**Levels**	**Condition Number**	**Comments**
`x == 0 ? 1 : sin(x) / x`	(-∞, ∞)	3.0e-1	4.0e-1	15	159	This highly oscillatory integral isn't handled at all well by tanh-sinh quadrature: there is so much cancellation in the sum that the result is essentially worthless. The argument transformation of the infinite integral behaves somewhat badly as well, in fact we do *slightly* better integrating over 2 symmetrical and large finite limits.
`sqrt(x / (1 - x * x))`	(0,1)	1e-8	1e-8	5	1	This an example of an integral that has all its area close to a non-zero endpoint, the problem here is that the function being integrated returns "garbage" values for x very close to 1. We can easily fix this issue by passing a 2 argument functor to the integrator: the second argument gives the distance to the nearest endpoint, and we can use that information to return accurate values, and thus fix the integral calculation.

Integral	Range	Error	Actual measured error	Levels	Condition Number	Comments
`x < 0.5 ? sqrt(x) / sqrt(1 - x * x) : sqrt(x / ((x + 1) * (xc)))`	(0,1)	0	0	5	1	This is the 2-argument version of the previous integral, the second argument *xc* is `1-x` in this case, and we use $1\text{-}x^2$ == (1-x)(1+x) to calculate $1\text{-}x^2$ with greater accuracy.

Although the tanh-sinh quadrature can compute integral over infinite domains by variable transformations, these transformations can create a very poorly behaved integrand. For this reason, double-exponential variable transformations have been provided that allow stable computation over infinite domains; these being the exp-sinh and sinh-sinh quadrature.

Complex integrals

The tanh_sinh integrator supports integration of functions which return complex results, for example the sine-integral `Si(z)` has the integral representation:

$$Si(z) = \frac{\pi}{2} - \int_0^{\frac{\pi}{2}} e^{-z\cos(t)}\cos(z\sin(t))dt$$

Which we can code up directly as:

```
template <class Complex>
Complex Si(Complex z)
{
   typedef typename Complex::value_type value_type;
   using std::sin;  using std::cos; using std::exp;
   auto f = [&z](value_type t) { return -exp(-z * cos(t)) * cos(z * sin(t)); };
   boost::math::quadrature::tanh_sinh<value_type> integrator;
   return integrator.integrate(f, 0, boost::math::con↲
stants::half_pi<value_type>()) + boost::math::constants::half_pi<value_type>();
}
```

Handling functions with large features near an endpoint with tanh-sinh quadrature

Tanh-sinh quadrature has a unique feature which makes it well suited to handling integrals with either singularities or large features of interest near one or both endpoints, it turns out that when we calculate and store the abscissa values at which we will be evaluating the function, we can equally well calculate the difference between the abscissa value and the nearest endpoint. This makes it possible to perform quadrature arbitrarily close to an endpoint, without suffering cancellation error. Note however, that we never actually reach the endpoint, so any endpoint singularity will always be excluded from the quadrature.

The tanh_sinh integration routine will use this additional information internally when performing range transformation, so that for example, if one end of the range is zero (or infinite) then our transformations will get arbitrarily close to the endpoint without precision loss.

However, there are some integrals which may have all of their area near *both* endpoints, or else near the non-zero endpoint, and in that situation there is a very real risk of loss of precision. For example:

```
tanh_sinh<double> integrator;
auto f = [](double x) { return sqrt(x / (1 - x * x); };
double Q = integrator.integrate(f, 0.0, 1.0);
```

Results in very low accuracy as all the area of the integral is near 1, and the `1 - x * x` term suffers from cancellation error here.

However, both of tanh_sinh's integration routines will automatically handle 2 argument functors: in this case the first argument is the abscissa value as before, while the second is the distance to the nearest endpoint, ie `a - x` or `b - x` if we are integrating over (a,b). You can always differentiate between these 2 cases because the second argument will be negative if we are nearer to the left endpoint.

Knowing this, we can rewrite our lambda expression to take advantage of this additional information:

```
tanh_sinh<double> integrator;
auto f = [](double x, double xc) { re↵
turn x <= 0.5 ? sqrt(x) / sqrt(1 - x * x) : sqrt(x / ((x + 1) * (xc))); };
double Q = integrator.integrate(f, 0.0, 1.0);
```

Not only is this form accurate to full machine precision, but it converges to the result faster as well.

sinh_sinh

```
template<class Real>
class sinh_sinh
{
public:
    sinh_sinh(size_t max_refinements = 9);

    template<class F>
    auto integrate(const F f,
                   Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                   Real* error = nullptr,
                   Real* L1 = nullptr,
                   size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;;
};
```

The sinh-sinh quadrature allows computation over the entire real line, and is called as follows:

```
sinh_sinh<double> integrator;
auto f = [](double x) { return exp(-x*x); };
double error;
double L1;
double Q = integrator.integrate(f, &error, &L1);
```

Note that the limits of integration are understood to be (-∞, ∞).

Complex valued integrands are supported as well, for example the Dirichlet Eta function can be represented via:

$$\eta(s) = \int_{-\infty}^{\infty} \frac{\left(\frac{1}{2} + it\right)^{-s}}{e^{\pi t} + e^{-\pi t}} dt$$

which we can directly code up as:

```
template <class Complex>
Complex eta(Complex s)
{
   typedef typename Complex::value_type value_type;
   using std::pow;  using std::exp;
   Complex i(0, 1);
   value_type pi = boost::math::constants::pi<value_type>();
   auto f = [&, s, i](value_type t) { re↵
turn pow(0.5 + i * t, -s) / (exp(pi * t) + exp(-pi * t)); };
   boost::math::quadrature::sinh_sinh<value_type> integrator;
   return integrator.integrate(f);
}
```

exp_sinh

```
template<class Real>
class exp_sinh
{
public:
    exp_sinh(size_t max_refinements = 9);

    template<class F>
    auto integrate(const F f, Real a, Real b,
                   Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                   Real* error = nullptr,
                   Real* L1 = nullptr,
                   size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;
    template<class F>
    auto integrate(const F f,
                   Real tol = sqrt(std::numeric_limits<Real>::epsilon()),
                   Real* error = nullptr,
                   Real* L1 = nullptr,
                   size_t* levels = nullptr)->decltype(std::declval<F>()(std::declv↵
al<Real>())) const;
};
```

For half-infinite intervals, the exp-sinh quadrature is provided:

```
exp_sinh<double> integrator;
auto f = [](double x) { return exp(-3*x); };
double termination = sqrt(std::numeric_limits<double>::epsilon());
double error;
double L1;
double Q = integrator.integrate(f, termination, &error, &L1);
```

The native integration range of this integrator is (0, ∞), but we also support (a, ∞), (-∞, 0) and (-∞, b) via argument transformations.

Endpoint singularities and complex-valued integrands are supported by exp-sinh.

For example, the modified Bessel function K can be represented via:

$$K_\alpha(x) = \int_o^\infty e^{-x\cosh(t)} \cosh(\alpha t)dt$$

Which we can code up as:

```
template <class Complex>
Complex bessel_K(Complex alpha, Complex z)
{
   typedef typename Complex::value_type value_type;
   using std::cosh;  using std::exp;
   auto f = [&, alpha, z](value_type t)
   {
      value_type ct = cosh(t);
      if (ct > log(std::numeric_limits<value_type>::max()))
         return Complex(0);
      return exp(-z * ct) * cosh(alpha * t);
   };
   boost::math::quadrature::exp_sinh<value_type> integrator;
   return integrator.integrate(f);
}
```

The only wrinkle in the above code is the need to check for large `cosh(t)` in which case we assume that `exp(-x cosh(t))` tends to zero faster than `cosh(alpha x)` tends to infinity and return `0`. Without that check we end up with `0 * Infinity` as the result (a NaN).

Setting the Termination Condition for Integration

The integrate method for all three double-exponential quadratures supports *tolerance* argument that acts as the termination condition for integration.

The tolerance is met when two subsequent estimates of the integral have absolute error less than `tolerance*L1`.

It is highly recommended that the tolerance be left at the default value of $\sqrt{\varepsilon}$, or something similar. Since double exponential quadrature converges exponentially fast for functions in Hardy spaces, then once the routine has **proved** that the error is ~$\sqrt{\varepsilon}$, then the error should in fact be ~ε.

If you request that the error be ~ε, this tolerance might never be achieved (as the summation is not stabilized ala Kahan), and the routine will simply flounder, dividing the interval in half in order to increase the precision of the integrand, only to be thwarted by floating point roundoff.

If for some reason, the default value doesn't quite achieve full precision, then you could try something a little smaller such as $\sqrt{\varepsilon}/4$ or $\varepsilon^{2/3}$. However, more likely, you need to check that your function to be integrated is able to return accurate values, and that there are no other issues with your integration scheme.

Setting the Maximum Interval Halvings and Memory Requirements

The max interval halvings is the maximum number of times the interval can be cut in half before giving up. If the accuracy is not met at that time, the routine returns its best estimate, along with the `error` and `L1`, which allows the user to decide if another quadrature routine should be employed.

An example of this is

```
double tol = std::sqrt(std::numeric_limits<double>::epsilon());
size_t max_halvings = 12;
tanh_sinh<double> integrator(max_halvings);
auto f = [](double x) { return 5*x + 7; };
double error, L1;
double Q = integrator.integrate(f, (double) 0, (double) 1, &error, &L1);
if (error*L1 > 0.01)
{
    Q = some_other_quadrature_method(f, (double) 0, (double) 1);
}
```

It's important to remember that the number of sample points doubles with each new level, as does the memory footprint of the integrator object. Further, if the integral is smooth, then the precision will be doubling with each new level, so that for example, many integrals can achieve 100 decimal digit precision after just 7 levels. That said, abscissa-weight pairs for new levels are computed only when a new level is actually required (see thread safety), none the less, you should avoid setting the maximum arbitrarily high "just in case" as the time and space requirements for a large number of levels can quickly grow out of control.

Thread Safety

All three of the double-exponential integrators are thread safe as long as BOOST_MATH_NO_ATOMIC_INT is not set. Since the integrators store a large amount of fairly hard to compute data, it is recommended that these objects are stored and reused as much as possible.

Internally all three of the double-exponential integrators use the same caching strategy: they allocate all the vectors needed to store the maximum permitted levels, but only populate the first few levels when constructed. This means a minimal amount of memory is actually allocated when the integrator is first constructed, and already populated levels can be accessed via a lockfree atomic read, and only populating new levels requires a thread lock.

In addition, the three built in types (plus `__float128` when available), have the first 7 levels pre-computed: this is generally sufficient for the vast majority of integrals - even at quad precision - and means that integrators for these types are relatively cheap to construct.

Caveats

A few things to keep in mind while using the tanh-sinh, exp-sinh, and sinh-sinh quadratures:

These routines are **very** aggressive about approaching the endpoint singularities. This allows lots of significant digits to be extracted, but also has another problem: Roundoff error can cause the function to be evaluated at the endpoints. A few ways to avoid this: Narrow up the bounds of integration to say, [a + ε, b - ε], make sure (a+b)/2 and (b-a)/2 are representable, and finally, if you think the compromise between accuracy an usability has gone too far in the direction of accuracy, file a ticket.

Both exp-sinh and sinh-sinh quadratures evaluate the functions they are passed at **very** large argument. You might understand that x^{12}exp(-x) is should be zero when x^{12} overflows, but IEEE floating point arithmetic does not. Hence `std::pow(x, 12)*std::exp(-x)` is an indeterminate form whenever `std::pow(x, 12)` overflows. So make sure your functions have the correct limiting behavior; for example

```
auto f = [](double x) {
    double t = exp(-x);
    if(t == 0)
    {
        return 0;
    }
    return t*pow(x, 12);
};
```

has the correct behavior for large *x*, but `auto f = [](double x) { return exp(-x)*pow(x, 12); };` does not.

Oscillatory integrals, such as the sinc integral, are poorly approximated by double-exponential quadrature. Fortunately the error estimates and L1 norm are massive for these integrals, but nonetheless, oscillatory integrals require different techniques.

A special mention should be made about integrating through zero: while our range adaptors preserve precision when one endpoint is zero, things get harder when the origin is neither in the center of the range, nor at an endpoint. Consider integrating:

```
1 / (1 +x^2)
```

Over (a, ∞). As long as `a >= 0` both the tanh_sinh and the exp_sinh integrators will handle this just fine: in fact they provide a rather efficient method for this kind of integral. However, if we have `a < 0` then we are forced to adapt the range in a way that produces abscissa values near zero that have an absolute error of ε, and since all of the area of the integral is near zero both integrators thrash around trying to reach the target accuracy, but never actually get there for `a << 0`. On the other hand, the simple expedient of breaking the integral into two domains: (a, 0) and (0, b) and integrating each seperately using the tanh-sinh integrator, works just fine.

Finally, some endpoint singularities are too strong to be handled by tanh_sinh or equivalent methods, for example consider integrating the function:

```
double p = some_value;
tanh_sinh<double> integrator;
auto f = [&](double x){ return pow(tan(x), p); };
auto Q = integrator.integrate(f, 0, constants::half_pi<double>());
```

The first problem with this function, is that the singularity is at π/2, so if we're integrating over (0, π/2) then we can never approach closer to the singularity than ε, and for p less than but close to 1, we need to get *very* close to the singularity to find all the area under the function. If we recall the identity `tan(π/2 - x) == 1/tan(x)` then we can rewrite the function like this:

```
auto f = [&](double x){ return pow(tan(x), -p); };
```

And now the singularity is at the origin and we can get much closer to it when evaluating the integral: all we have done is swap the integral endpoints over.

This actually works just fine for p < 0.95, but after that the tanh_sinh integrator starts thrashing around and is unable to converge on the integral. The problem is actually a lack of exponent range: if we simply swap type double for something with a greater exponent range (an 80-bit long double or a quad precision type), then we can get to at least p = 0.99. If we want to go beyond that, or stick with type double, then we have to get smart.

The easiest method is to notice that for small x, then `tan(x) ≅ x`, and so we are simply integrating x^{-p}. Therefore we can use this approximation over (0, small), and integrate numerically from (small, π/2), using ε as a suitable crossover point seems sensible:

```
double p = some_value;
double crossover = std::numeric_limits<double>::epsilon();
tanh_sinh<double> integrator;
auto f = [&](double x){ return pow(tan(x), p); };
auto Q = integrator.integrate(f, crossover, constants::half_pi<double>()) + pow(crossov↵
er, 1 - p) / (1 - p);
```

There is an alternative, more complex method, which is applicable when we are dealing with expressions which can be simplified by evaluating by logs. Let's suppose that as in this case, all the area under the graph is infinitely close to zero, now inagine that we could expand that region out over a much larger range of abscissa values: that's exactly what happens if we perform argument substitution, replacing `x` by `exp(-x)` (note that we must also multiply by the derivative of `exp(-x)`). Now the singularity at zero is moved to +∞, and the π/2 bound to -log(π/2). Initially our argument substituted function looks like:

```
auto f = [&](double x){ return exp(-x) * pow(tan(exp(-x)), -p); };
```

Which is hardly any better, as we still run out of exponent range just as before. However, if we replace `tan(exp(-x))` by `exp(-x)` for suitably small `exp(-x)`, and therefore `x > -log(ε)`, we can greatly simplify the expression and evaluate by logs:

```
auto f = [&](double x)
{
   static const double crossover = -log(std::numeric_limits<double>::epsilon());
   return x > crossover ? exp((p - 1) * x) : exp(-x) * pow(tan(exp(-x)), -p);
};
```

This form integrates just fine over (-log(π/2), +∞) using either the tanh_sinh or exp_sinh classes.

References

- Hidetosi Takahasi and Masatake Mori, *Double Exponential Formulas for Numerical Integration* Publ. Res. Inst. Math. Sci., 9 (1974), pp. 721-741.
- Masatake Mori, *An IMT-Type Double Exponential Formula for Numerical Integration*, Publ RIMS, Kyoto Univ. 14 (1978), 713-729.
- David H. Bailey, Karthik Jeyabalan and Xiaoye S. Li *A Comparison of Three High-Precision Quadrature Schemes* Office of Scientific & Technical Information Technical Reports.
- Tanaka, Ken'ichiro, et al. *Function classes for double exponential integration formulas.* Numerische Mathematik 111.4 (2009): 631-655.

Fourier Integrals

Synopsis

```
#include <boost/math/quadrature/ooura_fourier_integrals.hpp>

namespace boost { namespace math { namespace quadrature {

template<class Real>
class ooura_fourier_sin {
public:
    ooura_fourier_sin(const Real relative_error_tolerance = tools::root_epsi↵
lon<Real>(), size_t levels = sizeof(Real));

    template<class F>
    std::pair<Real, Real> integrate(F const & f, Real omega);

};


template<class Real>
class ooura_fourier_cos {
public:
    ooura_fourier_cos(const Real relative_error_tolerance = tools::root_epsi↵
lon<Real>(), size_t levels = sizeof(Real))

    template<class F>
    std::pair<Real, Real> integrate(F const & f, Real omega);
};

}}} // namespaces
```

Ooura's method for Fourier integrals computes

$$\int_0^\infty f(t)\sin(\omega\, t)\, dt$$

and

$$\int_0^\infty f(t)\cos(\omega\, t)\, dt$$

by a double exponentially decaying transformation. These integrals arise when computing continuous Fourier transform of odd and even functions, respectively. Oscillatory integrals are known to cause trouble for standard quadrature methods, so these routines are provided to cope with the most common oscillatory use case.

The basic usage is shown below:

```
ooura_fourier_sin<double>integrator = ooura_fourier_sin<double>();
// Use the default tolerance root_epsilon and eight levels for type double.

auto f = [](double x)
{ // Simple reciprocal function for sinc.
 return 1 / x;
};

double omega = 1;
std::pair<double, double> result = integrator.integrate(f, omega);
std::cout << "Integral = " << result.first << ", relative error estimate " << res↵
ult.second << std::endl;
```

and compare with the expected value $\pi/2$ of the integral.

```
constexpr double expected = half_pi<double>();
std::cout << "pi/2 =      " << expected << ", difference " << result.first - expec↵
ted << std::endl;
```

The output is

```
integral = 1.5707963267948966, relative error estimate 1.2655356398390254e-11
pi/2 =     1.5707963267948966, difference 0
```

Note

This integrator is more insistent about examining the error estimate, than (say) tanh-sinh, which just returns the value of the integral.

With the macro BOOST_MATH_INSTRUMENT_OOURA defined, we can follow the progress:

```
ooura_fourier_sin with relative error goal 1.4901161193847656e-08 & 8 levels.
h = 1.000000000000000, I_h = 1.571890732004545 = 0x1.92676e56d853500p+0, absolute error estim↵
ate = nan
h = 0.500000000000000, I_h = 1.570793292491940 = 0x1.921f825c076f600p+0, absolute error estim↵
ate = 1.097439512605325e-03
h = 0.250000000000000, I_h = 1.570796326814776 = 0x1.921fb54458acf00p+0, absolute error estim↵
ate = 3.034322835882008e-06
h = 0.125000000000000, I_h = 1.570796326794897 = 0x1.921fb54442d1800p+0, absolute error estim↵
ate = 1.987898734512328e-11
Integral = 1.570796326794897e+00, relative error estimate 1.265535639839025e-11
pi/2 =     1.570796326794897e+00, difference 0.000000000000000e+00
```

Working code of this example is at ooura_fourier_integrals_example.cpp

A classical cosine transform is presented below:

```
auto integrator = ooura_fourier_cos<double>();
// Use the default tolerance root_epsilon and eight levels for type double.

auto f = [](double x)
{ // More complex example function.
 return 1 / (x * x + 1);
};

double omega = 1;

auto [result, relative_error] = integrator.integrate(f, omega);
std::cout << "Integral = " << result << ", relative error estimate " << relative_er↵
ror << std::endl;
```

The value of this integral should be π/(2e) and can be shown :

```
constexpr double expected = half_pi<double>() / e<double>();
std::cout << "pi/(2e) =  " << expected << ", difference " << result - expected << std::endl;
```

or with the macro BOOST_MATH_INSTRUMENT_OOURA defined, we can follow the progress:

```
ooura_fourier_cos with relative error goal 1.4901161193847656e-08 & 8 levels.
epsilon for type = 2.2204460492503131e-16
h = 1.000000000000000, I_h = 0.588268622591776 = 0x1.2d318b7e96dbe00p-1, absolute error estim↵
ate = nan
h = 0.500000000000000, I_h = 0.577871642184837 = 0x1.27decab8f07b200p-1, absolute error estim↵
ate = 1.039698040693926e-02
h = 0.250000000000000, I_h = 0.577863671186883 = 0x1.27ddbf42969be00p-1, absolute error estim↵
ate = 7.970997954576120e-06
h = 0.125000000000000, I_h = 0.577863674895461 = 0x1.27ddbf6271dc000p-1, absolute error estim↵
ate = 3.708578555361441e-09
Integral = 5.778636748954611e-01, relative error estimate 6.417739540441515e-09
pi/(2e)  = 5.778636748954609e-01, difference 2.220446049250313e-16
```

Working code of this example is at ooura_fourier_integrals_consine_example.cpp

Performance

The integrator precomputes nodes and weights, and hence can be reused for many different frequencies with good efficiency. The integrator is pimpl'd and hence can be shared between threads without a `memcpy` of the nodes and weights.

Ooura and Mori's paper identifies criteria for rapid convergence based on the position of the poles of the integrand in the complex plane. If these poles are too close to the real axis the convergence is slow. It is not trivial to predict the convergence rate a priori, so if you are interested in figuring out if the convergence is rapid, compile with `-DBOOST_MATH_INSTRUMENT_OOURA` and some amount of printing will give you a good idea of how well this method is performing.

Higher precision

It is simple to extend to higher precision using Boost.Multiprecision.

```
// Use the default parameters for tolerance root_epsilon and eight levels for a type of 8 bytes.
//auto integrator = ooura_fourier_cos<Real>();
// Decide on a (tight) tolerance.
const Real tol = 2 * std::numeric_limits<Real>::epsilon();
auto integrator = ooura_fourier_cos<Real>(tol, 8); // Loops or gets worse for more than 8.

auto f = [](Real x)
{ // More complex example function.
 return 1 / (x * x + 1);
};

double omega = 1;
auto [result, relative_error] = integrator.integrate(f, omega);
```

```
std::cout << "Integral = " << result << ", relative error estimate " << relative_er↵
ror << std::endl;

const Real expected = half_pi<Real>() / e<Real>(); // Expect integral = 1/(2e)
std::cout << "pi/(2e)  = " << expected << ", difference " << result - expected << std::endl;
```

with output:

```
Integral = 0.57786367489546085895504659165635015 87, relative error estimate ↵
4.6098146845221638952642773126108302 78e-17
pi/(2e) = 0.5778636748954608659545328919193707407, difference -
6.999486300263020581921171645225573 3758e-18
```

And with diagnostics on:

```
ooura_fourier_cos with relative error goal 3.851859888774471706111955885169854637e-34 & 15 ↵
levels.
epsilon for type = 1.925929944387235853055977942584927319e-34
h = 1.000000000000000000000000000000000, I_h = 0.588268622591776615359568690603776 = ↵
0.5882686225917766153595686906037760, absolute error estimate = nan
h = 0.500000000000000000000000000000000, I_h = 0.577871642184837461311756940493259 = ↵
0.5778716421848374613117569404932595, absolute error estimate = ↵
1.039698040693915404781175011051656e-02
h = 0.250000000000000000000000000000000, I_h = 0.577863671186882539559996800783122 = ↵
0.5778636711868825395599968007831220, absolute error estimate = ↵
7.970997954921751760139710137450075e-06
h = 0.125000000000000000000000000000000, I_h = 0.577863674895460885593491133506723 = ↵
0.5778636748954608855934911335067232, absolute error estimate = ↵
3.708578346033494332723601147051768e-09
h = 0.062500000000000000000000000000000, I_h = 0.577863674895460858955046591656350 = ↵
0.5778636748954608589550465916563502, absolute error estimate = ↵
2.663844454185037302771663314961535e-17
h = 0.031250000000000000000000000000000, I_h = 0.577863674895460858955046591656348 = ↵
0.5778636748954608589550465916563484, absolute error estimate = ↵
1.733336949948512267750380148326435e-33
h = 0.015625000000000000000000000000000, I_h = 0.577863674895460858955046591656348 = ↵
0.5778636748954608589550465916563479, absolute error estimate = ↵
4.814824860968089632639944856462318e-34
h = 0.007812500000000000000000000000000, I_h = 0.577863674895460858955046591656347 = ↵
0.5778636748954608589550465916563473, absolute error estimate = ↵
6.740754805355325485695922799047246e-34
h = 0.003906250000000000000000000000000, I_h = 0.577863674895460858955046591656347 = ↵
0.5778636748954608589550465916563475, absolute error estimate = ↵
1.925929944387235853055977942584927e-34
Integral = 5.778636748954608589550465916563475e-01, relative error estimate ↵
3.332844800697411177051445985473052e-34
pi/(2e)  = 5.778636748954608589550465916563481e-01, difference -
6.740754805355325485695922799047246e-34
```

Working code of this example is at ooura_fourier_integrals_multiprecision_example.cpp

For more examples of other functions and tests, see the full test suite at ooura_fourier_integral_test.cpp.

Ngyen and Nuyens make use of Boost.Multiprecision in their extension to multiple dimensions, showing relative errors reducing to ≅ 10^{-2000}!

Rationale

This implementation is base on Ooura's 1999 paper rather than the later 2005 paper. It does not preclude a second future implementation based on the later work.

Some of the features of the Ooura's 2005 paper that were less appealing were:

- The advance of that paper is that one can compute *both* the Fourier sine transform and Fourier cosine transform in a single shot. But there are examples, like sinc integral, where the Fourier sine would converge, but the Fourier cosine would diverge.

- It would force users to live in the complex plane, when many potential applications only need real.

References

- Ooura, Takuya, and Masatake Mori, *A robust double exponential formula for Fourier-type integrals.* Journal of computational and applied mathematics, 112.1-2 (1999): 229-241.
- Ooura, Takuya, *A Double Exponential Formula for the Fourier Transforms.* Publ. RIMS, Kyoto Univ., 41 (2005), 971-977. https://pdfs.semanticscholar.org/16ec/a5d76fd6b3d7acaaff0b2a6e8a70caa70190.pdf
- Khatibi, Arezoo and Khatibi, Omid,*Criteria for the Application of Double Exponential Transformation.* (2017) 1704.05752.pdf.
- Nguyen, Dong T.P. and Nuyens, Dirk, *Multivariate integration over Reals with exponential rate of convergence.* (2016) https://core.ac.uk/download/pdf/80799199.pdf.

Naive Monte Carlo Integration

Synopsis

```
#include <boost/math/quadrature/naive_monte_carlo.hpp>
namespace boost { namespace math { namespace quadrature {

tem↲
plate<class Real, class F, class RNG = std::mt19937_64, class Policy = boost::math::policies::policy<>>
class naive_monte_carlo
{
public:
    naive_monte_carlo(const F& integrand,
                      std::vector<std::pair<Real, Real>> const & bounds,
                      Real error_goal,
                      bool singular = true,
                      size_t threads = std::thread::hardware_concurrency());

    std::future<Real> integrate();

    void cancel();

    Real current_error_estimate() const;

    std::chrono::duration<Real> estimated_time_to_completion() const;

    void update_target_error(Real new_target_error);

    Real progress() const;

    Real current_estimate() const;

    size_t calls() const;
};
}}} // namespaces
```

Description

The class `naive_monte_carlo` performs Monte-Carlo integration on a square integrable function *f* on a domain Ω. The theoretical background of Monte-Carlo integration is nicely discussed at Wikipedia, and as such will not be discussed here. However, despite being "naive", it is a mistake to assume that naive Monte-Carlo integration is not powerful, as the simplicity of the method affords a robustness not easily provided by more sophisticated tools. The multithreaded nature of the routine allows us to compute a large number of sample points with great speed, and hence the slow convergence is mitigated by exploiting the full power of modern hardware.

The naive Monte-Carlo integration provided by Boost exemplifies the programming techniques needed to cope with high-performance computing. For instance, since the convergence is only $(N^{-1/2})$, the compute time is very sensitive to the error goal. Users can easily specify an error goal which causes computation to last months-or just a few seconds. Without progress reporting, this situation is disorienting and causes the user to behave in a paranoid manner. Even with progress reporting, a user might need to cancel a job due to shifting priorities of the employing institution, and as such cancellation must be supported. A cancelled job which returns no results is wasted, so the cancellation must be graceful, returning the best estimate of the result thus far. In addition, a task might finish, and the user may well decide to try for a lower error bound. Hence restarting without loss of the preceding effort must be supported. Finally, on an HPC system, we generally wish to use all available threads. But if the computation is performed on a users workstation, employing every thread will cause the browser, email, or music apps to become unresponsive, so leaving a single thread available for other apps is appreciated.

All these use cases are supported, so let's get to the code:

```
// Define a function to integrate:
auto g = [](std::vector<double> const & x)
{
  constexpr const double A = 1.0 / (M_PI * M_PI * M_PI);
  return A / (1.0 - cos(x[0])*cos(x[1])*cos(x[2]));
};
std::vector<std::pair<double, double>> bounds{{0, M_PI}, {0, M_PI}, {0, M_PI}};
double error_goal = 0.001
naive_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal);

std::future<double> task = mc.integrate();
while (task.wait_for(std::chrono::seconds(1)) != std::future_status::ready)
{
    // The user must decide on a reasonable way to display the progress depending on their en↲
vironment:
    display_progress(mc.progress(),
                     mc.current_error_estimate(),
                     mc.current_estimate(),
                     mc.estimated_time_to_completion());
    if (some_signal_heard())
    {
        mc.cancel();
        std::cout << "\nCancelling because this is too slow!\n";
    }
}
double y = task.get();
```

First off, we define the function we wish to integrate. This function must accept a `std::vector<Real> const &`, and return a `Real`. Next, we define the domain of integration. Infinite domains are indicated by the bound `std::numeric_limits<Real>::infinity()`. The call

```
naive_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal);
```

creates an instance of the monte carlo integrator. This is also where the number of threads can be set, for instance

```
naive_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal, true, std::thread::hardware_con↲
currency() - 1);
```

might be more appropriate for running on a user's hardware (the default taking all the threads). The call to `integrate()` does not return the value of the integral, but rather a `std::future<Real>`. This allows us to do progress reporting from the master thread via

```
while (task.wait_for(std::chrono::seconds(1)) != std::future_status::ready)
{
    // some reasonable method of displaying progress, based on the requirements of your app.
    display_progress(mc.progress(),
                     mc.current_error_estimate(),
                     mc.current_estimate(),
                     mc.estimated_time_to_completion());
}
```

The file `example/naive_monte_carlo_example.cpp` has an implementation of `display_progress` which is reasonable for command line apps. In addition, we can call `mc.cancel()` in this loop to stop the integration. Progress reporting is especially useful if you accidentally pass in an integrand which is not square integrable-the variance increases without bound, and the progress decreases from some noisy initial value down to zero with time. Calling `task.get()` returns the current estimate. Once the future is ready, we can get the value of the integral via

```
double result = task.get();
```

At this point, the user may wish to reduce the error goal. This is achieved by

```
double new_target_error = 0.0005;
mc.update_target_error(new_target_error);
task = mc.integrate();
y = task.get();
```

There is one additional "advanced" parameter: Whether or not the integrand is singular on the boundary. If the integrand is **not** singular on the boundary, then the integrand is evaluated over the closed set $\prod_\mathrm{i}$ [a_i, b_i]. If the integrand is singular (the default) then the integrand is evaluated over the closed set $\prod_\mathrm{i}$ [/a(1+ε)/, /b(1-ε)/]. (Note that there is sadly no such thing as an open set in floating point arithmetic.) When does the difference matter? Recall the stricture to never peel a high-dimensional orange, because when you do, nothing is left. The same idea applied here. The fraction of the volume within a distance ε of the boundary is approximately ε*d*, where *d* is the number of dimensions. If the number of dimensions is large and the precision of the type is low, then it is possible that no correct digits will be obtained. If the integrand is singular on the boundary, you have no options; you simply must resort to higher precision computations. If the integrand is not singular on the boundary, then you can tell this to the integration routine via

```
naive_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal, /*singular = */ false);
```

and this problem will not be encountered. In practice, you will need ~1,000 dimensions for this to be relevant in 16 bit floating point, ~100,000 dimensions in 32 bit floating point, and an astronomical number of dimensions in double precision.

Finally, alternative random number generators may be provided to the class. The default random number generator is the standard library `std::mt19937_64`. However, here is an example which uses the 32-bit Mersenne twister random number generator instead:

```
naive_monte_carlo<Real, decltype(g), std::mt19937> mc(g, bounds, (Real) 0.001);
```

Numerical Differentiation

Synopsis

```
#include <boost/math/differentiaton/finite_difference.hpp>


namespace boost { namespace math { namespace differentiation {

    template <class F, class Real>
    Real complex_step_derivative(const F f, Real x);

    template <class F, class Real, size_t order = 6>
    Real finite_difference_derivative(const F f, Real x, Real* error = nullptr);

}}} // namespaces
```

Description

The function `finite_difference_derivative` calculates a finite-difference approximation to the derivative of a function *f* at point *x*. A basic usage is

```
auto f = [](double x) { return std::exp(x); };
double x = 1.7;
double dfdx = finite_difference_derivative(f, x);
```

Finite differencing is complicated, as differentiation is *infinitely* ill-conditioned. In addition, for any function implemented in finite-precision arithmetic, the "true" derivative is *zero* almost everywhere, and undefined at representables. However, some tricks allow for reasonable results to be obtained in many cases.

There are two sources of error from finite differences: One, the truncation error arising from using a finite number of samples to cancel out higher order terms in the Taylor series. The second is the roundoff error involved in evaluating the function. The truncation error goes to zero as $h \to 0$, but the roundoff error becomes unbounded. By balancing these two sources of error, we can choose a value of h that minimizes the maximum total error. For this reason boost's `finite_difference_derivative` does not require the user to input a stepsize. For more details about the theoretical error analysis involved in finite-difference approximations to the derivative, see here.

Despite the effort that has went into choosing a reasonable value of h, the problem is still fundamentally ill-conditioned, and hence an error estimate is essential. It can be queried as follows

```
double error_estimate;
double d = finite_difference_derivative(f, x, &error_estimate);
```

N.B.: Producing an error estimate requires additional function evaluations and as such is slower than simple evaluation of the derivative. It also expands the domain over which the function must be differentiable and requires the function to have two more continuous derivatives. The error estimate is computed under the assumption that *f* is evaluated to 1ULP. This might seem an extreme assumption, but it is the only sensible one, as the routine cannot know the functions rounding error. If the function cannot be evaluated with very great accuracy, Lanczos's smoothing differentiation is recommended as an alternative.

The default order of accuracy is 6, which reflects that fact that people tend to be interested in functions with many continuous derivatives. If your function does not have 7 continuous derivatives, is may be of interest to use a lower order method, which can be achieved via (say)

```
double d = finite_difference_derivative<decltype(f), Real, 2>(f, x);
```

This requests a second-order accurate derivative be computed.

It is emphatically *not* the case that higher order methods always give higher accuracy for smooth functions. Higher order methods require more addition of positive and negative terms, which can lead to catastrophic cancellation. A function which is very good at making a mockery of finite-difference differentiation is $\exp(x)/(\cos(x)^3 + \sin(x)^3)$. Differentiating this function by `finite_difference_derivative` in double precision at $x=5.5$ gives zero correct digits at order 4, 6, and 8, but recovers 5 correct digits at order 2. These are dangerous waters; use the error estimates to tread carefully.

For a finite-difference method of order k, the error is $C\,\varepsilon^{k/k+1}$. In the limit $k \to \infty$, we see that the error tends to ε, recovering the full precision for the type. However, this ignores the fact that higher-order methods require subtracting more nearly-equal (perhaps noisy) terms, so the constant C grows with k. Since C grows quickly and $\varepsilon^{k/k+1}$ approaches ε slowly, we can see there is a compromise between high-order accuracy and conditioning of the difference quotient. In practice we have found that $k=6$ seems to be a good compromise between the two (and have made this the default), but users are encouraged to examine the error estimates to choose an optimal order of accuracy for the given problem.

Table 124. Cost of Finite-Difference Numerical Differentiation

Order of Accuracy	Function Evaluations	Error	Continuous Derivatives Required for Error Estimate to Hold	Additional Function Evaluations to Produce Error Estimates
1	2	$\varepsilon^{1/2}$	2	1
2	2	$\varepsilon^{2/3}$	3	2
4	4	$\varepsilon^{4/5}$	5	2
6	6	$\varepsilon^{6/7}$	7	2
8	8	$\varepsilon^{8/9}$	9	2

Given all the caveats which must be kept in mind for successful use of finite-difference differentiation, it is reasonable to try to avoid it if possible. Boost provides two possibilities: The Chebyshev transform (see here) and the complex step derivative. If your function is the restriction to the real line of a holomorphic function which takes real values at real argument, then the **complex step derivative** can be used. The idea is very simple: Since f is complex-differentiable, $f(x+\ h) = f(x) + \ hf'(x) - h^2f''(x) + \ (h^3)$. As long as $f(x) \in\ $, then $f'(x) = \Im f(x+\ h)/h + \ (h^2)$. This method requires a single complex function evaluation and is not subject to the catastrophic subtractive cancellation that plagues finite-difference calculations.

An example usage:

```
double x = 7.2;
double e_prime = complex_step_derivative(std::exp<std::complex<double>>, x);
```

References:

- Squire, William, and George Trapp. *Using complex variables to estimate derivatives of real functions.* Siam Review 40.1 (1998): 110-112.
- Fornberg, Bengt. *Generation of finite difference formulas on arbitrarily spaced grids.* Mathematics of computation 51.184 (1988): 699-706.
- Corless, Robert M., and Nicolas Fillion. *A graduate introduction to numerical methods.* AMC 10 (2013): 12.

Automatic Differentiation

Synopsis

```
#include <boost/math/differentiation/autodiff.hpp>

namespace boost {
namespace math {
namespace differentiation {

// Function returning a single variable of differentiation. Recommended: Use auto for type.
template <typename RealType, size_t Order, size_t... Orders>
autodiff_fvar<RealType, Order, Orders...> make_fvar(RealType const& ca);

// Function returning multiple independent variables of differentiation in a std::tuple.
template<typename RealType, size_t... Orders, typename... RealTypes>
auto make_ftuple(RealTypes const&... ca);

// Type of combined autodiff types. Recommended: Use auto for return type (C++14).
template <typename RealType, typename... RealTypes>
using promote = typename detail::promote_args_n<RealType, RealTypes...>::type;

namespace detail {

// Single autodiff variable. Use make_fvar() or make_ftuple() to instantiate.
template <typename RealType, size_t Order>
class fvar {
 public:
  // Query return value of function to get the derivatives.
  template <typename... Orders>
  get_type_at<RealType, sizeof...(Orders) - 1> derivative(Orders... orders) const;

  // All of the arithmetic and comparison operators are overloaded.
  template <typename RealType2, size_t Order2>
  fvar& operator+=(fvar<RealType2, Order2> const&);

  fvar& operator+=(root_type const&);

  // ...
};

// Standard math functions are overloaded and called via argument-dependent lookup (ADL).
template <typename RealType, size_t Order>
fvar<RealType, Order> floor(fvar<RealType, Order> const&);

template <typename RealType, size_t Order>
fvar<RealType, Order> exp(fvar<RealType, Order> const&);

// ...
} // namespace detail

} // namespace differentiation
} // namespace math
} // namespace boost
```

Description

Autodiff is a header-only C++ library that facilitates the automatic differentiation (forward mode) of mathematical functions of single and multiple variables.

This implementation is based upon the Taylor series expansion of an analytic function *f* at the point x_0:

$$\begin{aligned} f(x_0+\varepsilon) &= f(x_0) + f'(x_0)\varepsilon + \frac{f''(x_0)}{2!}\varepsilon^2 + \frac{f'''(x_0)}{3!}\varepsilon^3 + \cdots \\ &= \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\varepsilon^n + O\left(\varepsilon^{N+1}\right). \end{aligned}$$

The essential idea of autodiff is the substitution of numbers with polynomials in the evaluation of $f(x_0)$. By substituting the number x_0 with the first-order polynomial $x_0+\varepsilon$, and using the same algorithm to compute $f(x_0+\varepsilon)$, the resulting polynomial in ε contains the function's derivatives $f'(x_0)$, $f''(x_0)$, $f'''(x_0)$, ... within the coefficients. Each coefficient is equal to the derivative of its respective order, divided by the factorial of the order.

In greater detail, assume one is interested in calculating the first *N* derivatives of *f* at x_0. Without loss of precision to the calculation of the derivatives, all terms $O(\varepsilon^{N+1})$ that include powers of ε greater than *N* can be discarded. (This is due to the fact that each term in a polynomial depends only upon equal and lower-order terms under arithmetic operations.) Under these truncation rules, *f* provides a polynomial-to-polynomial transformation:

$$f \qquad : \qquad x_0+\varepsilon \qquad \mapsto \qquad \sum_{n=0}^{N} y_n\varepsilon^n = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\varepsilon^n.$$

C++'s ability to overload operators and functions allows for the creation of a class `fvar` (forward-mode autodiff variable) that represents polynomials in ε. Thus the same algorithm *f* that calculates the numeric value of $y_0=f(x_0)$, when written to accept and return variables of a generic (template) type, is also used to calculate the polynomial $\Sigma_n y_n\varepsilon^n=f(x_0+\varepsilon)$. The derivatives $f^{(n)}(x_0)$ are then found from the product of the respective factorial *n!* and coefficient y_n:

$$\frac{d^n f}{dx^n}(x_0) = n!y_n.$$

Examples

Example 1: Single-variable derivatives

Calculate derivatives of $f(x)=x^4$ at x=2.

In this example, `make_fvar<double, Order>(2.0)` instantiates the polynomial 2+ε. The `Order=5` means that enough space is allocated (on the stack) to hold a polynomial of up to degree 5 during the proceeding computation.

Internally, this is modeled by a `std::array<double,6>` whose elements `{2, 1, 0, 0, 0, 0}` correspond to the 6 coefficients of the polynomial upon initialization. Its fourth power, at the end of the computation, is a polynomial with coefficients `y = {16, 32, 24, 8, 1, 0}`. The derivatives are obtained using the formula $f^{(n)}(2)=n!*y[n]$.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>

template <typename T>
T fourth_power(T const& x) {
  T x4 = x * x;  // retval in operator*() uses x4's memory via NRVO.
  x4 *= x4;      // No copies of x4 are made within operator*=() even when squaring.
  return x4;     // x4 uses y's memory in main() via NRVO.
}

int main() {
  using namespace boost::math::differentiation;

  constexpr unsigned Order = 5;                  // Highest order derivative to be calculated.
  auto const x = make_fvar<double, Order>(2.0);  // Find derivatives at x=2.
  auto const y = fourth_power(x);
  for (unsigned i = 0; i <= Order; ++i)
    std::cout << "y.derivative(" << i << ") = " << y.derivative(i) << std::endl;
  return 0;
}
/*
Output:
y.derivative(0) = 16
y.derivative(1) = 32
y.derivative(2) = 48
y.derivative(3) = 48
y.derivative(4) = 24
y.derivative(5) = 0
*/
```

The above calculates

$$\begin{aligned}
\texttt{y.derivative}(0) &= & f(2) &= & x^4\big|_{x=2} &= 16\\
\texttt{y.derivative}(1) &= & f'(2) &= & 4\cdot x^3\big|_{x=2} &= 32\\
\texttt{y.derivative}(2) &= & f''(2) &= & 4\cdot 3\cdot x^2\big|_{x=2} &= 48\\
\texttt{y.derivative}(3) &= & f'''(2) &= & 4\cdot 3\cdot 2\cdot x\big|_{x=2} &= 48\\
\texttt{y.derivative}(4) &= & f^{(4)}(2) &= & 4\cdot 3\cdot 2\cdot 1 &= 24\\
\texttt{y.derivative}(5) &= & f^{(5)}(2) &= & 0 &
\end{aligned}$$

Example 2: Multi-variable mixed partial derivatives with multi-precision data type

Calculate $\frac{\partial^{12}f}{\partial w^3\partial x^2\partial y^4\partial z^3}(11,12,13,14)$ with a precision of about 50 decimal digits, where $f(w,x,y,z)=\exp\left(w\sin\left(\frac{x\log(y)}{z}\right)+\sqrt{\frac{wz}{xy}}\right)+\frac{w^2}{\tan(z)}$.

In this example, `make_ftuple<float50, Nw, Nx, Ny, Nz>(11, 12, 13, 14)` returns a `std::tuple` of 4 independent `fvar` variables, with values of 11, 12, 13, and 14, for which the maximum order derivative to be calculated for each are 3, 2, 4, 3, respectively. The order of the variables is important, as it is the same order used when calling `v.derivative(Nw, Nx, Ny, Nz)` in the example below.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <boost/multiprecision/cpp_bin_float.hpp>
#include <iostream>

using namespace boost::math::differentiation;

template <typename W, typename X, typename Y, typename Z>
promote<W, X, Y, Z> f(const W& w, const X& x, const Y& y, const Z& z) {
  using namespace std;
  return exp(w * sin(x * log(y) / z) + sqrt(w * z / (x * y))) + w * w / tan(z);
}

int main() {
  using float50 = boost::multiprecision::cpp_bin_float_50;

  constexpr unsigned Nw = 3;  // Max order of derivative to calculate for w
  constexpr unsigned Nx = 2;  // Max order of derivative to calculate for x
  constexpr unsigned Ny = 4;  // Max order of derivative to calculate for y
  constexpr unsigned Nz = 3;  // Max order of derivative to calculate for z
  // Declare 4 independent variables together into a std::tuple.
  auto const variables = make_ftuple<float50, Nw, Nx, Ny, Nz>(11, 12, 13, 14);
  auto const& w = std::get<0>(variables);  // Up to Nw derivatives at w=11
  auto const& x = std::get<1>(variables);  // Up to Nx derivatives at x=12
  auto const& y = std::get<2>(variables);  // Up to Ny derivatives at y=13
  auto const& z = std::get<3>(variables);  // Up to Nz derivatives at z=14
  auto const v = f(w, x, y, z);
  // Calculated from Mathematica symbolic differentiation.
  float50 const answer("1976.319600747797717779881875290418720908121189218755");
  std::cout << std::setprecision(std::numeric_limits<float50>::digits10)
            << "mathematica   : " << answer << '\n'
            << "autodiff      : " << v.derivative(Nw, Nx, Ny, Nz) << '\n'
            << std::setprecision(3)
            << "relative error: " << (v.derivative(Nw, Nx, Ny, Nz) / answer - 1) << '\n';
  return 0;
}
/*
Output:
mathematica   : 1976.3196007477977177798818752904187209081211892188
autodiff      : 1976.3196007477977177798818752904187209081211892188
relative error: 2.67e-50
*/
```

Example 3: Black-Scholes Option Pricing with Greeks Automatically Calculated

Calculate greeks directly from the Black-Scholes pricing function.

Below is the standard Black-Scholes pricing function written as a function template, where the price, volatility (sigma), time to expiration (tau) and interest rate are template parameters. This means that any greek based on these 4 variables can be calculated using autodiff. The below example calculates delta and gamma where the variable of differentiation is only the price. For examples of more exotic greeks, see `example/black_scholes.cpp`.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>

using namespace boost::math::constants;
using namespace boost::math::differentiation;

// Equations and function/variable names are from
// https://en.wikipedia.org/wiki/Greeks_(finance)#Formulas_for_European_option_Greeks

// Standard normal cumulative distribution function
template <typename X>
X Phi(X const& x) {
  return 0.5 * erfc(-one_div_root_two<X>() * x);
}

enum class CP { call, put };

// Assume zero annual dividend yield (q=0).
template <typename Price, typename Sigma, typename Tau, typename Rate>
promote<Price, Sigma, Tau, Rate> black_scholes_option_price(CP cp,
                                                            double K,
                                                            Price const& S,
                                                            Sigma const& sigma,
                                                            Tau const& tau,
                                                            Rate const& r) {
  using namespace std;
  auto const d1 = (log(S / K) + (r + sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
  auto const d2 = (log(S / K) + (r - sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
  switch (cp) {
    case CP::call:
      return S * Phi(d1) - exp(-r * tau) * K * Phi(d2);
    case CP::put:
      return exp(-r * tau) * K * Phi(-d2) - S * Phi(-d1);
  }
}

int main() {
  double const K = 100.0;                    // Strike price.
  auto const S = make_fvar<double, 2>(105);  // Stock price.
  double const sigma = 5;                    // Volatility.
  double const tau = 30.0 / 365;             // Time to expiration in years. (30 days).
  double const r = 1.25 / 100;               // Interest rate.
  auto const call_price = black_scholes_option_price(CP::call, K, S, sigma, tau, r);
  auto const put_price = black_scholes_option_price(CP::put, K, S, sigma, tau, r);

  std::cout << "black-scholes call price = " << call_price.derivative(0) << '\n'
            << "black-scholes put  price = " << put_price.derivative(0) << '\n'
            << "call delta = " << call_price.derivative(1) << '\n'
            << "put  delta = " << put_price.derivative(1) << '\n'
            << "call gamma = " << call_price.derivative(2) << '\n'
            << "put  gamma = " << put_price.derivative(2) << '\n';
  return 0;
}
/*
Output:
black-scholes call price = 56.5136
```

```
black-scholes put  price = 51.4109
call delta = 0.773818
put  delta = -0.226182
call gamma = 0.00199852
put  gamma = 0.00199852
*/
```

Advantages of Automatic Differentiation

The above examples illustrate some of the advantages of using autodiff:

- Elimination of code redundancy. The existence of *N* separate functions to calculate derivatives is a form of code redundancy, with all the liabilities that come with it:
  - Changes to one function require *N* additional changes to other functions. In the 3rd example above, consider how much larger and inter-dependent the above code base would be if a separate function were written for each Greek value.
  - Dependencies upon a derivative function for a different purpose will break when changes are made to the original function. What doesn't need to exist cannot break.
  - Code bloat, reducing conceptual integrity. Control over the evolution of code is easier/safer when the code base is smaller and able to be intuitively grasped.
- Accuracy of derivatives over finite difference methods. Single-iteration finite difference methods always include a Δx free variable that must be carefully chosen for each application. If Δx is too small, then numerical errors become large. If Δx is too large, then mathematical errors become large. With autodiff, there are no free variables to set and the accuracy of the answer is generally superior to finite difference methods even with the best choice of Δx.

Manual

Additional details are in the autodiff manual.

Lanczos Smoothing Derivatives

Synopsis

```
#include <boost/math/differentiation/lanczos_smoothing.hpp>

namespace boost::math::differentiation {

    template <class Real, size_t order=1>
    class discrete_lanczos_derivative {
    public:
        discrete_lanczos_derivative(Real spacing,
                                    size_t n = 18,
                                    size_t approximation_order = 3);

        template<class RandomAccessContainer>
        Real operator()(RandomAccessContainer const & v, size_t i) const;

        template<class RandomAccessContainer>
        void operator()(RandomAccessContainer const & v, RandomAccessContainer & dvdt) const;

        template<class RandomAccessContainer>
        RandomAccessContainer operator()(RandomAccessContainer const & v) const;

        Real get_spacing() const;
    };

} // namespaces
```

Description

The `discrete_lanczos_derivative` class calculates a finite-difference approximation to the derivative of a noisy sequence of equally-spaced values *v*. A basic usage is

```
std::vector<double> v(500);
// fill v with noisy data.
double spacing = 0.001;
using boost::math::differentiation::discrete_lanczos_derivative;
auto lanczos = discrete_lanczos_derivative(spacing);
// Compute dvdt at index 30:
double dvdt30 = lanczos(v, 30);
// Compute derivative of entire vector:
std::vector<double> dvdt = lanczos(v);
```

Noise-suppressing second derivatives can also be computed. The syntax is as follows:

```
std::vector<double> v(500);
// fill v with noisy data.
auto lanczos = lanczos_derivative<double, 2>(spacing);
// evaluate second derivative at a point:
double d2vdt2 = lanczos(v, 18);
// evaluate second derivative of entire vector:
std::vector<double> d2vdt2 = lanczos(v);
```

For memory conscious programmers, you can reuse the memory space for the derivative as follows:

```
std::vector<double> v(500);
std::vector<double> dvdt(500);
// . . . define spacing, create and instance of discrete_lanczos_derivative

// populate dvdt, perhaps in a loop:
lanczos(v, dvdt);
```

If the data has variance σ^2, then the variance of the computed derivative is roughly $\sigma^2 p^3 n^{-3} \Delta t^{-2}$, i.e., it increases cubically with the approximation order *p*, linearly with the data variance, and decreases at the cube of the filter length *n*. In addition, we must not forget the discretization error which is $O(\Delta t^p)$. You can play around with the approximation order *p* and the filter length *n*:

```
size_t n = 12;
size_t p = 2;
auto lanczos = lanczos_derivative(spacing, n, p);
double dvdt = lanczos(v, i);
```

If $p=2n$, then the discrete Lanczos derivative is not smoothing: It reduces to the standard $2n+1$-point finite-difference formula. For $p>2n$, an assertion is hit as the filter is undefined.

In our tests with AWGN, we have found the error decreases monotonically with *n*, as is expected from the theory discussed above. So the choice of *n* is simple: As high as possible given your speed requirements (larger *n* implies a longer filter and hence more compute), balanced against the danger of overfitting and averaging over non-stationarity.

The choice of approximation order *p* for a given *n* is more difficult. If your signal is believed to be a polynomial, it does not make sense to set *p* to larger than the polynomial degree- though it may be sensible to take *p* less than this.

For a sinusoidal signal contaminated with AWGN, we ran a few tests showing that for SNR = 1, p = n/8 gave the best results, for SNR = 10, p = n/7 was the best, and for SNR = 100, p = n/6 was the most reasonable choice. For SNR = 0.1, the method appears to be useless. The user is urged to use these results with caution: they have no theoretical backing and are extrapolated from a single case.

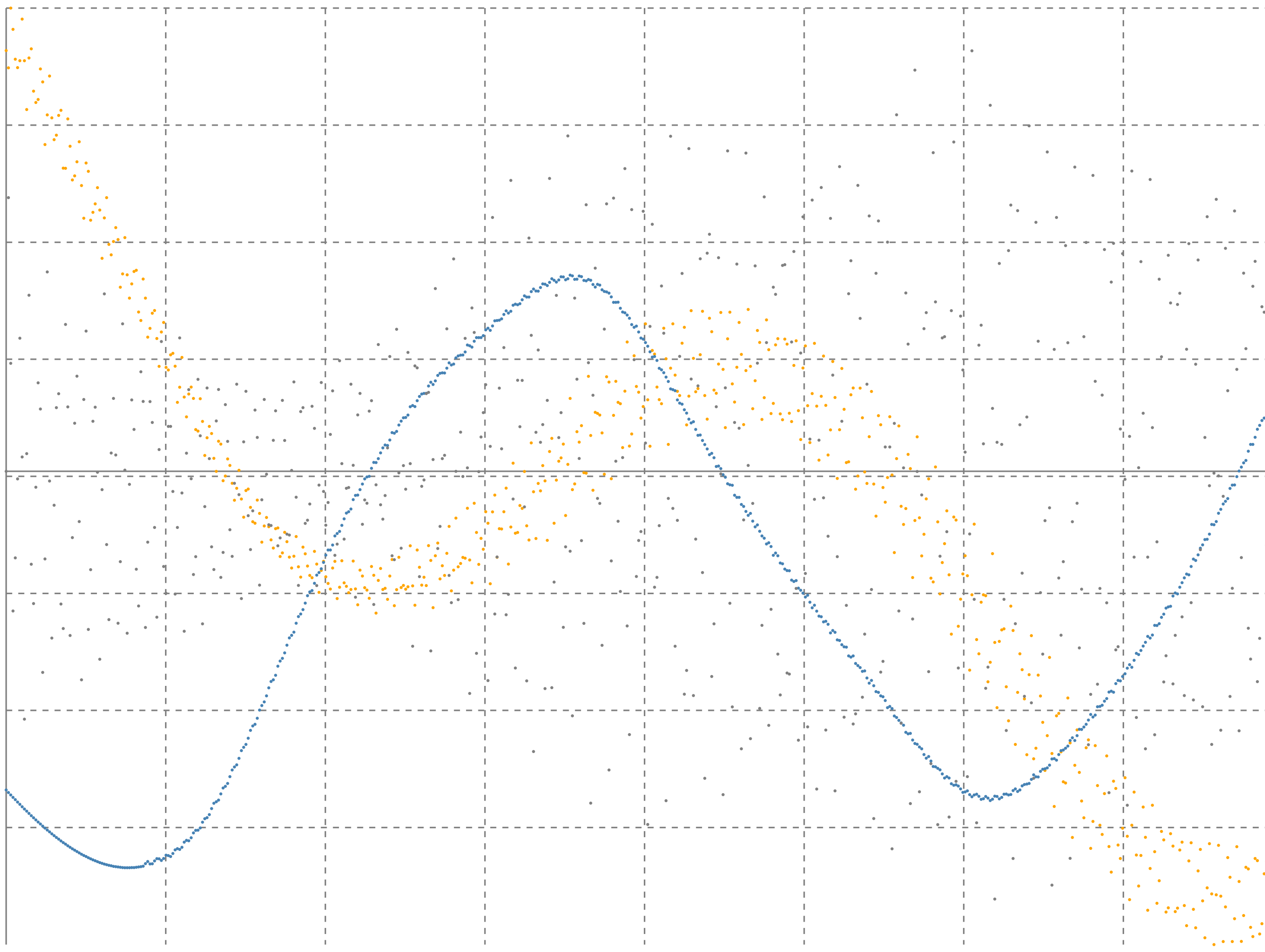
The filters are (regrettably) computed at runtime-the vast number of combinations of approximation order and filter length makes the number of filters that must be stored excessive for compile-time data. The constructor call computes the filters. Since each filter has length $2n+1$ and there are *n* filters, whose element each consist of *p* summands, the complexity of the constructor call is $O(n^2 p)$. This is not cheap-though for most cases small *p* and *n* not too large (< 20) is desired. However, for concreteness, on the author's 2.7GHz Intel laptop CPU, the $n=16$, $p=3$ filter takes 9 microseconds to compute. This is far from negligible, and as such the filters may be used with multiple data, and even shared between threads:

```
std::vector<double> v1(500);
std::vector<double> v2(500);
// fill v1, v2 with noisy data.
auto lanczos = lanczos_derivative(spacing);
std::vector<double> dv1dt = lanczos(v1); // threadsafe
std::vector<double> dv2dt = lanczos(v2); // threadsafe
// need to use a different spacing?
lanczos.reset_spacing(0.02); // not threadsafe
```

The implementation follows McDevitt, 2012, who vastly expanded the ideas of Lanczos to create a very general framework for numerically differentiating noisy equispaced data.

Example

We have extracted some data from the LIGO signal and differentiated it using the $(n, p) = (60, 4)$ Lanczos smoothing derivative, as well as using the $(n, p) = (4, 8)$ (nonsmoothing) derivative.



The original data is in orange, the smoothing derivative in blue, and the non-smoothing standard finite difference formula is in gray. (Each time series has been rescaled to fit in the same graph.) We can see that the smoothing derivative tracks the increase and decrease in the trend well, whereas the standard finite difference formula produces nonsense and amplifies noise.

Caveats

The computation of the filters is ill-conditioned for large *p*. In order to mitigate this problem, we have computed the filters in higher precision and cast the results to the desired type. However, this simply pushes the problem to larger *p*. In practice, this is not a problem, as large *p* corresponds to less powerful denoising, but keep it in mind.

In addition, the `-ffast-math` flag has a very large effect on the speed of these functions. In our benchmarks, they were 50% faster with the flag enabled, which is much larger than the usual performance increases we see by turning on this flag. Hence, if the default performance is not sufficient, this flag is available, though it comes with its own problems.

This requires C++17 `if constexpr`.

References

- Corless, Robert M., and Nicolas Fillion. *A graduate introduction to numerical methods.* AMC 10 (2013): 12.
- Lanczos, Cornelius. *Applied analysis.* Courier Corporation, 1988.

- Timothy J. McDevitt (2012): *Discrete Lanczos derivatives of noisy data*, International Journal of Computer Mathematics, 89:7, 916-931

Complex Number Functions

The following complex number algorithms are the inverses of trigonometric functions currently present in the C++ standard. Equivalents to these functions are part of the C99 standard, and are part of the Technical Report on C++ Library Extensions.

Implementation and Accuracy

Although there are deceptively simple formulae available for all of these functions, a naive implementation that used these formulae would fail catastrophically for some input values. The Boost versions of these functions have been implemented using the methodology described in "Implementing the Complex Arcsine and Arccosine Functions Using Exception Handling" by T. E. Hull Thomas F. Fairgrieve and Ping Tak Peter Tang, ACM Transactions on Mathematical Software, Vol. 23, No. 3, September 1997. This means that the functions are well defined over the entire complex number range, and produce accurate values even at the extremes of that range, where as a naive formula would cause overflow or underflow to occur during the calculation, even though the result is actually a representable value. The maximum theoretical relative error for all of these functions is less than 9.5ε for every machine-representable point in the complex plane. Please refer to comments in the header files themselves and to the above mentioned paper for more information on the implementation methodology.

asin

Header:

```
#include <boost/math/complex/asin.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> asin(const std::complex<T>& z);
```

Effects: returns the inverse sine of the complex number z.

Formula: $\sin^{-1}(z) = -i\log(iz + \sqrt{1 - z^2})$

acos

Header:

```
#include <boost/math/complex/acos.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> acos(const std::complex<T>& z);
```

Effects: returns the inverse cosine of the complex number z.

Formula: $\cos^{-1}(z) = \frac{\pi}{2} + i\log(iz + \sqrt{1-z^2})$

atan

Header:

```
#include <boost/math/complex/atan.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> atan(const std::complex<T>& z);
```

Effects: returns the inverse tangent of the complex number z.

Formula: $\tan^{-1}(z) = \frac{i}{2}(\log(1 - iz) - \log(iz + 1)) = -i \tanh^{-1}(iz)$

asinh

Header:

```
#include <boost/math/complex/asinh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> asinh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic sine of the complex number z.

Formula: $\sinh^{-1}(z) = \log(z + \sqrt{z^2 + 1}) = i \sin^{-1}(-iz)$

acosh

Header:

```
#include <boost/math/complex/acosh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> acosh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic cosine of the complex number z.

Formula: $\cosh^{-1}(z) = \log(z + \sqrt{z-1}\sqrt{z+1}) = \pm i\cos^{-1}(z)$

atanh

Header:

```
#include <boost/math/complex/atanh.hpp>
```

Synopsis:

```
template<class T>
std::complex<T> atanh(const std::complex<T>& z);
```

Effects: returns the inverse hyperbolic tangent of the complex number z.

Formula: $\tanh^{-1}(z) = \frac{1}{2}(\log(1+z) - \log(1-z))$

History

- 2005/12/17: Added support for platforms with no meaningful numeric_limits<>::infinity().
- 2005/12/01: Initial version, added as part of the TR1 library.

Quaternions

Overview

Quaternions are a relative of complex numbers.

Quaternions are in fact part of a small hierarchy of structures built upon the real numbers, which comprise only the set of real numbers (traditionally named ***R***), the set of complex numbers (traditionally named ***C***), the set of quaternions (traditionally named ***H***) and the set of octonions (traditionally named ***O***), which possess interesting mathematical properties (chief among which is the fact that they are *division algebras*, *i.e.* where the following property is true: if y is an element of that algebra and is **not equal to zero**, then $yx = yx'$, where x and x' denote elements of that algebra, implies that $x = x'$). Each member of the hierarchy is a super-set of the former.

One of the most important aspects of quaternions is that they provide an efficient way to parameterize rotations in $\boldsymbol{R}^3$ (the usual three-dimensional space) and $\boldsymbol{R}^4$.

In practical terms, a quaternion is simply a quadruple of real numbers (α,β,γ,δ), which we can write in the form $q = \alpha + \beta i + \gamma j + \delta k$, where i is the same object as for complex numbers, and j and k are distinct objects which play essentially the same kind of role as i.

An addition and a multiplication is defined on the set of quaternions, which generalize their real and complex counterparts. The main novelty here is that **the multiplication is not commutative** (i.e. there are quaternions x and y such that $xy \neq yx$). A good mnemotechnical way of remembering things is by using the formula $i*i = j*j = k*k = -1$.

Quaternions (and their kin) are described in far more details in this other document (with errata and addenda).

Some traditional constructs, such as the exponential, carry over without too much change into the realms of quaternions, but other, such as taking a square root, do not.

Header File

The interface and implementation are both supplied by the header file quaternion.hpp.

Synopsis

```
namespace boost{ namespace math{

template<typename T> class quaternion;
template<>           class quaternion<float>;
template<>           class quaternion<double>;
template<>           class quaternion<long double>;

// operators
template<typename T> quaternion<T> operator + (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator + (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);

template<typename T> quaternion<T> operator - (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator - (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);

template<typename T> quaternion<T> operator * (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator * (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);

template<typename T> quaternion<T> operator / (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator / (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);

template<typename T> quaternion<T> operator + (quaternion<T> const & q);
template<typename T> quaternion<T> operator - (quaternion<T> const & q);

template<typename T> bool operator == (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T> bool operator != (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, quaternion<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (quaternion<T> const & lhs, quaternion<T> const & rhs);

template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits>& operator >> (::std::basic_istream<charT,traits> & is, qua↵
ternion<T> & q);

template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator operator << (::std::basic_os↵
tream<charT,traits> & os, quaternion<T> const & q);

// values
template<typename T> T              real(quaternion<T> const & q);
template<typename T> quaternion<T>  unreal(quaternion<T> const & q);

template<typename T> T              sup(quaternion<T> const & q);
template<typename T> T              l1(quaternion<T> const & q);
template<typename T> T              abs(quaternion<T> const & q);
template<typename T> T              norm(quaternion<T>const  & q);
template<typename T> quaternion<T>  conj(quaternion<T> const & q);

template<typename T> quaternion<T>  math_quaternions.creation_spheric↵
al(T const & rho, T const & theta, T const & phi1, T const & phi2);
template<typename T> quaternion<T>  semipolar(T const & rho, T const & al↵
pha, T const & theta1, T const & theta2);
template<typename T> quaternion<T>  multi↵
polar(T const & rho1, T const & theta1, T const & rho2, T const & theta2);
template<typename T> quaternion<T>  cylindrospherical(T const & t, T const & radi↵
us, T const & longitude, T const & latitude);
template<typename T> quaternion<T>  cylindric↵
al(T const & r, T const & angle, T const & h1, T const & h2);

// transcendentals
template<typename T> quaternion<T>  exp(quaternion<T> const & q);
template<typename T> quaternion<T>  cos(quaternion<T> const & q);
template<typename T> quaternion<T>  sin(quaternion<T> const & q);
template<typename T> quaternion<T>  tan(quaternion<T> const & q);
template<typename T> quaternion<T>  cosh(quaternion<T> const & q);
template<typename T> quaternion<T>  sinh(quaternion<T> const & q);
template<typename T> quaternion<T>  tanh(quaternion<T> const & q);
template<typename T> quaternion<T>  pow(quaternion<T> const & q, int n);

} // namespace math
} // namespace boost
```

Template Class quaternion

```
namespace boost{ namespace math{

template<typename T>
class quaternion
{
public:

   typedef T value_type;

   explicit quaternion(T const & requested_a = T(), T const & requested_b = T(), T const & re↵
quested_c = T(), T const & requested_d = T());
   explicit quaternion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::com↵
plex<T>());
   template<typename X>
   explicit quaternion(quaternion<X> const & a_recopier);

   T                   real() const;
   quaternion<T>       unreal() const;
   T                   R_component_1() const;
   T                   R_component_2() const;
   T                   R_component_3() const;
   T                   R_component_4() const;
   ::std::complex<T>   C_component_1() const;
   ::std::complex<T>   C_component_2() const;

   quaternion<T>&      operator = (quaternion<T> const  & a_affecter);
   template<typename X>
   quaternion<T>&      operator = (quaternion<X> const  & a_affecter);
   quaternion<T>&      operator = (T const  & a_affecter);
   quaternion<T>&      operator = (::std::complex<T> const & a_affecter);

   quaternion<T>&      operator += (T const & rhs);
   quaternion<T>&      operator += (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&      operator += (quaternion<X> const & rhs);

   quaternion<T>&      operator -= (T const & rhs);
   quaternion<T>&      operator -= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&      operator -= (quaternion<X> const & rhs);

   quaternion<T>&      operator *= (T const & rhs);
   quaternion<T>&      operator *= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&      operator *= (quaternion<X> const & rhs);

   quaternion<T>&      operator /= (T const & rhs);
   quaternion<T>&      operator /= (::std::complex<T> const & rhs);
   template<typename X>
   quaternion<T>&      operator /= (quaternion<X> const & rhs);
};

} // namespace math
} // namespace boost
```

Quaternion Specializations

```
namespace boost{ namespace math{

template<>
class quaternion<float>
{
public:
   typedef float value_type;

   explicit quaternion(float const & requested_a = 0.0f, float const & reques↲
ted_b = 0.0f, float const & requested_c = 0.0f, float const & requested_d = 0.0f);
   explicit quaternion(::std::complex<float> const & z0, ::std::com↲
plex<float> const & z1 = ::std::complex<float>());
   explicit quaternion(quaternion<double> const & a_recopier);
   explicit quaternion(quaternion<long double> const & a_recopier);

   float                    real() const;
   quaternion<float>        unreal() const;
   float                    R_component_1() const;
   float                    R_component_2() const;
   float                    R_component_3() const;
   float                    R_component_4() const;
   ::std::complex<float>    C_component_1() const;
   ::std::complex<float>    C_component_2() const;

   quaternion<float>&       operator = (quaternion<float> const  & a_affecter);
   template<typename X>
   quaternion<float>&       operator = (quaternion<X> const  & a_affecter);
   quaternion<float>&       operator = (float const  & a_affecter);
   quaternion<float>&       operator = (::std::complex<float> const & a_affecter);

   quaternion<float>&       operator += (float const & rhs);
   quaternion<float>&       operator += (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&       operator += (quaternion<X> const & rhs);

   quaternion<float>&       operator -= (float const & rhs);
   quaternion<float>&       operator -= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&       operator -= (quaternion<X> const & rhs);

   quaternion<float>&       operator *= (float const & rhs);
   quaternion<float>&       operator *= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&       operator *= (quaternion<X> const & rhs);
   quaternion<float>&     operator /= (float const & rhs);
   quaternion<float>&     operator /= (::std::complex<float> const & rhs);
   template<typename X>
   quaternion<float>&     operator /= (quaternion<X> const & rhs);
};
```

```
template<>
class quaternion<double>
{
public:
   typedef double value_type;

   explicit quaternion(double const & requested_a = 0.0, double const & reques↵
ted_b = 0.0, double const & requested_c = 0.0, double const & requested_d = 0.0);
   explicit quaternion(::std::complex<double> const & z0, ::std::com↵
plex<double> const & z1 = ::std::complex<double>());
   explicit quaternion(quaternion<float> const & a_recopier);
   explicit quaternion(quaternion<long double> const & a_recopier);

   double                  real() const;
   quaternion<double>      unreal() const;
   double                  R_component_1() const;
   double                  R_component_2() const;
   double                  R_component_3() const;
   double                  R_component_4() const;
   ::std::complex<double>  C_component_1() const;
   ::std::complex<double>  C_component_2() const;

   quaternion<double>&     operator = (quaternion<double> const  & a_affecter);
   template<typename X>
   quaternion<double>&     operator = (quaternion<X> const  & a_affecter);
   quaternion<double>&     operator = (double const  & a_affecter);
   quaternion<double>&     operator = (::std::complex<double> const & a_affecter);

   quaternion<double>&     operator += (double const & rhs);
   quaternion<double>&     operator += (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator += (quaternion<X> const & rhs);

   quaternion<double>&     operator -= (double const & rhs);
   quaternion<double>&     operator -= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator -= (quaternion<X> const & rhs);

   quaternion<double>&     operator *= (double const & rhs);
   quaternion<double>&     operator *= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator *= (quaternion<X> const & rhs);
   quaternion<double>&     operator /= (double const & rhs);
   quaternion<double>&     operator /= (::std::complex<double> const & rhs);
   template<typename X>
   quaternion<double>&     operator /= (quaternion<X> const & rhs);
};
```

```
template<>
class quaternion<long double>
{
public:
   typedef long double value_type;

   explicit quaternion(long double const & requested_a = 0.0L, long double const & reques↵
ted_b = 0.0L, long double const & requested_c = 0.0L, long double const & requested_d = 0.0L);
   explicit quaternion(::std::complex<long double> const & z0, ::std::com↵
plex<long double> const & z1 = ::std::complex<long double>());
   explicit quaternion(quaternion<float> const & a_recopier);
   explicit quaternion(quaternion<double> const & a_recopier);

   long double                    real() const;
   quaternion<long double>        unreal() const;
   long double                    R_component_1() const;
   long double                    R_component_2() const;
   long double                    R_component_3() const;
   long double                    R_component_4() const;
   ::std::complex<long double>    C_component_1() const;
   ::std::complex<long double>    C_component_2() const;

   quaternion<long double>&     operator = (quaternion<long double> const  & a_affecter);
   template<typename X>
   quaternion<long double>&     operator = (quaternion<X> const  & a_affecter);
   quaternion<long double>&     operator = (long double const  & a_affecter);
   quaternion<long double>&     operator = (::std::complex<long double> const & a_affecter);

   quaternion<long double>&     operator += (long double const & rhs);
   quaternion<long double>&     operator += (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator += (quaternion<X> const & rhs);

   quaternion<long double>&     operator -= (long double const & rhs);
   quaternion<long double>&     operator -= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator -= (quaternion<X> const & rhs);

   quaternion<long double>&     operator *= (long double const & rhs);
   quaternion<long double>&     operator *= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator *= (quaternion<X> const & rhs);

   quaternion<long double>&     operator /= (long double const & rhs);
   quaternion<long double>&     operator /= (::std::complex<long double> const & rhs);
   template<typename X>
   quaternion<long double>&     operator /= (quaternion<X> const & rhs);
};

} // namespace math
} // namespace boost
```

Quaternion Member Typedefs

value_type

Template version:

```
typedef T value_type;
```

Float specialization version:

```
typedef float value_type;
```

Double specialization version:

```
typedef double value_type;
```

Long double specialization version:

```
typedef long double value_type;
```

These provide easy acces to the type the template is built upon.

Quaternion Member Functions

Constructors

Template version:

```
explicit quaternion(T const & requested_a = T(), T const & requested_b = T(), T const & reques↵
ted_c = T(), T const & requested_d = T());
explicit quaternion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::com↵
plex<T>());
template<typename X>
explicit quaternion(quaternion<X> const & a_recopier);
```

Float specialization version:

```
explicit quaternion(float const & requested_a = 0.0f, float const & reques↵
ted_b = 0.0f, float const & requested_c = 0.0f, float const & requested_d = 0.0f);
explicit quaternion(::std::complex<float> const & z0,::std::com↵
plex<float> const & z1 = ::std::complex<float>());
explicit quaternion(quaternion<double> const & a_recopier);
explicit quaternion(quaternion<long double> const & a_recopier);
```

Double specialization version:

```
explicit quaternion(double const & requested_a = 0.0, double const & reques↵
ted_b = 0.0, double const & requested_c = 0.0, double const & requested_d = 0.0);
explicit quaternion(::std::complex<double> const & z0, ::std::com↵
plex<double> const & z1 = ::std::complex<double>());
explicit quaternion(quaternion<float> const & a_recopier);
explicit quaternion(quaternion<long double> const & a_recopier);
```

Long double specialization version:

```
explicit quaternion(long double const & requested_a = 0.0L, long double const & reques↵
ted_b = 0.0L, long double const & requested_c = 0.0L, long double const & requested_d = 0.0L);
explicit quaternion( ::std::complex<long double> const & z0, ::std::com↵
plex<long double> const & z1 = ::std::complex<long double>());
explicit quaternion(quaternion<float> const & a_recopier);
explicit quaternion(quaternion<double> const & a_recopier);
```

A default constructor is provided for each form, which initializes each component to the default values for their type (i.e. zero for floating numbers). This constructor can also accept one to four base type arguments. A constructor is also provided to build quaternions from one or two complex numbers sharing the same base type. The unspecialized template also sports a templarized copy constructor, while the specialized forms have copy constructors from the other two specializations, which are explicit when a risk of precision loss exists. For the unspecialized form, the base type's constructors must not throw.

Destructors and untemplated copy constructors (from the same type) are provided by the compiler. Converting copy constructors make use of a templated helper function in a "detail" subnamespace.

Other member functions

Real and Unreal Parts

```
T             real() const;
quaternion<T> unreal() const;
```

Like complex number, quaternions do have a meaningful notion of "real part", but unlike them there is no meaningful notion of "imaginary part". Instead there is an "unreal part" which itself is a quaternion, and usually nothing simpler (as opposed to the complex number case). These are returned by the first two functions.

Individual Real Components

```
T R_component_1() const;
T R_component_2() const;
T R_component_3() const;
T R_component_4() const;
```

A quaternion having four real components, these are returned by these four functions. Hence real and R_component_1 return the same value.

Individual Complex Components

```
::std::complex<T>  C_component_1() const;
::std::complex<T>  C_component_2() const;
```

A quaternion likewise has two complex components, and as we have seen above, for any quaternion $q = \alpha + \beta i + \gamma j + \delta k$ we also have $q = (\alpha + \beta i) + (\gamma + \delta i)j$. These functions return them. The real part of `q.C_component_1()` is the same as `q.real()`.

Quaternion Member Operators

Assignment Operators

```
quaternion<T>& operator = (quaternion<T> const & a_affecter);
template<typename X>
quaternion<T>& operator = (quaternion<X> const& a_affecter);
quaternion<T>& operator = (T const& a_affecter);
quaternion<T>& operator = (::std::complex<T> const& a_affecter);
```

These perform the expected assignment, with type modification if necessary (for instance, assigning from a base type will set the real part to that value, and all other components to zero). For the unspecialized form, the base type's assignment operators must not throw.

Addition Operators

```
quaternion<T>& operator += (T const & rhs)
quaternion<T>& operator += (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator += (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)+rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Subtraction Operators

```
quaternion<T>& operator -= (T const & rhs)
quaternion<T>& operator -= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator -= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)-rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Multiplication Operators

```
quaternion<T>& operator *= (T const & rhs)
quaternion<T>& operator *= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator *= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)*rhs` **in this order** (order is important as multiplication is not commutative for quaternions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Division Operators

```
quaternion<T>& operator /= (T const & rhs)
quaternion<T>& operator /= (::std::complex<T> const & rhs);
template<typename X>
quaternion<T>& operator /= (quaternion<X> const & rhs);
```

These perform the mathematical operation `(*this)*inverse_of(rhs)` **in this order** (order is important as multiplication is not commutative for quaternions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

Quaternion Non-Member Operators

Unary Plus

```
template<typename T>
quaternion<T> operator + (quaternion<T> const & q);
```

This unary operator simply returns q.

Unary Minus

```
template<typename T>
quaternion<T> operator - (quaternion<T> const & q);
```

This unary operator returns the opposite of q.

Binary Addition Operators

```
template<typename T> quaternion<T> operator + (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator + (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator + (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) += rhs`.

Binary Subtraction Operators

```
template<typename T> quaternion<T> operator - (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator - (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator - (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) -= rhs`.

Binary Multiplication Operators

```
template<typename T> quaternion<T> operator * (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator * (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator * (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) *= rhs`.

Binary Division Operators

```
template<typename T> quaternion<T> operator / (T const & lhs, quaternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, T const & rhs);
template<typename T> quaternion<T> operator / (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> quaternion<T> operator / (quaternion<T> const & lhs, qua↵
ternion<T> const & rhs);
```

These operators return `quaternion<T>(lhs) /= rhs`. It is of course still an error to divide by zero...

Equality Operators

```
template<typename T> bool  operator == (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool  operator == (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool  operator == (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> bool  operator == (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> bool  operator == (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These return true if and only if the four components of `quaternion<T>(lhs)` are equal to their counterparts in `quaternion<T>(rhs)`. As with any floating-type entity, this is essentially meaningless.

Inequality Operators

```
template<typename T> bool  operator != (T const & lhs, quaternion<T> const & rhs);
template<typename T> bool  operator != (quaternion<T> const & lhs, T const & rhs);
template<typename T> bool  operator != (::std::complex<T> const & lhs, qua↵
ternion<T> const & rhs);
template<typename T> bool  operator != (quaternion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> bool  operator != (quaternion<T> const & lhs, quaternion<T> const & rhs);
```

These return true if and only if `quaternion<T>(lhs) == quaternion<T>(rhs)` is false. As with any floating-type entity, this is essentially meaningless.

Stream Extractor

```
template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits>& operator >> (::std::basic_istream<charT,traits> & is, qua↵
ternion<T> & q);
```

Extracts a quaternion q of one of the following forms (with a, b, c and d of type `T`):

`a (a), (a,b), (a,b,c), (a,b,c,d) (a,(c)), (a,(c,d)), ((a)), ((a),c), ((a),(c)), ((a),(c,d)), ((a,b)), ((a,b),c), ((a,b),(c)), ((a,b),(c,d))`

The input values must be convertible to `T`. If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw ios::failure (27.4.5.3)).

Returns: `is`.

The rationale for the list of accepted formats is that either we have a list of up to four reals, or else we have a couple of complex numbers, and in that case if it formated as a proper complex number, then it should be accepted. Thus potential ambiguities are lifted (for instance (a,b) is (a,b,0,0) and not (a,0,b,0), i.e. it is parsed as a list of two real numbers and not two complex numbers which happen to have imaginary parts equal to zero).

Stream Inserter

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator << (::std::basic_ostream<charT,traits> & os, qua↵
ternion<T> const & q);
```

Inserts the quaternion q onto the stream `os` as if it were implemented as follows:

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits>& operator << (
               ::std::basic_ostream<charT,traits> & os,
               quaternion<T> const & q)
{
   ::std::basic_ostringstream<charT,traits>  s;

   s.flags(os.flags());
   s.imbue(os.getloc());
   s.precision(os.precision());

   s << '(' << q.R_component_1() << ','
            << q.R_component_2() << ','
            << q.R_component_3() << ','
            << q.R_component_4() << ')';

   return os << s.str();
}
```

Quaternion Value Operations

real and unreal

```
template<typename T> T              real(quaternion<T> const & q);
template<typename T> quaternion<T>  unreal(quaternion<T> const & q);
```

These return `q.real()` and `q.unreal()` respectively.

conj

```
template<typename T> quaternion<T>  conj(quaternion<T> const & q);
```

This returns the conjugate of the quaternion.

sup

template<typename T> T sup(quaternion<T> const & q);

This return the sup norm (the greatest among `abs(q.R_component_1())...abs(q.R_component_4())`) of the quaternion.

l1

```
template<typename T> T  l1(quaternion<T> const & q);
```

This return the l1 norm `(abs(q.R_component_1())+...+abs(q.R_component_4()))` of the quaternion.

abs

```
template<typename T> T  abs(quaternion<T> const & q);
```

This return the magnitude (Euclidian norm) of the quaternion.

norm

```
template<typename T> T  norm(quaternion<T>const  & q);
```

This return the (Cayley) norm of the quaternion. The term "norm" might be confusing, as most people associate it with the Euclidian norm (and quadratic functionals). For this version of (the mathematical objects known as) quaternions, the Euclidian norm (also known as magnitude) is the square root of the Cayley norm.

Quaternion Creation Functions

```
template<typename T> quaternion<T> spheric↵
al(T const & rho, T const & theta, T const & phi1, T const & phi2);
template<typename T> quaternion<T> semipolar(T const & rho, T const & al↵
pha, T const & theta1, T const & theta2);
template<typename T> quaternion<T> multi↵
polar(T const & rho1, T const & theta1, T const & rho2, T const & theta2);
template<typename T> quaternion<T> cylindrospherical(T const & t, T const & radius, T const & lon↵
gitude, T const & latitude);
template<typename T> quaternion<T> cylindric↵
al(T const & r, T const & angle, T const & h1, T const & h2);
```

These build quaternions in a way similar to the way polar builds complex numbers, as there is no strict equivalent to polar coordinates for quaternions.

`spherical` is a simple transposition of `polar`, it takes as inputs a (positive) magnitude and a point on the hypersphere, given by three angles. The first of these, `theta` has a natural range of `-pi` to `+pi`, and the other two have natural ranges of `-pi/2` to `+pi/2` (as is the case with the usual spherical coordinates in $\boldsymbol{R}^3$). Due to the many symmetries and periodicities, nothing untoward happens if the magnitude is negative or the angles are outside their natural ranges. The expected degeneracies (a magnitude of zero ignores the angles settings...) do happen however.

`cylindrical` is likewise a simple transposition of the usual cylindrical coordinates in $\boldsymbol{R}^3$, which in turn is another derivative of planar polar coordinates. The first two inputs are the polar coordinates of the first $\boldsymbol{C}$ component of the quaternion. The third and fourth inputs are placed into the third and fourth $\boldsymbol{R}$ components of the quaternion, respectively.

`multipolar` is yet another simple generalization of polar coordinates. This time, both $\boldsymbol{C}$ components of the quaternion are given in polar coordinates.

`cylindrospherical` is specific to quaternions. It is often interesting to consider $\boldsymbol{H}$ as the cartesian product of $\boldsymbol{R}$ by $\boldsymbol{R}^3$ (the quaternionic multiplication as then a special form, as given here). This function therefore builds a quaternion from this representation, with the $\boldsymbol{R}^3$ component given in usual $\boldsymbol{R}^3$ spherical coordinates.

`semipolar` is another generator which is specific to quaternions. It takes as a first input the magnitude of the quaternion, as a second input an angle in the range `0` to `+pi/2` such that magnitudes of the first two $\boldsymbol{C}$ components of the quaternion are the product of the first input and the sine and cosine of this angle, respectively, and finally as third and fourth inputs angles in the range `-pi/2` to `+pi/2` which represent the arguments of the first and second $\boldsymbol{C}$ components of the quaternion, respectively. As usual, nothing untoward happens if what should be magnitudes are negative numbers or angles are out of their natural ranges, as symmetries and periodicities kick in.

In this version of our implementation of quaternions, there is no analogue of the complex value operation `arg` as the situation is somewhat more complicated. Unit quaternions are linked both to rotations in $\boldsymbol{R}^3$ and in $\boldsymbol{R}^4$, and the correspondences are not too complicated, but there is currently a lack of standard (de facto or de jure) matrix library with which the conversions could work. This should be remedied in a further revision. In the mean time, an example of how this could be done is presented here for $\boldsymbol{R}^3$, and here for $\boldsymbol{R}^4$ (example test file).

Quaternion Transcendentals

There is no `log` or `sqrt` provided for quaternions in this implementation, and `pow` is likewise restricted to integral powers of the exponent. There are several reasons to this: on the one hand, the equivalent of analytic continuation for quaternions ("branch cuts") remains to be investigated thoroughly (by me, at any rate...), and we wish to avoid the nonsense introduced in the standard by exponentiations of complexes by complexes (which is well defined, but not in the standard...). Talking of nonsense, saying that `pow(0,0)` is "implementation defined" is just plain brain-dead...

We do, however provide several transcendentals, chief among which is the exponential. This author claims the complete proof of the "closed formula" as his own, as well as its independant invention (there are claims to prior invention of the formula, such as one by Professor Shoemake, and it is possible that the formula had been known a couple of centuries back, but in absence of bibliographical reference, the matter is pending, awaiting further investigation; on the other hand, the definition and existence of the exponential on the quaternions, is of course a fact known for a very long time). Basically, any converging power series with real coefficients which allows for a closed formula in ***C*** can be transposed to ***H***. More transcendentals of this type could be added in a further revision upon request. It should be noted that it is these functions which force the dependency upon the boost/math/special_functions/sinc.hpp and the boost/math/special_functions/sinhc.hpp headers.

exp

```
template<typename T> quaternion<T> exp(quaternion<T> const & q);
```

Computes the exponential of the quaternion.

cos

```
template<typename T> quaternion<T>  cos(quaternion<T> const & q);
```

Computes the cosine of the quaternion

sin

```
template<typename T> quaternion<T>  sin(quaternion<T> const & q);
```

Computes the sine of the quaternion.

tan

```
template<typename T> quaternion<T>  tan(quaternion<T> const & q);
```

Computes the tangent of the quaternion.

cosh

```
template<typename T> quaternion<T>  cosh(quaternion<T> const & q);
```

Computes the hyperbolic cosine of the quaternion.

sinh

```
template<typename T> quaternion<T>  sinh(quaternion<T> const & q);
```

Computes the hyperbolic sine of the quaternion.

tanh

```
template<typename T> quaternion<T>  tanh(quaternion<T> const & q);
```

Computes the hyperbolic tangent of the quaternion.

pow

```
template<typename T> quaternion<T>  pow(quaternion<T> const & q, int n);
```

Computes the n-th power of the quaternion q.

Test Program

The quaternion_test.cpp test program tests quaternions specializations for float, double and long double (sample output, with message output enabled).

If you define the symbol TEST_VERBOSE, you will get additional output (verbose output); this will only be helpfull if you enable message output at the same time, of course (by uncommenting the relevant line in the test or by adding `--log_level=messages` to your command line,...). In that case, and if you are running interactively, you may in addition define the symbol BOOST_INTERACTIVE_TEST_INPUT_ITERATOR to interactively test the input operator with input of your choice from the standard input (instead of hard-coding it in the test).

The Quaternionic Exponential

Please refer to the following PDF's:

- The Quaternionic Exponential (and beyond)
- The Quaternionic Exponential (and beyond) ERRATA & ADDENDA

Acknowledgements

The mathematical text has been typeset with Nisus Writer. Jens Maurer has helped with portability and standard adherence, and was the Review Manager for this library. More acknowledgements in the History section. Thank you to all who contributed to the discution about this library.

History

- 1.5.9 - 13/5/2013: Incorporated into Boost.Math.
- 1.5.8 - 17/12/2005: Converted documentation to Quickbook Format.
- 1.5.7 - 24/02/2003: transitionned to the unit test framework; <boost/config.hpp> now included by the library header (rather than the test files).
- 1.5.6 - 15/10/2002: Gcc2.95.x and stlport on linux compatibility by Alkis Evlogimenos (alkis@routescience.com).
- 1.5.5 - 27/09/2002: Microsoft VCPP 7 compatibility, by Michael Stevens (michael@acfr.usyd.edu.au); requires the /Za compiler option.
- 1.5.4 - 19/09/2002: fixed problem with multiple inclusion (in different translation units); attempt at an improved compatibility with Microsoft compilers, by Michael Stevens (michael@acfr.usyd.edu.au) and Fredrik Blomqvist; other compatibility fixes.
- 1.5.3 - 01/02/2002: bugfix and Gcc 2.95.3 compatibility by Douglas Gregor (gregod@cs.rpi.edu).
- 1.5.2 - 07/07/2001: introduced namespace math.
- 1.5.1 - 07/06/2001: (end of Boost review) now includes <boost/math/special_functions/sinc.hpp> and <boost/math/special_functions/sinhc.hpp> instead of <boost/special_functions.hpp>; corrected bug in sin (Daryle Walker); removed check for self-assignment (Gary Powel); made converting functions explicit (Gary Powel); added overflow guards for division operators and abs (Peter Schmitteckert); added sup and l1; used Vesa Karvonen's CPP metaprograming technique to simplify code.
- 1.5.0 - 26/03/2001: boostification, inlining of all operators except input, output and pow, fixed exception safety of some members (template version) and output operator, added spherical, semipolar, multipolar, cylindrospherical and cylindrical.
- 1.4.0 - 09/01/2001: added tan and tanh.
- 1.3.1 - 08/01/2001: cosmetic fixes.
- 1.3.0 - 12/07/2000: pow now uses Maarten Hilferink's (mhilferink@tip.nl) algorithm.
- 1.2.0 - 25/05/2000: fixed the division operators and output; changed many signatures.
- 1.1.0 - 23/05/2000: changed sinc into sinc_pi; added sin, cos, sinh, cosh.
- 1.0.0 - 10/08/1999: first public version.

To Do

- Improve testing.
- Rewrite input operatore using Spirit (creates a dependency).
- Put in place an Expression Template mechanism (perhaps borrowing from uBlas).
- Use uBlas for the link with rotations (and move from the example implementation to an efficient one).

Octonions

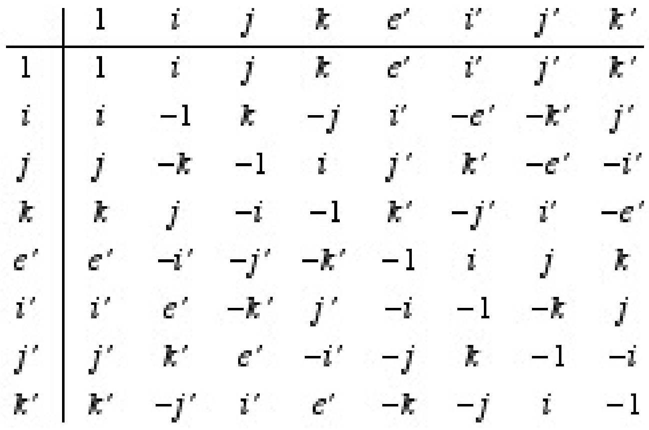
Overview

Octonions, like quaternions, are a relative of complex numbers.

Octonions see some use in theoretical physics.

In practical terms, an octonion is simply an octuple of real numbers (α,β,γ,δ,ε,ζ,η,θ), which we can write in the form $o = \alpha + \beta i + \gamma j + \delta k + \varepsilon e' + \zeta i' + \eta j' + \theta k'$, where i, j and k are the same objects as for quaternions, and e', i', j' and k' are distinct objects which play essentially the same kind of role as i (or j or k).

Addition and a multiplication is defined on the set of octonions, which generalize their quaternionic counterparts. The main novelty this time is that **the multiplication is not only not commutative, is now not even associative** (i.e. there are octonions x, y and z such that $x(yz) \neq (xy)z$). A way of remembering things is by using the following multiplication table:



	1	i	j	k	e'	i'	j'	k'
1	1	i	j	k	e'	i'	j'	k'
i	i	-1	k	$-j$	i'	$-e'$	$-k'$	j'
j	j	$-k$	-1	i	j'	k'	$-e'$	$-i'$
k	k	j	$-i$	-1	k'	$-j'$	i'	$-e'$
e'	e'	$-i'$	$-j'$	$-k'$	-1	i	j	k
i'	i'	e'	$-k'$	j'	$-i$	-1	$-k$	j
j'	j'	k'	e'	$-i'$	$-j$	k	-1	$-i$
k'	k'	$-j'$	i'	e'	$-k$	$-j$	i	-1

Octonions (and their kin) are described in far more details in this other document (with errata and addenda).

Some traditional constructs, such as the exponential, carry over without too much change into the realms of octonions, but other, such as taking a square root, do not (the fact that the exponential has a closed form is a result of the author, but the fact that the exponential exists at all for octonions is known since quite a long time ago).

Header File

The interface and implementation are both supplied by the header file octonion.hpp.

Synopsis

```
namespace boost{ namespace math{

template<typename T> class octonion;
template<>           class octonion<float>;
template<>           class octonion<double>;
template<>           class octonion<long double>;

// operators

template<typename T> octonion<T> operator + (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator + (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator + (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator - (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator - (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator - (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator * (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator * (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator * (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator / (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator / (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator / (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> octonion<T> operator + (octonion<T> const & o);
template<typename T> octonion<T> operator - (octonion<T> const & o);
template<typename T> bool operator == (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T> bool operator != (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, octonion<T> const & rhs);

template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits> & operator >> (::std::basic_istream<charT,traits> & is, oc↵
tonion<T> & o);

template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << (::std::basic_ostream<charT,traits> & os, oc↵
tonion<T> const & o);

// values

template<typename T> T           real(octonion<T> const & o);
template<typename T> octonion<T> unreal(octonion<T> const & o);

template<typename T> T           sup(octonion<T> const & o);
template<typename T> T           l1(octonion<T>const & o);
template<typename T> T           abs(octonion<T> const & o);
template<typename T> T           norm(octonion<T>const  & o);
template<typename T> octonion<T> conj(octonion<T> const & o);

template<typename T> octonion<T> spheric↵
al(T const & rho, T const & theta, T const & phi1, T const & phi2, T const & phi3, T const & phi4, T const & phi5, T const & phi6);
template<typename T> octonion<T> multi↵
polar(T const & rho1, T const & theta1, T const & rho2, T const & theta2, T const & rho3, T const & theta3, T const & rho4, T const & theta4);
template<typename T> octonion<T> cylindric↵
al(T const & r, T const & angle, T const & h1, T const & h2, T const & h3, T const & h4, T const & h5, T const & h6);

// transcendentals

template<typename T> octonion<T> exp(octonion<T> const & o);
template<typename T> octonion<T> cos(octonion<T> const & o);
template<typename T> octonion<T> sin(octonion<T> const & o);
template<typename T> octonion<T> tan(octonion<T> const & o);
template<typename T> octonion<T> cosh(octonion<T> const & o);
template<typename T> octonion<T> sinh(octonion<T> const & o);
template<typename T> octonion<T> tanh(octonion<T> const & o);

template<typename T> octonion<T> pow(octonion<T> const & o, int n);

}  } // namespaces
```

Template Class octonion

```
namespace boost{ namespace math {

template<typename T>
class octonion
{
public:
   typedef T value_type;

   explicit  octonion(T const & requested_a = T(), T const & requested_b = T(), T const & re↵
quested_c = T(), T const & requested_d = T(), T const & requested_e = T(), T const & reques↵
ted_f = T(), T const & requested_g = T(), T const & requested_h = T());
   explicit  octonion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::com↵
plex<T>(), ::std::complex<T> const & z2 = ::std::complex<T>(), ::std::com↵
plex<T> const & z3 = ::std::complex<T>());
   explicit  octonion(::boost::math::quaternion<T> const & q0, ::boost::math::qua↵
ternion<T> const & q1 = ::boost::math::quaternion<T>());
   template<typename X>
   explicit  octonion(octonion<X> const & a_recopier);

   T                              real() const;
   octonion<T>                    unreal() const;

   T                              R_component_1() const;
   T                              R_component_2() const;
   T                              R_component_3() const;
   T                              R_component_4() const;
   T                              R_component_5() const;
   T                              R_component_6() const;
   T                              R_component_7() const;
   T                              R_component_8() const;

   ::std::complex<T>              C_component_1() const;
   ::std::complex<T>              C_component_2() const;
   ::std::complex<T>              C_component_3() const;
   ::std::complex<T>              C_component_4() const;

   ::boost::math::quaternion<T>   H_component_1() const;
   ::boost::math::quaternion<T>   H_component_2() const;

   octonion<T> & operator = (octonion<T> const  & a_affecter);
   template<typename X>
   octonion<T> & operator = (octonion<X> const  & a_affecter);
   octonion<T> & operator = (T const  & a_affecter);
   octonion<T> & operator = (::std::complex<T> const & a_affecter);
   octonion<T> & operator = (::boost::math::quaternion<T> const & a_affecter);

   octonion<T> & operator += (T const & rhs);
   octonion<T> & operator += (::std::complex<T> const & rhs);
   octonion<T> & operator += (::boost::math::quaternion<T> const & rhs);
   template<typename X>
   octonion<T> & operator += (octonion<X> const & rhs);

   octonion<T> & operator -= (T const & rhs);
   octonion<T> & operator -= (::std::complex<T> const & rhs);
   octonion<T> & operator -= (::boost::math::quaternion<T> const & rhs);
   template<typename X>
   octonion<T> & operator -= (octonion<X> const & rhs);

   octonion<T> & operator *= (T const & rhs);
   octonion<T> & operator *= (::std::complex<T> const & rhs);
```

```
    octonion<T> & operator *= (::boost::math::quaternion<T> const & rhs);
    template<typename X>
    octonion<T> & operator *= (octonion<X> const & rhs);

    octonion<T> & operator /= (T const & rhs);
    octonion<T> & operator /= (::std::complex<T> const & rhs);
    octonion<T> & operator /= (::boost::math::quaternion<T> const & rhs);
    template<typename X>
    octonion<T> & operator /= (octonion<X> const & rhs);
};

} } // namespaces
```

Octonion Specializations

```
namespace boost{ namespace math{

template<>
class octonion<float>
{
public:
   typedef float value_type;

   explicit  octonion(float const & requested_a = 0.0f, float const & reques↵
ted_b = 0.0f, float const & requested_c = 0.0f, float const & reques↵
ted_d = 0.0f, float const & requested_e = 0.0f, float const & reques↵
ted_f = 0.0f, float const & requested_g = 0.0f, float const & requested_h = 0.0f);
   explicit  octonion(::std::complex<float> const & z0, ::std::com↵
plex<float> const & z1 = ::std::complex<float>(), ::std::complex<float> const & z2 = ::std::com↵
plex<float>(), ::std::complex<float> const & z3 = ::std::complex<float>());
   explicit  octonion(::boost::math::quaternion<float> const & q0, ::boost::math::qua↵
ternion<float> const & q1 = ::boost::math::quaternion<float>());
   explicit  octonion(octonion<double> const & a_recopier);
   explicit  octonion(octonion<long double> const & a_recopier);

   float                                 real() const;
   octonion<float>                       unreal() const;

   float                                 R_component_1() const;
   float                                 R_component_2() const;
   float                                 R_component_3() const;
   float                                 R_component_4() const;
   float                                 R_component_5() const;
   float                                 R_component_6() const;
   float                                 R_component_7() const;
   float                                 R_component_8() const;

   ::std::complex<float>                 C_component_1() const;
   ::std::complex<float>                 C_component_2() const;
   ::std::complex<float>                 C_component_3() const;
   ::std::complex<float>                 C_component_4() const;

   ::boost::math::quaternion<float>      H_component_1() const;
   ::boost::math::quaternion<float>      H_component_2() const;

   octonion<float> & operator = (octonion<float> const  & a_affecter);
   template<typename X>
   octonion<float> & operator = (octonion<X> const  & a_affecter);
   octonion<float> & operator = (float const  & a_affecter);
   octonion<float> & operator = (::std::complex<float> const & a_affecter);
   octonion<float> & operator = (::boost::math::quaternion<float> const & a_affecter);

   octonion<float> & operator += (float const & rhs);
   octonion<float> & operator += (::std::complex<float> const & rhs);
   octonion<float> & operator += (::boost::math::quaternion<float> const & rhs);
   template<typename X>
   octonion<float> & operator += (octonion<X> const & rhs);

   octonion<float> & operator -= (float const & rhs);
   octonion<float> & operator -= (::std::complex<float> const & rhs);
   octonion<float> & operator -= (::boost::math::quaternion<float> const & rhs);
   template<typename X>
   octonion<float> & operator -= (octonion<X> const & rhs);

   octonion<float> & operator *= (float const & rhs);
```

```
    octonion<float> & operator *= (::std::complex<float> const & rhs);
    octonion<float> & operator *= (::boost::math::quaternion<float> const & rhs);
    template<typename X>
    octonion<float> & operator *= (octonion<X> const & rhs);

    octonion<float> & operator /= (float const & rhs);
    octonion<float> & operator /= (::std::complex<float> const & rhs);
    octonion<float> & operator /= (::boost::math::quaternion<float> const & rhs);
    template<typename X>
    octonion<float> & operator /= (octonion<X> const & rhs);
};
```

```
template<>
class octonion<double>
{
public:
    typedef double value_type;

    explicit  octonion(double const & requested_a = 0.0, double const & reques↵
ted_b = 0.0, double const & requested_c = 0.0, double const & reques↵
ted_d = 0.0, double const & requested_e = 0.0, double const & reques↵
ted_f = 0.0, double const & requested_g = 0.0, double const & requested_h = 0.0);
    explicit  octonion(::std::complex<double> const & z0, ::std::com↵
plex<double> const & z1 = ::std::complex<double>(), ::std::com↵
plex<double> const & z2 = ::std::complex<double>(), ::std::com↵
plex<double> const & z3 = ::std::complex<double>());
    explicit  octonion(::boost::math::quaternion<double> const & q0, ::boost::math::qua↵
ternion<double> const & q1 = ::boost::math::quaternion<double>());
    explicit  octonion(octonion<float> const & a_recopier);
    explicit  octonion(octonion<long double> const & a_recopier);

    double                             real() const;
    octonion<double>                   unreal() const;

    double                             R_component_1() const;
    double                             R_component_2() const;
    double                             R_component_3() const;
    double                             R_component_4() const;
    double                             R_component_5() const;
    double                             R_component_6() const;
    double                             R_component_7() const;
    double                             R_component_8() const;

    ::std::complex<double>             C_component_1() const;
    ::std::complex<double>             C_component_2() const;
    ::std::complex<double>             C_component_3() const;
    ::std::complex<double>             C_component_4() const;

    ::boost::math::quaternion<double>  H_component_1() const;
    ::boost::math::quaternion<double>  H_component_2() const;

    octonion<double> & operator = (octonion<double> const  & a_affecter);
    template<typename X>
    octonion<double> & operator = (octonion<X> const  & a_affecter);
    octonion<double> & operator = (double const  & a_affecter);
    octonion<double> & operator = (::std::complex<double> const & a_affecter);
    octonion<double> & operator = (::boost::math::quaternion<double> const & a_affecter);

    octonion<double> & operator += (double const & rhs);
    octonion<double> & operator += (::std::complex<double> const & rhs);
    octonion<double> & operator += (::boost::math::quaternion<double> const & rhs);
    template<typename X>
    octonion<double> & operator += (octonion<X> const & rhs);

    octonion<double> & operator -= (double const & rhs);
    octonion<double> & operator -= (::std::complex<double> const & rhs);
    octonion<double> & operator -= (::boost::math::quaternion<double> const & rhs);
    template<typename X>
    octonion<double> & operator -= (octonion<X> const & rhs);

    octonion<double> & operator *= (double const & rhs);
    octonion<double> & operator *= (::std::complex<double> const & rhs);
    octonion<double> & operator *= (::boost::math::quaternion<double> const & rhs);
    template<typename X>
    octonion<double> & operator *= (octonion<X> const & rhs);

    octonion<double> & operator /= (double const & rhs);
    octonion<double> & operator /= (::std::complex<double> const & rhs);
    octonion<double> & operator /= (::boost::math::quaternion<double> const & rhs);
    template<typename X>
    octonion<double> & operator /= (octonion<X> const & rhs);
};
```

```
template<>
class octonion<long double>
{
public:
    typedef long double value_type;

    explicit   octonion(long double const & requested_a = 0.0L, long double const & reques↵
ted_b = 0.0L, long double const & requested_c = 0.0L, long double const & reques↵
ted_d = 0.0L, long double const & requested_e = 0.0L, long double const & reques↵
ted_f = 0.0L, long double const & requested_g = 0.0L, long double const & requested_h = 0.0L);
    explicit   octonion( ::std::complex<long double> const & z0, ::std::com↵
plex<long double> const & z1 = ::std::complex<long double>(), ::std::com↵
plex<long double> const & z2 = ::std::complex<long double>(), ::std::com↵
plex<long double> const & z3 = ::std::complex<long double>());
   explicit   octonion( ::boost::math::quaternion<long double> const & q0, ::boost::math::qua↵
ternion<long double> const & z1 = ::boost::math::quaternion<long double>());
    explicit   octonion(octonion<float> const & a_recopier);
    explicit   octonion(octonion<double> const & a_recopier);

    long double                              real() const;
    octonion<long double>                    unreal() const;

    long double                              R_component_1() const;
    long double                              R_component_2() const;
    long double                              R_component_3() const;
    long double                              R_component_4() const;
    long double                              R_component_5() const;
    long double                              R_component_6() const;
    long double                              R_component_7() const;
    long double                              R_component_8() const;

    ::std::complex<long double>              C_component_1() const;
    ::std::complex<long double>              C_component_2() const;
    ::std::complex<long double>              C_component_3() const;
    ::std::complex<long double>              C_component_4() const;

    ::boost::math::quaternion<long double>   H_component_1() const;
    ::boost::math::quaternion<long double>   H_component_2() const;

    octonion<long double> & operator = (octonion<long double> const  & a_affecter);
    template<typename X>
   octonion<long double> & operator = (octonion<X> const  & a_affecter);
   octonion<long double> & operator = (long double const  & a_affecter);
   octonion<long double> & operator = (::std::complex<long double> const & a_affecter);
   octonion<long double> & operator = (::boost::math::quaternion<long double> const & a_af↵
fecter);

   octonion<long double> & operator += (long double const & rhs);
   octonion<long double> & operator += (::std::complex<long double> const & rhs);
   octonion<long double> & operator += (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator += (octonion<X> const & rhs);

   octonion<long double> & operator -= (long double const & rhs);
   octonion<long double> & operator -= (::std::complex<long double> const & rhs);
   octonion<long double> & operator -= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator -= (octonion<X> const & rhs);

   octonion<long double> & operator *= (long double const & rhs);
   octonion<long double> & operator *= (::std::complex<long double> const & rhs);
   octonion<long double> & operator *= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator *= (octonion<X> const & rhs);

   octonion<long double> & operator /= (long double const & rhs);
   octonion<long double> & operator /= (::std::complex<long double> const & rhs);
   octonion<long double> & operator /= (::boost::math::quaternion<long double> const & rhs);
   template<typename X>
   octonion<long double> & operator /= (octonion<X> const & rhs);
};

} } // namespaces
```

Octonion Member Typedefs

value_type

Template version:

```
typedef T value_type;
```

Float specialization version:

```
typedef float value_type;
```

Double specialization version:

```
typedef double value_type;
```

Long double specialization version:

```
typedef long double value_type;
```

These provide easy acces to the type the template is built upon.

Octonion Member Functions

Constructors

Template version:

```
explicit  octonion(T const & requested_a = T(), T const & requested_b = T(), T const & reques↵
ted_c = T(), T const & requested_d = T(), T const & requested_e = T(), T const & reques↵
ted_f = T(), T const & requested_g = T(), T const & requested_h = T());
explicit  octonion(::std::complex<T> const & z0, ::std::complex<T> const & z1 = ::std::com↵
plex<T>(), ::std::complex<T> const & z2 = ::std::complex<T>(), ::std::com↵
plex<T> const & z3 = ::std::complex<T>());
explicit  octonion(::boost::math::quaternion<T> const & q0, ::boost::math::qua↵
ternion<T> const & q1 = ::boost::math::quaternion<T>());
template<typename X>
explicit octonion(octonion<X> const & a_recopier);
```

Float specialization version:

```
explicit  octonion(float const & requested_a = 0.0f, float const & reques↵
ted_b = 0.0f, float const & requested_c = 0.0f, float const & reques↵
ted_d = 0.0f, float const & requested_e = 0.0f, float const & reques↵
ted_f = 0.0f, float const & requested_g = 0.0f, float const & requested_h = 0.0f);
explicit  octonion(::std::complex<float> const & z0, ::std::com↵
plex<float> const & z1 = ::std::complex<float>(), ::std::complex<float> const & z2 = ::std::com↵
plex<float>(), ::std::complex<float> const & z3 = ::std::complex<float>());
explicit  octonion(::boost::math::quaternion<float> const & q0, ::boost::math::qua↵
ternion<float> const & q1 = ::boost::math::quaternion<float>());
explicit  octonion(octonion<double> const & a_recopier);
explicit  octonion(octonion<long double> const & a_recopier);
```

Double specialization version:

```
explicit  octonion(double const & requested_a = 0.0, double const & reques↵
ted_b = 0.0, double const & requested_c = 0.0, double const & reques↵
ted_d = 0.0, double const & requested_e = 0.0, double const & reques↵
ted_f = 0.0, double const & requested_g = 0.0, double const & requested_h = 0.0);
explicit  octonion(::std::complex<double> const & z0, ::std::com↵
plex<double> const & z1 = ::std::complex<double>(), ::std::com↵
plex<double> const & z2 = ::std::complex<double>(), ::std::com↵
plex<double> const & z3 = ::std::complex<double>());
explicit  octonion(::boost::math::quaternion<double> const & q0, ::boost::math::qua↵
ternion<double> const & q1 = ::boost::math::quaternion<double>());
explicit  octonion(octonion<float> const & a_recopier);
explicit  octonion(octonion<long double> const & a_recopier);
```

Long double specialization version:

```
explicit  octonion(long double const & requested_a = 0.0L, long double const & reques↵
ted_b = 0.0L, long double const & requested_c = 0.0L, long double const & reques↵
ted_d = 0.0L, long double const & requested_e = 0.0L, long double const & reques↵
ted_f = 0.0L, long double const & requested_g = 0.0L, long double const & requested_h = 0.0L);
explicit  octonion( ::std::complex<long double> const & z0, ::std::com↵
plex<long double> const & z1 = ::std::complex<long double>(), ::std::com↵
plex<long double> const & z2 = ::std::complex<long double>(), ::std::com↵
plex<long double> const & z3 = ::std::complex<long double>());
explicit  octonion(::boost::math::quaternion<long double> const & q0, ::boost::math::qua↵
ternion<long double> const & q1 = ::boost::math::quaternion<long double>());
explicit  octonion(octonion<float> const & a_recopier);
explicit  octonion(octonion<double> const & a_recopier);
```

A default constructor is provided for each form, which initializes each component to the default values for their type (i.e. zero for floating numbers). This constructor can also accept one to eight base type arguments. A constructor is also provided to build octonions from one to four complex numbers sharing the same base type, and another taking one or two quaternions sharing the same base type. The unspecialized template also sports a templarized copy constructor, while the specialized forms have copy constructors from the other two specializations, which are explicit when a risk of precision loss exists. For the unspecialized form, the base type's constructors must not throw.

Destructors and untemplated copy constructors (from the same type) are provided by the compiler. Converting copy constructors make use of a templated helper function in a "detail" subnamespace.

Other member functions

Real and Unreal Parts

```
T             real()   const;
octonion<T>   unreal() const;
```

Like complex number, octonions do have a meaningful notion of "real part", but unlike them there is no meaningful notion of "imaginary part". Instead there is an "unreal part" which itself is a octonion, and usually nothing simpler (as opposed to the complex number case). These are returned by the first two functions.

Individual Real Components

```
T   R_component_1() const;
T   R_component_2() const;
T   R_component_3() const;
T   R_component_4() const;
T   R_component_5() const;
T   R_component_6() const;
T   R_component_7() const;
T   R_component_8() const;
```

A octonion having eight real components, these are returned by these eight functions. Hence real and R_component_1 return the same value.

Individual Complex Components

```
::std::complex<T> C_component_1() const;
::std::complex<T> C_component_2() const;
::std::complex<T> C_component_3() const;
::std::complex<T> C_component_4() const;
```

A octonion likewise has four complex components. Actually, octonions are indeed a (left) vector field over the complexes, but beware, as for any octonion $o = \alpha + \beta i + \gamma j + \delta k + \varepsilon e' + \zeta i' + \eta j' + \theta k'$ we also have $o = (\alpha + \beta i) + (\gamma +$

`δi)j + (ε + ζi)e' + (η - θi)j'` (note the **minus** sign in the last factor). What the C_component_n functions return, however, are the complexes which could be used to build the octonion using the constructor, and **not** the components of the octonion on the basis `(1, j, e', j')`.

Individual Quaternion Components

```
::boost::math::quaternion<T> H_component_1() const;
::boost::math::quaternion<T> H_component_2() const;
```

Likewise, for any octonion `o = α + βi + γj + δk + εe' + ζi' + ηj' + θk'` we also have `o = (α + βi + γj + δk) + (ε + ζi + ηj - θj)e'`, though there is no meaningful vector-space-like structure based on the quaternions. What the H_component_n functions return are the quaternions which could be used to build the octonion using the constructor.

Octonion Member Operators

Assignment Operators

```
octonion<T> & operator = (octonion<T> const & a_affecter);
template<typename X>
octonion<T> & operator = (octonion<X> const & a_affecter);
octonion<T> & operator = (T const & a_affecter);
octonion<T> & operator = (::std::complex<T> const & a_affecter);
octonion<T> & operator = (::boost::math::quaternion<T> const & a_affecter);
```

These perform the expected assignment, with type modification if necessary (for instance, assigning from a base type will set the real part to that value, and all other components to zero). For the unspecialized form, the base type's assignment operators must not throw.

Other Member Operators

```
octonion<T> & operator += (T const & rhs)
octonion<T> & operator += (::std::complex<T> const & rhs);
octonion<T> & operator += (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator += (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)+rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

```
octonion<T> & operator -= (T const & rhs)
octonion<T> & operator -= (::std::complex<T> const & rhs);
octonion<T> & operator -= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator -= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)-rhs` and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw.

```
octonion<T> & operator *= (T const & rhs)
octonion<T> & operator *= (::std::complex<T> const & rhs);
octonion<T> & operator *= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator *= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)*rhs` in this order (order is important as multiplication is not commutative for octonions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw. Also, for clarity's sake, you should always group the factors in a multiplication by groups of two, as the multiplication is not even associative on the octonions (though there are of course cases where this does not matter, it usually does).

```
octonion<T> & operator /= (T const & rhs)
octonion<T> & operator /= (::std::complex<T> const & rhs);
octonion<T> & operator /= (::boost::math::quaternion<T> const & rhs);
template<typename X>
octonion<T> & operator /= (octonion<X> const & rhs);
```

These perform the mathematical operation `(*this)*inverse_of(rhs)` in this order (order is important as multiplication is not commutative for octonions) and store the result in `*this`. The unspecialized form has exception guards, which the specialized forms do not, so as to insure exception safety. For the unspecialized form, the base type's assignment operators must not throw. As for the multiplication, remember to group any two factors using parenthesis.

Octonion Non-Member Operators

Unary Plus and Minus Operators

```
template<typename T> octonion<T> operator + (octonion<T> const & o);
```

This unary operator simply returns o.

```
template<typename T> octonion<T> operator - (octonion<T> const & o);
```

This unary operator returns the opposite of o.

Binary Addition Operators

```
template<typename T> octonion<T> operator + (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator + (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator + (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator + (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) += rhs`.

Binary Subtraction Operators

```
template<typename T> octonion<T> operator - (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator - (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator - (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator - (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) -= rhs`.

Binary Multiplication Operators

```
template<typename T> octonion<T> operator * (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator * (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator * (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator * (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) *= rhs`.

Binary Division Operators

```
template<typename T> octonion<T> operator / (T const & lhs, octonion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, T const & rhs);
template<typename T> octonion<T> operator / (::std::complex<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::std::com↵
plex<T> const & rhs);
template<typename T> octonion<T> operator / (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> octonion<T> operator / (octonion<T> const & lhs, octonion<T> const & rhs);
```

These operators return `octonion<T>(lhs) /= rhs`. It is of course still an error to divide by zero...

Binary Equality Operators

```
template<typename T> bool operator == (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator == (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator == (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> bool operator == (octonion<T> const & lhs, octonion<T> const & rhs);
```

These return true if and only if the four components of `octonion<T>(lhs)` are equal to their counterparts in `octonion<T>(rhs)`. As with any floating-type entity, this is essentially meaningless.

Binary Inequality Operators

```
template<typename T> bool operator != (T const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, T const & rhs);
template<typename T> bool operator != (::std::complex<T> const & lhs, octonion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::std::complex<T> const & rhs);
template<typename T> bool operator != (::boost::math::quaternion<T> const & lhs, octo↵
nion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, ::boost::math::qua↵
ternion<T> const & rhs);
template<typename T> bool operator != (octonion<T> const & lhs, octonion<T> const & rhs);
```

These return true if and only if `octonion<T>(lhs) == octonion<T>(rhs)` is false. As with any floating-type entity, this is essentially meaningless.

Stream Extractor

```
template<typename T, typename charT, class traits>
::std::basic_istream<charT,traits> & operator >> (::std::basic_istream<charT,traits> & is, oc↵
tonion<T> & o);
```

Extracts an octonion `o`. We accept any format which seems reasonable. However, since this leads to a great many ambiguities, decisions were made to lift these. In case of doubt, stick to lists of reals.

The input values must be convertible to T. If bad input is encountered, calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.5.3)).

Returns `is`.

Stream Inserter

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << (::std::basic_ostream<charT,traits> & os, oc↵
tonion<T> const & o);
```

Inserts the octonion `o` onto the stream `os` as if it were implemented as follows:

```
template<typename T, typename charT, class traits>
::std::basic_ostream<charT,traits> & operator << ( ::std::basic_ostream<charT,traits> & os,
octonion<T> const & o)
{
   ::std::basic_ostringstream<charT,traits> s;

   s.flags(os.flags());
   s.imbue(os.getloc());
   s.precision(os.precision());

   s << '(' << o.R_component_1() << ','
      << o.R_component_2() << ','
      << o.R_component_3() << ','
      << o.R_component_4() << ','
      << o.R_component_5() << ','
      << o.R_component_6() << ','
      << o.R_component_7() << ','
      << o.R_component_8() << ')';

   return os << s.str();
}
```

Octonion Value Operations

Real and Unreal

```
template<typename T> T  real(octonion<T> const & o);
template<typename T> octonion<T> unreal(octonion<T> const & o);
```

These return `o.real()` and `o.unreal()` respectively.

conj

```
template<typename T> octonion<T> conj(octonion<T> const & o);
```

This returns the conjugate of the octonion.

sup

```
template<typename T> T   sup(octonion<T> const & o);
```

This return the sup norm (the greatest among `abs(o.R_component_1())...abs(o.R_component_8())`) of the octonion.

l1

```
template<typename T> T   l1(octonion<T> const & o);
```

This return the l1 norm (`abs(o.R_component_1())+...+abs(o.R_component_8())`) of the octonion.

abs

```
template<typename T> T   abs(octonion<T> const & o);
```

This return the magnitude (Euclidian norm) of the octonion.

norm

```
template<typename T> T  norm(octonion<T>const  & o);
```

This return the (Cayley) norm of the octonion. The term "norm" might be confusing, as most people associate it with the Euclidian norm (and quadratic functionals). For this version of (the mathematical objects known as) octonions, the Euclidian norm (also known as magnitude) is the square root of the Cayley norm.

Octonion Creation Functions

```
template<typename T> octonion<T> spheric↵
al(T const & rho, T const & theta, T const & phi1, T const & phi2, T const & phi3, T const & phi4, T const & phi5, T const & phi6);
template<typename T> octonion<T> multi↵
polar(T const & rho1, T const & theta1, T const & rho2, T const & theta2, T const & rho3, T const & theta3, T const & rho4, T const & theta4);
template<typename T> octonion<T> cylindric↵
al(T const & r, T const & angle, T const & h1, T const & h2, T const & h3, T const & h4, T const & h5, T const & h6);
```

These build octonions in a way similar to the way polar builds complex numbers, as there is no strict equivalent to polar coordinates for octonions.

`spherical` is a simple transposition of `polar`, it takes as inputs a (positive) magnitude and a point on the hypersphere, given by three angles. The first of these, *theta* has a natural range of -pi to +pi, and the other two have natural ranges of -pi/2 to +pi/2 (as is the case with the usual spherical coordinates in $\boldsymbol{R}^3$). Due to the many symmetries and periodicities, nothing untoward happens if the magnitude is negative or the angles are outside their natural ranges. The expected degeneracies (a magnitude of zero ignores the angles settings...) do happen however.

`cylindrical` is likewise a simple transposition of the usual cylindrical coordinates in $\boldsymbol{R}^3$, which in turn is another derivative of planar polar coordinates. The first two inputs are the polar coordinates of the first ***C*** component of the octonion. The third and fourth inputs are placed into the third and fourth ***R*** components of the octonion, respectively.

`multipolar` is yet another simple generalization of polar coordinates. This time, both ***C*** components of the octonion are given in polar coordinates.

In this version of our implementation of octonions, there is no analogue of the complex value operation arg as the situation is somewhat more complicated.

Octonions Transcendentals

There is no `log` or `sqrt` provided for octonions in this implementation, and `pow` is likewise restricted to integral powers of the exponent. There are several reasons to this: on the one hand, the equivalent of analytic continuation for octonions ("branch cuts") remains to be investigated thoroughly (by me, at any rate...), and we wish to avoid the nonsense introduced in the standard by exponentiations of complexes by complexes (which is well defined, but not in the standard...). Talking of nonsense, saying that `pow(0,0)` is "implementation defined" is just plain brain-dead...

We do, however provide several transcendentals, chief among which is the exponential. That it allows for a "closed formula" is a result of the author (the existence and definition of the exponential, on the octonions among others, on the other hand, is a few centuries old). Basically, any converging power series with real coefficients which allows for a closed formula in ***C*** can be transposed to ***O***. More transcendentals of this type could be added in a further revision upon request. It should be noted that it is these functions which force the dependency upon the boost/math/special_functions/sinc.hpp and the boost/math/special_functions/sinhc.hpp headers.

exp

```
template<typename T>
octonion<T> exp(octonion<T> const & o);
```

Computes the exponential of the octonion.

cos

```
template<typename T>
octonion<T> cos(octonion<T> const & o);
```

Computes the cosine of the octonion

sin

```
template<typename T>
octonion<T> sin(octonion<T> const & o);
```

Computes the sine of the octonion.

tan

```
template<typename T>
octonion<T> tan(octonion<T> const & o);
```

Computes the tangent of the octonion.

cosh

```
template<typename T>
octonion<T> cosh(octonion<T> const & o);
```

Computes the hyperbolic cosine of the octonion.

sinh

```
template<typename T>
octonion<T> sinh(octonion<T> const & o);
```

Computes the hyperbolic sine of the octonion.

tanh

```
template<typename T>
octonion<T> tanh(octonion<T> const & o);
```

Computes the hyperbolic tangent of the octonion.

pow

```
template<typename T>
octonion<T>  pow(octonion<T> const & o, int n);
```

Computes the n-th power of the octonion q.

Test Program

The octonion_test.cpp test program tests octonions specialisations for float, double and long double (sample output).

If you define the symbol BOOST_OCTONION_TEST_VERBOSE, you will get additional output (verbose output); this will only be helpfull if you enable message output at the same time, of course (by uncommenting the relevant line in the test or by adding --log_level=messages to your command line,...). In that case, and if you are running interactively, you may in addition define the symbol BOOST_INTERACTIVE_TEST_INPUT_ITERATOR to interactively test the input operator with input of your choice from the standard input (instead of hard-coding it in the test).

Acknowledgements

The mathematical text has been typeset with Nisus Writer. Jens Maurer has helped with portability and standard adherence, and was the Review Manager for this library. More acknowledgements in the History section. Thank you to all who contributed to the discussion about this library.

History

- 1.5.9 - 13/5/2013: Incorporated into Boost.Math.
- 1.5.8 - 17/12/2005: Converted documentation to Quickbook Format.
- 1.5.7 - 25/02/2003: transitionned to the unit test framework; <boost/config.hpp> now included by the library header (rather than the test files), via <boost/math/quaternion.hpp>.
- 1.5.6 - 15/10/2002: Gcc2.95.x and stlport on linux compatibility by Alkis Evlogimenos (alkis@routescience.com).
- 1.5.5 - 27/09/2002: Microsoft VCPP 7 compatibility, by Michael Stevens (michael@acfr.usyd.edu.au); requires the /Za compiler option.
- 1.5.4 - 19/09/2002: fixed problem with multiple inclusion (in different translation units); attempt at an improved compatibility with Microsoft compilers, by Michael Stevens (michael@acfr.usyd.edu.au) and Fredrik Blomqvist; other compatibility fixes.
- 1.5.3 - 01/02/2002: bugfix and Gcc 2.95.3 compatibility by Douglas Gregor (gregod@cs.rpi.edu).
- 1.5.2 - 07/07/2001: introduced namespace math.
- 1.5.1 - 07/06/2001: (end of Boost review) now includes <boost/math/special_functions/sinc.hpp> and <boost/math/special_functions/sinhc.hpp> instead of <boost/special_functions.hpp>; corrected bug in sin (Daryle Walker); removed check for self-assignment (Gary Powel); made converting functions explicit (Gary Powel); added overflow guards for division operators and abs (Peter Schmitteckert); added sup and l1; used Vesa Karvonen's CPP metaprograming technique to simplify code.
- 1.5.0 - 23/03/2001: boostification, inlining of all operators except input, output and pow, fixed exception safety of some members (template version).
- 1.4.0 - 09/01/2001: added tan and tanh.
- 1.3.1 - 08/01/2001: cosmetic fixes.
- 1.3.0 - 12/07/2000: pow now uses Maarten Hilferink's (mhilferink@tip.nl) algorithm.
- 1.2.0 - 25/05/2000: fixed the division operators and output; changed many signatures.
- 1.1.0 - 23/05/2000: changed sinc into sinc_pi; added sin, cos, sinh, cosh.
- 1.0.0 - 10/08/1999: first public version.

To Do

- Improve testing.
- Rewrite input operatore using Spirit (creates a dependency).
- Put in place an Expression Template mechanism (perhaps borrowing from uBlas).

Integer Utilities (Greatest Common Divisor and Least Common Multiple)

This code has now been moved to Boost.Integer, please see here.

Note that for the time being the headers `<boost/math/common_factor.hpp>`, `<boost/math/common_factor_ct.hpp>` and `<boost/math/common_factor_rt.hpp>` will continue to exist and redirect to the moved headers.

Internal Details: Series, Rationals and Continued Fractions, Testing, and Development Tools

Overview

This section contains internal utilities used by the library's implementation along with tools used in development and testing. These tools have limitied documentation, but now have quite stable interfaces and may also be useful outside Boost.Math.

There is no doubt that these components can be improved, but they are also largely incidental to the main purpose of this library.

These tools are designed to "just get the job done", and receive minimal documentation here, in the hopes that they will help stimulate further submissions to this library.

Internal tools

Series Evaluation

Synopsis

```
#include <boost/math/tools/series.hpp>
```

```
namespace boost{ namespace math{ namespace tools{

template <class Functor, class U, class V>
inline typename Functor::result_type sum_series(Functor& func, const U& toler↵
ance, boost::uintmax_t& max_terms, const V& init_value);

template <class Functor, class U, class V>
inline typename Functor::result_type sum_series(Functor& func, const U& toler↵
ance, boost::uintmax_t& max_terms);

//
// The following interfaces are now deprecated:
//
template <class Functor>
typename Functor::result_type sum_series(Functor& func, int bits);

template <class Functor>
typename Functor::result_type sum_series(Functor& func, int bits, boost::uintmax_t& max_terms);

template <class Functor, class U>
typename Functor::result_type sum_series(Functor& func, int bits, U init_value);

template <class Functor, class U>
typename Functor::result_type sum_series(Func↵
tor& func, int bits, boost::uintmax_t& max_terms, U init_value);

template <class Functor>
typename Functor::result_type kahan_sum_series(Functor& func, int bits);

template <class Functor>
typename Functor::result_type kahan_sum_series(Func↵
tor& func, int bits, boost::uintmax_t& max_terms);

}}} // namespaces
```

Description

These algorithms are intended for the summation of infinite series.

Each of the algorithms takes a nullary-function object as the first argument: the function object will be repeatedly invoked to pull successive terms from the series being summed.

The second argument is the precision required, summation will stop when the next term is less than *tolerance* times the result. The deprecated versions of sum_series take an integer number of bits here - internally they just convert this to a tolerance and forward the call.

The third argument *max_terms* sets an upper limit on the number of terms of the series to evaluate. In addition, on exit the function will set *max_terms* to the actual number of terms of the series that were evaluated: this is particularly useful for profiling the convergence properties of a new series.

The final optional argument *init_value* is the initial value of the sum to which the terms of the series should be added. This is useful in two situations:

- Where the first value of the series has a different formula to successive terms. In this case the first value in the series can be passed as the last argument and the logic of the function object can then be simplified to return subsequent terms.
- Where the series is being added (or subtracted) from some other value: termination of the series will likely occur much more rapidly if that other value is passed as the last argument. For example, there are several functions that can be expressed as *1 - S(z)* where S(z) is an infinite series. In this case, pass -1 as the last argument and then negate the result of the summation to get the result of *1 - S(z)*.

The two *kahan_sum_series* variants of these algorithms maintain a carry term that corrects for roundoff error during summation. They are inspired by the *Kahan Summation Formula* that appears in What Every Computer Scientist Should Know About Floating-Point Arithmetic. However, it should be pointed out that there are very few series that require summation in this way.

Examples

These examples are all in ../../example/series.cpp

Let's suppose we want to implement *log(1+x)* via its infinite series,

$$\log(1+x) \quad = \quad \sum_{k=1}^{\infty} \frac{(-1)^{k-1} x^k}{k}$$

We begin by writing a small function object to return successive terms of the series:

```
template <class T>
struct log1p_series
{
   // we must define a result_type typedef:
   typedef T result_type;

   log1p_series(T x)
      : k(0), m_mult(-x), m_prod(-1) {}

   T operator()()
   {
      // This is the function operator invoked by the summation
      // algorithm, the first call to this operator should return
      // the first term of the series, the second call the second
      // term and so on.
      m_prod *= m_mult;
      return m_prod / ++k;
   }

private:
   int k;
   const T m_mult;
   T m_prod;
};
```

Implementing log(1+x) is now fairly trivial:

```
template <class T>
T log1p(T x)
{
   // We really should add some error checking on x here!
   assert(std::fabs(x) < 1);

   // Construct the series functor:
   log1p_series<T> s(x);
   // Set a limit on how many iterations we permit:
   boost::uintmax_t max_iter = 1000;
   // Add it up, with enough precision for full machine precision:
   return boost::math::tools::sum_series(s, std::numeric_limits<T>::epsilon(), max_iter);
}
```

We can almost use the code above for complex numbers as well - unfortunately we need a slightly different definition for epsilon, and within the functor, mixed complex and integer arithmetic is sadly not supported (as of C++17), so we need to cast out integers to floats:

```
template <class T>
struct log1p_series<std::complex<T> >
{
   // we must define a result_type typedef:
   typedef std::complex<T> result_type;

   log1p_series(std::complex<T> x)
      : k(0), m_mult(-x), m_prod(-1) {}

   std::complex<T> operator()()
   {
      // This is the function operator invoked by the summation
      // algorithm, the first call to this operator should return
      // the first term of the series, the second call the second
      // term and so on.
      m_prod *= m_mult;
      return m_prod / T(++k);
   }

private:
   int k;
   const std::complex<T> m_mult;
   std::complex<T> m_prod;
};


template <class T>
std::complex<T> log1p(std::complex<T> x)
{
   // We really should add some error checking on x here!
   assert(abs(x) < 1);

   // Construct the series functor:
   log1p_series<std::complex<T> > s(x);
   // Set a limit on how many iterations we permit:
   boost::uintmax_t max_iter = 1000;
   // Add it up, with enough precision for full machine precision:
   return boost::math::tools::sum_series(s, std::complex<T>(std::numeric_limits<T>::epsi↵
lon()), max_iter);
}
```

Of course with a few traits classes and a bit of meta-programming we could fold these two implementations into one, but that's beyond the scope of these examples.

Continued Fraction Evaluation

Synopsis

```
#include <boost/math/tools/fraction.hpp>
```

```
namespace boost{ namespace math{ namespace tools{

template <class Gen, class U>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_b(Gen& g, const U& tolerance, boost::uintmax_t& max_terms)

template <class Gen, class U>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_b(Gen& g, const U& tolerance)

template <class Gen, class U>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_a(Gen& g, const U& tolerance, boost::uintmax_t& max_terms)

template <class Gen, class U>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_a(Gen& g, const U& tolerance)

//
// These interfaces are present for legacy reasons, and are now deprecated:
//
template <class Gen>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_b(Gen& g, int bits);

template <class Gen>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_b(Gen& g, int bits, boost::uintmax_t& max_terms);

template <class Gen>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_a(Gen& g, int bits);

template <class Gen>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_a(Gen& g, int bits, boost::uintmax_t& max_terms);

}}} // namespaces
```

Description

Continued fractions are a common method of approximation. These functions all evaluate the continued fraction described by the *generator* type argument. The functions with an "_a" suffix evaluate the fraction:

$$\cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4} + \cdots}}}$$

and those with a "_b" suffix evaluate the fraction:

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4} + \ldots}}}$$

This latter form is somewhat more natural in that it corresponds with the usual definition of a continued fraction, but note that the first *a* value returned by the generator is discarded. Further, often the first *a* and *b* values in a continued fraction have different defining equations to the remaining terms, which may make the "_a" suffixed form more appropriate.

The generator type should be a function object which supports the following operations:

Expression	**Description**
Gen::result_type	The type that is the result of invoking operator(). This can be either an arithmetic or complex type, or a std::pair<> of arithmetic or complex types.
g()	Returns an object of type Gen::result_type. Each time this operator is called then the next pair of *a* and *b* values is returned. Or, if result_type is an arithmetic type, then the next *b* value is returned and all the *a* values are assumed to 1.

In all the continued fraction evaluation functions the *tolerance* parameter is the precision desired in the result, evaluation of the fraction will continue until the last term evaluated leaves the relative error in the result less than *tolerance*. The deprecated interfaces take a number of digits precision here, internally they just convert this to a tolerance and forward call.

If the optional *max_terms* parameter is specified then no more than *max_terms* calls to the generator will be made, and on output, *max_terms* will be set to actual number of calls made. This facility is particularly useful when profiling a continued fraction for convergence.

Implementation

Internally these algorithms all use the modified Lentz algorithm: refer to Numeric Recipes in C++, W. H. Press et all, chapter 5, (especially 5.2 Evaluation of continued fractions, p 175 - 179) for more information, also Lentz, W.J. 1976, Applied Optics, vol. 15, pp. 668-671.

Examples

All of these examples are in continued_fractions.cpp.

The golden ratio phi = 1.618033989... can be computed from the simplest continued fraction of all:

$$\text{Golden Ratio} \quad = \quad \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}}}$$

We begin by defining a generator function:

```
template <class T>
struct golden_ratio_fraction
{
   typedef T result_type;

   result_type operator()()
   {
      return 1;
   }
};
```

The golden ratio can then be computed to double precision using:

```
golden_ratio_fraction<double> func;
double gr = continued_fraction_a(
   func,
   std::numeric_limits<double>::epsilon());
std::cout << "The golden ratio is: " << gr << std::endl;
```

It's more usual though to have to define both the *a*'s and the *b*'s when evaluating special functions by continued fractions, for example the tan function is defined by:

$$\tan(z) = \cfrac{z}{1 - \cfrac{z^2}{3 - \cfrac{z^2}{5 - \cfrac{z^2}{7 - \cfrac{z^2}{9 - \cdots}}}}}$$

So its generator object would look like:

```
template <class T>
struct tan_fraction
{
private:
   T a, b;
public:
   tan_fraction(T v)
      : a(-v * v), b(-1)
   {}

   typedef std::pair<T, T> result_type;

   std::pair<T, T> operator()()
   {
      b += 2;
      return std::make_pair(a, b);
   }
};
```

Notice that if the continuant is subtracted from the *b* terms, as is the case here, then all the *a* terms returned by the generator will be negative. The tangent function can now be evaluated using:

```
template <class T>
T tan(T a)
{
   tan_fraction<T> fract(a);
   return a / continued_fraction_b(fract, std::numeric_limits<T>::epsilon());
}
```

Notice that this time we're using the "_b" suffixed version to evaluate the fraction: we're removing the leading *a* term during fraction evaluation as it's different from all the others.

Now we'll look at a couple of complex number examples, starting with the exponential integral which can be calculated via:

$$E_n(x) = \cfrac{e^{-x}}{n + x - \cfrac{n}{2 + n + x - \cfrac{2(n+1)}{4 + n + x - \cfrac{3(n+2)}{6 + n + x - \cfrac{4(n+3)}{8 + n + x - \ldots}}}}}$$

So our functor looks like this:

```
template <class T>
struct expint_fraction
{
   typedef std::pair<T, T> result_type;
   expint_fraction(unsigned n_, T z_) : b(z_ + T(n_)), i(-1), n(n_) {}
   std::pair<T, T> operator()()
   {
      std::pair<T, T> result = std::make_pair(-static_cast<T>((i + 1) * (n + i)), b);
      b += 2;
      ++i;
      return result;
   }
private:
   T b;
   int i;
   unsigned n;
};
```

We can finish the example by wrapping everything up in a function:

```
template <class T>
inline std::complex<T> expint_as_fraction(unsigned n, std::complex<T> const& z)
{
   boost::uintmax_t max_iter = 1000;
   expint_fraction<std::complex<T> > f(n, z);
   std::complex<T> result = boost::math::tools::continued_fraction_b(
      f,
      std::complex<T>(std::numeric_limits<T>::epsilon()),
      max_iter);
   result = exp(-z) / result;
   return result;
}
```

Notice how the termination condition is still expressed as a complex number, albeit one with zero imaginary part.

Our final example will use `continued_fraction_a`, in fact there is only one special function in our code which uses that variant, and it's the upper incomplete gamma function (Q), which can be calculated via:

$$\Gamma(a, x) \quad = \quad \frac{x^a e^{-x}}{x - a + 1 + \frac{a_k}{b_k + \frac{a_{k+1}}{b_{k+1} + \cdots}}} \quad ; \quad a_k = k(a-k) \quad ; \quad b_k = x - a + 2k + 1$$

In this case the first couple of terms are different from the rest, so our fraction will start with the first "regular" a term:

```
template <class T>
struct upper_incomplete_gamma_fract
{
private:
   typedef typename T::value_type scalar_type;
   T z, a;
   int k;
public:
   typedef std::pair<T, T> result_type;

   upper_incomplete_gamma_fract(T a1, T z1)
      : z(z1 - a1 + scalar_type(1)), a(a1), k(0)
   {
   }

   result_type operator()()
   {
      ++k;
      z += scalar_type(2);
      return result_type(scalar_type(k) * (a - scalar_type(k)), z);
   }
};
```

So now we can implement Q, this time using `continued_fraction_a`:

```
template <class T>
inline std::complex<T> gamma_Q_as_fraction(const std::complex<T>& a, const std::complex<T>& z)
{
   upper_incomplete_gamma_fract<std::complex<T> > f(a, z);
   std::complex<T> eps(std::numeric_limits<T>::epsilon());
   return pow(z, a) / (exp(z) *(z - a + T(1) + boost::math::tools::continued_frac↵
tion_a(f, eps)));
}
```

Tuples

Synopsis

```
#include <boost/math/tools/tuple.hpp>
```

Description

This header defines the type `boost::math::tuple`, the associated free functions `ignore`, `tie`, `make_tuple`, `get`, and associated types `tuple_size` and `tuple_element`.

These types and functions are aliases for:

- `std::tuple` etc when available, otherwise:
- `std::tr1::tuple` etc when available, otherwise:
- `boost::fusion::tuple` etc if the compiler supports it, otherwise:

• `boost::tuple`.

So this `boost::math::tuple` is strongly recommended for maximum portability.

Minimax Approximations and the Remez Algorithm

The directory libs/math/minimax contains a command line driven program for the generation of minimax approximations using the Remez algorithm. Both polynomial and rational approximations are supported, although the latter are tricky to converge: it is not uncommon for convergence of rational forms to fail. No such limitations are present for polynomial approximations which should always converge smoothly.

It's worth stressing that developing rational approximations to functions is often not an easy task, and one to which many books have been devoted. To use this tool, you will need to have a reasonable grasp of what the Remez algorithm is, and the general form of the approximation you want to achieve.

Unless you already familar with the Remez method, you should first read the brief background article explaining the principles behind the Remez algorithm.

The program consists of two parts:

main.cpp	Contains the command line parser, and all the calls to the Remez code.
f.cpp	Contains the function to approximate.

Therefore to use this tool, you must modify f.cpp to return the function to approximate. The tools supports multiple function approximations within the same compiled program: each as a separate variant:

```
NTL::RR f(const NTL::RR& x, int variant);
```

Returns the value of the function *variant* at point *x*. So if you wish you can just add the function to approximate as a new variant after the existing examples.

In addition to those two files, the program needs to be linked to a patched NTL library to compile.

Note that the function *f* must return the rational part of the approximation: for example if you are approximating a function *f(x)* then it is quite common to use:

```
f(x) = g(x)(Y + R(x))
```

where *g(x)* is the dominant part of *f(x)*, *Y* is some constant, and *R(x)* is the rational approximation part, usually optimised for a low absolute error compared to |Y|.

In this case you would define *f* to return *f(x)/g(x)* and then set the y-offset of the approximation to *Y* (see command line options below).

Many other forms are possible, but in all cases the objective is to split *f(x)* into a dominant part that you can evaluate easily using standard math functions, and a smooth and slowly changing rational approximation part. Refer to your favourite textbook for more examples.

Command line options for the program are as follows:

variant N	Sets the current function variant to N. This allows multiple functions that are to be approximated to be compiled into the same executable. Defaults to 0.
range a b	Sets the domain for the approximation to the range [a,b], defaults to [0,1].
relative	Sets the Remez code to optimise for relative error. This is the default at program startup. Note that relative error can only be used if f(x) has no roots over the range being optimised.

absolute	Sets the Remez code to optimise for absolute error.
pin [true|false]	"Pins" the code so that the rational approximation passes through the origin. Obviously only set this to *true* if R(0) must be zero. This is typically used when trying to preserve a root at [0,0] while also optimising for relative error.
order N D	Sets the order of the approximation to *N* in the numerator and *D* in the denominator. If *D* is zero then the result will be a polynomial approximation. There will be N+D+2 coefficients in total, the first coefficient of the numerator is zero if *pin* was set to true, and the first coefficient of the denominator is always one.
working-precision N	Sets the working precision of NTL::RR to *N* binary digits. Defaults to 250.
target-precision N	Sets the precision of printed output to *N* binary digits: set to the same number of digits as the type that will be used to evaluate the approximation. Defaults to 53 (for double precision).
skew val	"Skews" the initial interpolated control points towards one end or the other of the range. Positive values skew the initial control points towards the left hand side of the range, and negative values towards the right hand side. If an approximation won't converge (a common situation) try adjusting the skew parameter until the first step yields the smallest possible error. *val* should be in the range [-100,+100], the default is zero.
brake val	Sets a brake on each step so that the change in the control points is braked by *val%*. Defaults to 50, try a higher value if an approximation won't converge, or a lower value to get speedier convergence.
x-offset val	Sets the x-offset to *val*: the approximation will be generated for `f(S * (x + X)) + Y` where *X* is the x-offset, *S* is the x-scale and *Y* is the y-offset. Defaults to zero. To avoid rounding errors, take care to specify a value that can be exactly represented as a floating point number.
x-scale val	Sets the x-scale to *val*: the approximation will be generated for `f(S * (x + X)) + Y` where *S* is the x-scale, *X* is the x-offset and *Y* is the y-offset. Defaults to one. To avoid rounding errors, take care to specify a value that can be exactly represented as a floating point number.
y-offset val	Sets the y-offset to *val*: the approximation will be generated for `f(S * (x + X)) + Y` where *X* is the x-offset, *S* is the x-scale and *Y* is the y-offset. Defaults to zero. To avoid rounding errors, take care to specify a value that can be exactly represented as a floating point number.
y-offset auto	Sets the y-offset to the average value of f(x) evaluated at the two endpoints of the range plus the midpoint of the range. The calculated value is deliberately truncated to *float* precision (and should be stored as a *float* in your code). The approximation will be generated for `f(x + X) + Y` where *X* is the x-offset and *Y* is the y-offset. Defaults to zero.
graph N	Prints N evaluations of f(x) at evenly spaced points over the range being optimised. If unspecified then *N* defaults to 3. Use to check that f(x) is indeed smooth over the range of interest.
step N	Performs *N* steps, or one step if *N* is unspecified. After each step prints: the peek error at the extrema of the error function of the approximation, the theoretical error term solved for on the last step, and the maximum relative change in the location of the Chebyshev control points. The approximation is converged on the minimax solution when the two error terms are (approximately) equal, and the change in the control points has decreased to a suitably small value.
test [float|double|long]	Tests the current approximation at float, double, or long double precision. Useful to check for rounding errors in evaluating the approximation at fixed precision. Tests are conducted

	at the extrema of the error function of the approximation, and at the zeros of the error function.
test [float|double|long] N	Tests the current approximation at float, double, or long double precision. Useful to check for rounding errors in evaluating the approximation at fixed precision. Tests are conducted at N evenly spaced points over the range of the approximation. If none of [float|double|long] are specified then tests using NTL::RR, this can be used to obtain the error function of the approximation.
rescale a b	Takes the current Chebeshev control points, and rescales them over a new interval [a,b]. Sometimes this can be used to obtain starting control points for an approximation that can not otherwise be converged.
rotate	Moves one term from the numerator to the denominator, but keeps the Chebyshev control points the same. Sometimes this can be used to obtain starting control points for an approximation that can not otherwise be converged.
info	Prints out the current approximation: the location of the zeros of the error function, the location of the Chebyshev control points, the x and y offsets, and of course the coefficients of the polynomials.

Relative Error and Testing

Synopsis

```
#include <boost/math/tools/test.hpp>
```

Important

The header `boost/math/tools/test.hpp` is located under `libs/math/include_private` and is not installed to the usual locations by default, you will need to add `libs/math/include_private` to your compiler's include path in order to use this header.

```
template <class T>
T relative_error(T a, T b);

template <class A, class F1, class F2>
test_result<see-below> test(const A& a, F1 test_func, F2 expect_func);
```

Description

```
template <class T>
T relative_error(T a, T v);
```

Returns the relative error between *a* and *v* using the usual formula:

$$\max\left(\left|\frac{a-v}{a}\right|, \left|\frac{a-v}{v}\right|\right)$$

In addition the value returned is zero if:

- Both *a* and *v* are infinite.
- Both *a* and *v* are denormalised numbers or zero.

Otherwise if only one of *a* and *v* is zero then the value returned is 1.

```
template <class A, class F1, class F2>
test_result<see-below> test(const A& a, F1 test_func, F2 expect_func);
```

This function is used for testing a function against tabulated test data.

The return type contains statistical data on the relative errors (max, mean, variance, and the number of test cases etc), as well as the row of test data that caused the largest relative error. Public members of type test_result are:

`unsigned worst()const;`	Returns the row at which the worst error occurred.
`T min()const;`	Returns the smallest relative error found.
`T max()const;`	Returns the largest relative error found.
`T mean()const;`	Returns the mean error found.
`boost::uintmax_t count()const;`	Returns the number of test cases.
`T variance()const;`	Returns the variance of the errors found.
`T variance1()const;`	Returns the unbiased variance of the errors found.
`T rms()const`	Returns the Root Mean Square, or quadratic mean of the errors.
`test_result& operator+=(const test_result& t)`	Combines two test_result's into a single result.

The template parameter of test_result, is the same type as the values in the two dimensional array passed to function *test*, roughly that's `A::value_type::value_type`.

Parameter *a* is a matrix of test data: and must be a standard library Sequence type, that contains another Sequence type: typically it will be a two dimensional instance of `boost::array`. Each row of *a* should contain all the parameters that are passed to the function under test as well as the expected result.

Parameter *test_func* is the function under test, it is invoked with each row of test data in *a*. Typically type F1 is created with Boost.Lambda: see the example below.

Parameter *expect_func* is a functor that extracts the expected result from a row of test data in *a*. Typically type F2 is created with Boost.Lambda: see the example below.

If the function under test returns a non-finite value when a finite result is expected, or if a gross error is found, then a message is sent to `std::cerr`, and a call to BOOST_ERROR() made (which means that including this header requires you use Boost.Test). This is mainly a debugging/development aid (and a good place for a breakpoint).

Example

Suppose we want to test the tgamma and lgamma functions, we can create a two dimensional matrix of test data, each row is one test case, and contains three elements: the input value, and the expected results for the tgamma and lgamma functions respectively.

```
static const boost::array<boost::array<TestType, 3>, NumberOfTests>
   factorials = {
      /* big array of test data goes here */
   };
```

Now we can invoke the test function to test tgamma:

```
using namespace boost::math::tools;
using namespace boost::lambda;

// get a pointer to the function under test:
TestType (*funcp)(TestType) = boost::math::tgamma;

// declare something to hold the result:
test_result<TestType> result;
//
// and test tgamma against data:
//
result = test(
   factorials,
   bind(funcp, ret<TestType>(_1[0])), // calls tgamma with factorials[row][0]
   ret<TestType>(_1[1])               // extracts the expected result from factorials[row][1]
);
//
// Print out some results:
//
std::cout << "The Mean was " << result.mean() << std::endl;
std::cout << "The worst error was " << (result.max)() << std::endl;
std::cout << "The worst error was at row " << result.worst_case() << std::endl;
//
// same again with lgamma this time:
//
funcp = boost::math::lgamma;
result = test(
   factorials,
   bind(funcp, ret<TestType>(_1[0])), // calls tgamma with factorials[row][0]
   ret<TestType>(_1[2])               // extracts the expected result from factorials[row][2]
);
//
// etc ...
//
```

Graphing, Profiling, and Generating Test Data for Special Functions

The class `test_data` and associated helper functions are designed so that in just a few lines of code you should be able to:

- Profile a continued fraction, or infinite series for convergence and accuracy.
- Generate csv data from a special function that can be imported into your favorite graphing program (or spreadsheet) for further analysis.
- Generate high precision test data.

Synopsis

```
#include <boost/math/tools/test_data.hpp>
```

Important

This is a non-core Boost.Math header that is predominantly used for internal maintenance of the library: as a result the library is located under `libs/math/include_private` and you will need to add that directory to your include path in order to use this feature.

```
namespace boost{ namespace math{ namespace tools{

enum parameter_type
{
   random_in_range = 0,
   periodic_in_range = 1,
   power_series = 2,
   dummy_param = 0x80,
};

template <class T>
struct parameter_info;

template <class T>
parameter_info<T> make_random_param(T start_range, T end_range, int n_points);

template <class T>
parameter_info<T> make_periodic_param(T start_range, T end_range, int n_points);

template <class T>
parameter_info<T> make_power_param(T basis, int start_exponent, int end_exponent);

template <class T>
bool get_user_parameter_info(parameter_info<T>& info, const char* param_name);

template <class T>
class test_data
{
public:
   typedef std::vector<T> row_type;
   typedef row_type value_type;
private:
   typedef std::set<row_type> container_type;
public:
   typedef typename container_type::reference reference;
   typedef typename container_type::const_reference const_reference;
   typedef typename container_type::iterator iterator;
   typedef typename container_type::const_iterator const_iterator;
   typedef typename container_type::difference_type difference_type;
   typedef typename container_type::size_type size_type;

   // creation:
   test_data(){}
   template <class F>
   test_data(F func, const parameter_info<T>& arg1);

   // insertion:
   template <class F>
   test_data& insert(F func, const parameter_info<T>& arg1);

   template <class F>
   test_data& insert(F func, const parameter_info<T>& arg1,
                     const parameter_info<T>& arg2);

   template <class F>
   test_data& insert(F func, const parameter_info<T>& arg1,
                     const parameter_info<T>& arg2,
                     const parameter_info<T>& arg3);

   void clear();

   // access:
   iterator begin();
```

```
    iterator end();
    const_iterator begin()const;
    const_iterator end()const;
    bool operator==(const test_data& d)const;
    bool operator!=(const test_data& d)const;
    void swap(test_data& other);
    size_type size()const;
    size_type max_size()const;
    bool empty()const;

    bool operator < (const test_data& dat)const;
    bool operator <= (const test_data& dat)const;
    bool operator > (const test_data& dat)const;
    bool operator >= (const test_data& dat)const;
};

template <class charT, class traits, class T>
std::basic_ostream<charT, traits>& write_csv(
            std::basic_ostream<charT, traits>& os,
            const test_data<T>& data);

template <class charT, class traits, class T>
std::basic_ostream<charT, traits>& write_csv(
            std::basic_ostream<charT, traits>& os,
            const test_data<T>& data,
            const charT* separator);

template <class T>
std::ostream& write_code(std::ostream& os,
                         const test_data<T>& data,
                         const char* name);

}}} // namespaces
```

Description

This tool is best illustrated with the following series of examples.

The functionality of test_data is split into the following parts:

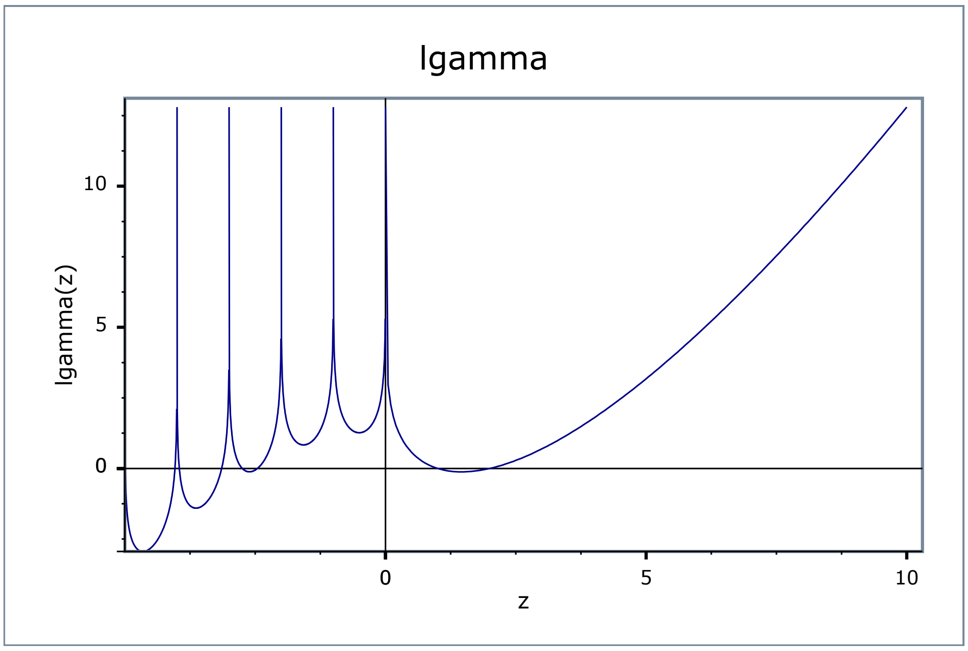
- A functor that implements the function for which data is being generated: this is the bit you have to write.
- One of more parameters that are to be passed to the functor, these are described in fairly abstract terms: give me N points distributed like *this* etc.
- The class test_data, that takes the functor and descriptions of the parameters and computes how ever many output points have been requested, these are stored in a sorted container.
- Routines to iterate over the test_data container and output the data in either csv format, or as C++ source code (as a table using Boost.Array).

Example 1: Output Data for Graph Plotting

For example, lets say we want to graph the lgamma function between -3 and 100, one could do this like so:

```
#include <boost/math/tools/test_data.hpp>
#include <boost/math/special_functions/gamma.hpp>

int main()
{
   using namespace boost::math::tools;

   // create an object to hold the data:
   test_data<double> data;

   // insert 500 points at uniform intervals between just after -3 and 100:
   double (*pf)(double) = boost::math::lgamma;
   data.insert(pf, make_periodic_param(-3.0 + 0.00001, 100.0, 500));

   // print out in csv format:
   write_csv(std::cout, data, ", ");
   return 0;
}
```

Which, when plotted, results in:



Example 2: Creating Test Data

As a second example, let's suppose we want to create highly accurate test data for a special function. Since many special functions have two or more independent parameters, it's very hard to effectively cover all of the possible parameter space without generating gigabytes of data at great computational expense. A second best approach is to provide the tools by which a user (or the library maintainer) can quickly generate more data on demand to probe the function over a particular domain of interest.

In this example we'll generate test data for the beta function using NTL::RR at 1000 bit precision. Rather than call our generic version of the beta function, we'll implement a deliberately naive version of the beta function using lgamma, and rely on the high precision of the data type used to get results accurate to at least 128-bit precision. In this way our test data is independent of whatever clever tricks we may wish to use inside the our beta function.

To start with then, here's the function object that creates the test data:

```
#include <boost/math/tools/ntl.hpp>
#include <boost/math/special_functions/gamma.hpp>
#include <boost/math/tools/test_data.hpp>
#include <fstream>

#include <boost/math/tools/test_data.hpp>

using namespace boost::math::tools;

struct beta_data_generator
{
   NTL::RR operator()(NTL::RR a, NTL::RR b)
   {
      //
      // If we throw a domain error then test_data will
      // ignore this input point. We'll use this to filter
      // out all cases where a < b since the beta function
      // is symmetrical in a and b:
      //
      if(a < b)
         throw std::domain_error("");

      // very naively calculate spots with lgamma:
      NTL::RR g1, g2, g3;
      int s1, s2, s3;
      g1 = boost::math::lgamma(a, &s1);
      g2 = boost::math::lgamma(b, &s2);
      g3 = boost::math::lgamma(a+b, &s3);
      g1 += g2 - g3;
      g1 = exp(g1);
      g1 *= s1 * s2 * s3;
      return g1;
   }
};
```

To create the data, we'll need to input the domains for a and b for which the function will be tested: the function `get_user_parameter_info` is designed for just that purpose. The start of main will look something like:

```
// Set the precision on RR:
NTL::RR::SetPrecision(1000); // bits.
NTL::RR::SetOutputPrecision(40); // decimal digits.

parameter_info<NTL::RR> arg1, arg2;
test_data<NTL::RR> data;

std::cout << "Welcome.\n"
   "This program will generate spot tests for the beta function:\n"
   "  beta(a, b)\n\n";

bool cont;
std::string line;

do{
   // prompt the user for the domain of a and b to test:
   get_user_parameter_info(arg1, "a");
   get_user_parameter_info(arg2, "b");

   // create the data:
   data.insert(beta_data_generator(), arg1, arg2);

   // see if the user want's any more domains tested:
   std::cout << "Any more data [y/n]?";
   std::getline(std::cin, line);
   boost::algorithm::trim(line);
   cont = (line == "y");
}while(cont);
```

Caution

At this point one potential stumbling block should be mentioned: test_data<>::insert will create a matrix of test data when there are two or more parameters, so if we have two parameters and we're asked for a thousand points on each, that's a *million test points in total*. Don't say you weren't warned!

There's just one final step now, and that's to write the test data to file:

```
std::cout << "Enter name of test data file [default=beta_data.ipp]";
std::getline(std::cin, line);
boost::algorithm::trim(line);
if(line == "")
   line = "beta_data.ipp";
std::ofstream ofs(line.c_str());
write_code(ofs, data, "beta_data");
```

The format of the test data looks something like:

```
#define SC_(x) static_cast<T>(BOOST_JOIN(x, L))
   static const boost::array<boost::array<T, 3>, 1830>
   beta_med_data = {
      SC_(0.4883005917072296142578125),
      SC_(0.4883005917072296142578125),
      SC_(3.245912809500479157065104747353807392371),
      SC_(3.5808107852935791015625),
      SC_(0.4883005917072296142578125),
      SC_(1.007653173802923954909901438393379243537),
      /* ... lots of rows skipped */
};
```

The first two values in each row are the input parameters that were passed to our functor and the last value is the return value from the functor. Had our functor returned a boost::math::tuple rather than a value, then we would have had one entry for each element in the tuple in addition to the input parameters.

The first #define serves two purposes:

- It reduces the file sizes considerably: all those `static_cast`'s add up to a lot of bytes otherwise (they are needed to suppress compiler warnings when `T` is narrower than a `long double`).
- It provides a useful customisation point: for example if we were testing a user-defined type that has more precision than a `long double` we could change it to:

```
#define SC_(x) lexical_cast<T>(BOOST_STRINGIZE(x))
```

in order to ensure that no truncation of the values occurs prior to conversion to `T`. Note that this isn't used by default as it's rather hard on the compiler when the table is large.

Example 3: Profiling a Continued Fraction for Convergence and Accuracy

Alternatively, lets say we want to profile a continued fraction for convergence and error. As an example, we'll use the continued fraction for the upper incomplete gamma function, the following function object returns the next a_N and b_N of the continued fraction each time it's invoked:

```
template <class T>
struct upper_incomplete_gamma_fract
{
private:
   T z, a;
   int k;
public:
   typedef std::pair<T,T> result_type;

   upper_incomplete_gamma_fract(T a1, T z1)
      : z(z1-a1+1), a(a1), k(0)
   {
   }

   result_type operator()()
   {
      ++k;
      z += 2;
      return result_type(k * (a - k), z);
   }
};
```

We want to measure both the relative error, and the rate of convergence of this fraction, so we'll write a functor that returns both as a boost::math::tuple: class test_data will unpack the tuple for us, and create one column of data for each element in the tuple (in addition to the input parameters):

```
#include <boost/math/tools/test_data.hpp>
#include <boost/math/tools/test.hpp>
#include <boost/math/special_functions/gamma.hpp>
#include <boost/math/tools/ntl.hpp>
#include <boost/math/tools/tuple.hpp>

template <class T>
struct profile_gamma_fraction
{
   typedef boost::math::tuple<T, T> result_type;

   result_type operator()(T val)
   {
      using namespace boost::math::tools;
      // estimate the true value, using arbitary precision
      // arithmetic and NTL::RR:
      NTL::RR rval(val);
      upper_incomplete_gamma_fract<NTL::RR> f1(rval, rval);
      NTL::RR true_val = continued_fraction_a(f1, 1000);
      //
      // Now get the aproximation at double precision, along with the number of
      // iterations required:
      boost::uintmax_t iters = std::numeric_limits<boost::uintmax_t>::max();
      upper_incomplete_gamma_fract<T> f2(val, val);
      T found_val = continued_fraction_a(f2, std::numeric_limits<T>::digits, iters);
      //
      // Work out the relative error, as measured in units of epsilon:
      T err = real_cast<T>(relative_error(true_val, NTL::RR(found_val)) / std::numeric_lim↵
its<T>::epsilon());
      //
      // now just return the results as a tuple:
      return boost::math::make_tuple(err, iters);
   }
};
```

Feeding that functor into test_data allows rapid output of csv data, for whatever type `T` we may be interested in:

```
int main()
{
   using namespace boost::math::tools;
   // create an object to hold the data:
   test_data<double> data;
   // insert 500 points at uniform intervals between just after 0 and 100:
   data.insert(profile_gamma_fraction<double>(), make_periodic_param(0.01, 100.0, 100));
   // print out in csv format:
   write_csv(std::cout, data, ", ");
   return 0;
}
```

This time there's no need to plot a graph, the first few rows are:

```
a and z,  Error/epsilon,  Iterations required

0.01,     9723.14,        4726
1.0099,   9.54818,        87
2.0098,   3.84777,        40
3.0097,   0.728358,       25
4.0096,   2.39712,        21
5.0095,   0.233263,       16
```

So it's pretty clear that this fraction shouldn't be used for small values of a and z.

reference

Most of this tool has been described already in the examples above, we'll just add the following notes on the non-member functions:

```
template <class T>
parameter_info<T> make_random_param(T start_range, T end_range, int n_points);
```

Tells class test_data to test *n_points* random values in the range [start_range,end_range].

```
template <class T>
parameter_info<T> make_periodic_param(T start_range, T end_range, int n_points);
```

Tells class test_data to test *n_points* evenly spaced values in the range [start_range,end_range].

```
template <class T>
parameter_info<T> make_power_param(T basis, int start_exponent, int end_exponent);
```

Tells class test_data to test points of the form $basis + R * 2^{expon}$ for each *expon* in the range [start_exponent, end_exponent], and *R* a random number in [0.5, 1].

```
template <class T>
bool get_user_parameter_info(parameter_info<T>& info, const char* param_name);
```

Prompts the user for the parameter range and form to use.

Finally, if we don't want the parameter to be included in the output, we can tell test_data by setting it a "dummy parameter":

```
parameter_info<double> p = make_random_param(2.0, 5.0, 10);
p.type |= dummy_param;
```

This is useful when the functor used transforms the parameter in some way before passing it to the function under test, usually the functor will then return both the transformed input and the result in a tuple, so there's no need for the original pseudo-parameter to be included in program output.

Use with User-Defined Floating-Point Types - Boost.Multiprecision and others

Using Boost.Math with High-Precision Floating-Point Libraries

The special functions, distributions, constants and tools in this library can be used with a number of high-precision libraries, including:

- Boost.Multiprecision
- e_float (TOMS Algorithm 910)
- NTL A Library for doing Number Theory
- GNU Multiple Precision Arithmetic Library
- GNU MPFR library
- GCC 128-bit floating-point type
- Intel _Quad type

The last four have some license restrictions; only Boost.Multiprecision when using the `cpp_float` backend can provide an unrestricted Boost license.

At present, the price of a free license is slightly lower speed.

Of course, the main cost of higher precision is very much decreased (usually at least hundred-fold) computation speed, and big increases in memory use.

Some libraries offer true arbitrary-precision arithmetic where the precision is limited only by available memory and compute time, but most are used at some arbitrarily-fixed precision, say 100 decimal digits, like Boost.Multiprecision `cpp_dec_float_100`.

Boost.Multiprecision can operate in both ways, but the most popular choice is likely to be about a hundred decimal digits, though examples of computing about a million digits have been demonstrated.

Why use a high-precision library rather than built-in floating-point types?

For nearly all applications, the built-in floating-point types, `double` (and `long double` if this offers higher precision than `double`) offer enough precision, typically a dozen decimal digits.

Some reasons why one would want to use a higher precision:

- A much more precise result (many more digits) is just a requirement.
- The range of the computed value exceeds the range of the type: factorials are the textbook example.
- Using `double` is (or may be) too inaccurate.
- Using `long double` (or may be) is too inaccurate.
- Using an extended-precision type implemented in software as double-double (Darwin) is sometimes unpredictably inaccurate.
- Loss of precision or inaccuracy caused by extreme arguments or cancellation errors.
- An accuracy as good as possible for a chosen built-in floating-point type is required.

- As a reference value, for example, to determine the inaccuracy of a value computed with a built-in floating point type, (perhaps even using some quick'n'dirty algorithm). The accuracy of many functions and distributions in Boost.Math has been measured in this way from tables of very high precision (up to 1000 decimal digits).

Many functions and distributions have differences from exact values that are only a few least significant bits - computation noise. Others, often those for which analytical solutions are not available, require approximations and iteration: these may lose several decimal digits of precision.

Much larger loss of precision can occur for boundary or corner cases, often caused by cancellation errors.

(Some of the worst and most common examples of cancellation error or loss of significance can be avoided by using complements: see why complements?).

If you require a value which is as accurate as can be represented in the floating-point type, and is thus the closest representable value correctly rounded to nearest, and has an error less than 1/2 a least significant bit or ulp it may be useful to use a higher-precision type, for example, `cpp_dec_float_50`, to generate this value. Conversion of this value to a built-in floating-point type ('float', `double` or `long double`) will not cause any further loss of precision. A decimal digit string will also be 'read' precisely by the compiler into a built-in floating-point type to the nearest representable value.

Note

In contrast, reading a value from an `std::istream` into a built-in floating-point type is **not guaranteed by the C++ Standard** to give the nearest representable value.

William Kahan coined the term Table-Maker's Dilemma for the problem of correctly rounding functions. Using a much higher precision (50 or 100 decimal digits) is a practical way of generating (almost always) correctly rounded values.

Using Boost.Multiprecision

All new projects are recommended to use Boost.Multiprecision.

Using Boost.Multiprecision `cpp_float` for numerical calculations with high precision.

The Boost.Multiprecision library can be used for computations requiring precision exceeding that of standard built-in types such as float, double and long double. For extended-precision calculations, Boost.Multiprecision supplies a template data type called cpp_dec_float. The number of decimal digits of precision is fixed at compile-time via template parameter.

To use these floating-point types and constants, we need some includes:

```
#include <boost/math/constants/constants.hpp>

#include <boost/multiprecision/cpp_dec_float.hpp>
// using boost::multiprecision::cpp_dec_float

#include <iostream>
#include <limits>
```

So now we can demonstrate with some trivial calculations:

Using `typedef cpp_dec_float_50` hides the complexity of multiprecision, allows us to define variables with 50 decimal digit precision just like built-in `double`.

```
using boost::multiprecision::cpp_dec_float_50;

cpp_dec_float_50 seventh = cpp_dec_float_50(1) / 7;  // 1 / 7
```

By default, output would only show the standard 6 decimal digits, so set precision to show all 50 significant digits, including any trailing zeros.

```
std::cout.precision(std::numeric_limits<cpp_dec_float_50>::digits10);
std::cout << std::showpoint << std::endl; // Append any trailing zeros.
std::cout << seventh << std::endl;
```

which outputs:

```
0.14285714285714285714285714285714285714285714285714
```

We can also use Boost.Math constants like π, guaranteed to be initialized with the very last bit of precision for the floating-point type.

```
std::cout << "pi = " << boost::math::constants::pi<cpp_dec_float_50>() << std::endl;
cpp_dec_float_50 circumference = boost::math::constants::pi<cpp_dec_float_50>() * 2 * seventh;
std::cout << "c =  "<< circumference << std::endl;
```

which outputs

```
pi = 3.1415926535897932384626433832795028841971693993751

c =  0.89759790102565521098932668093700082405633411410717
```

The full source of this example is at big_seventh.cpp

Using Boost.Multiprecision to generate a high-precision array of sine coefficents for use with FFT.

The Boost.Multiprecision library can be used for computations requiring precision exceeding that of standard built-in types such as `float`, `double` and `long double`. For extended-precision calculations, Boost.Multiprecision supplies a template data type called `cpp_bin_float`. The number of decimal digits of precision is fixed at compile-time via a template parameter.

One often needs to compute tables of numbers in mathematical software. To avoid the Table-maker's dilemma it is necessary to use a higher precision type to compute the table values so that they have the nearest representable bit-pattern for the type, say `double`, of the table value.

This example is a program `fft_since_table.cpp` that writes a header file `sines.hpp` containing an array of sine coefficients for use with a Fast Fourier Transform (FFT), that can be included by the FFT program.

To use Boost.Multiprecision's high-precision floating-point types and constants, we need some includes:

```
#include <boost/math/constants/constants.hpp>
// using boost::math::constants::pi;

#include <boost/multiprecision/cpp_bin_float.hpp> // for
// using boost::multiprecision::cpp_bin_float and
// using boost::multiprecision::cpp_bin_float_50;
// using boost::multiprecision::cpp_bin_float_quad;

#include <boost/array.hpp> // or <array> for std::array

#include <iostream>
#include <limits>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <iterator>
#include <fstream>
```

First, this example defines a prolog text string which is a C++ comment with the program licence, copyright etc. (You would of course, tailor this to your needs, including **your** copyright claim). This will appear at the top of the written header file `sines.hpp`.

```
using boost::multiprecision::cpp_bin_float_50;
using boost::math::constants::pi;
```

A fast Fourier transform (FFT), for example, may use a table of the values of sin((π/2^n) in its implementation details. In order to maximize the precision in the FFT implementation, the precision of the tabulated trigonometric values should exceed that of the built-in floating-point type used in the FFT.

The sample below computes a table of the values of sin(π/2^n) in the range 1 <= n <= 31.

This program makes use of, among other program elements, the data type `boost::multiprecision::cpp_bin_float_50` for a precision of 50 decimal digits from Boost.Multiprecision, the value of constant π retrieved from Boost.Math, guaranteed to be initialized with the very last bit of precision for the type, here `cpp_bin_float_50`, and a C++11 lambda function combined with `std::for_each()`.

define the number of values (32) in the array of sines.

```
std::size_t size = 32U;
//cpp_bin_float_50 p = pi<cpp_bin_float_50>();
cpp_bin_float_50 p = boost::math::constants::pi<cpp_bin_float_50>();

std::vector <cpp_bin_float_50> sin_values (size);
unsigned n = 1U;
// Generate the sine values.
std::for_each
(
  sin_values.begin (),
  sin_values.end (),
  [&n](cpp_bin_float_50& y)
  {
    y = sin( pi<cpp_bin_float_50>() / pow(cpp_bin_float_50 (2), n));
    ++n;
  }
);
```

Define the floating-point type for the generated file, either built-in `double,` float, or `long double`, or a user defined type like `cpp_bin_float_50`.

```
std::string fp_type = "double";

std::cout << "Generating an `std::array` or `boost::array` for floating-point type: "
  << fp_type << ". " << std::endl;
```

By default, output would only show the standard 6 decimal digits, so set precision to show enough significant digits for the chosen floating-point type. For `cpp_bin_float_50` is 50. (50 decimal digits should be ample for most applications).

```
std::streamsize precision = std::numeric_limits<cpp_bin_float_50>::digits10;

std::cout << "Sines table precision is " << precision << " decimal digits. " << std::endl;
```

Of course, one could also choose a lower precision for the table values, for example,

`std::streamsize precision = std::numeric_limits<cpp_bin_float_quad>::max_digits10;`

128-bit 'quad' precision of 36 decimal digits would be sufficient for the most precise current `long double` implementations using 128-bit. In general, it should be a couple of decimal digits more (guard digits) than `std::numeric_limits<RealType>::max_digits10` for the target system floating-point type. (If the implementation does not provide `max_digits10`, the the Kahan formula `std::numeric_limits<RealType>::digits * 3010/10000 + 2` can be used instead).

The compiler will read these values as decimal digits strings and use the nearest representation for the floating-point type.

Now output all the sine table, to a file of your chosen name.

```
const char sines_name[] = "sines.hpp";  // Assuming in same directory as .exe

std::ofstream fout(sines_name, std::ios_base::out);  // Creates if no file exists,
// & uses default overwrite/ ios::replace.
if (fout.is_open() == false)
{  // failed to open OK!
  std::cout << "Open file " << sines_name << " failed!" << std::endl;
  return EXIT_FAILURE;
}
else
{ // Write prolog etc as a C++ comment.
  std::cout << "Open file " << sines_name << " for output OK." << std::endl;
  fout << prolog
  << "// Table of " << sin_values.size() << " values with "
    << precision << " decimal digits precision,\n"
    "// generated by program fft_sines_table.cpp.\n" << std::endl;

fout << "#include <array> // std::array" << std::endl;

// Write the table of sines as a C++ array.
  fout <<  "\nstatic const std::array<double, " << size << "> sines =\n"
  "{{\n"; // 2nd { needed for some old GCC compiler versions.
  fout.precision(precision);

  for (unsigned int i = 0U; ;)
  {
    fout << "  " << sin_values[i];
    if (i == sin_values.size()-1)
    { // next is last value.
      fout << "\n}};  // array sines\n"; // 2nd } needed for some old GCC compiler versions.
      break;
    }
    else
    {
      fout << ",\n";
      i++;
    }
  } // for

  fout.close();
  std::cout << "Closed file " << sines_name << " for output." << std::endl;
}
```

The output file generated can be seen at ../../example/sines.hpp

The table output is:

```
The printed table is:

  1
  0.70710678118654752440084436210484903928483593768847
  0.38268343236508977172845998403039886676134456248563
  0.19509032201612826784828486847702224092769161775195
  0.098017140329560601994195563888641845861136673167501
  0.049067674327418014254954976942682658314745363025753
  0.024541228522912288031734529459282925065466119239451
  0.012271538285719926079408261951003212140372319591769
  0.0061358846491544753596402345903725809170578863173913
  0.0030679567629659762701453654909198425189446102134
52
  0.0015339801862847656123036971502640790799548645752374
  0.00076699031874270452693856835794857664314091945206328
  0.00038349518757139558907246168118138126339502603496474
  0.00019174759731070330743990956198900093346887403385916
  9.5873799095977345870517210976476351187065612851145e-05
  4.7936899603066884549003990494658872746866687685767e-05
  2.3968449808418218729186577165021820094761474895673e-05
  1.1984224905069706421521561596988984804731977538387e-05
  5.9921124526424278428797118088908617299871778780951e-06
  2.9960562263346607504548128083570598118251878683408e-06
  1.4980281131690112288542788461553611206917585861527e-06
  7.4901405658471572113049856673065563715595930217207e-07
  3.7450702829238412390316917908463317739740476297248e-07
  1.8725351414619534486882457659356361712045272098287e-07
  9.3626757073098082799067286680885620193236507169473e-08
  4.681337853654909269511551813854009695950362701667e-08
  2.3406689268274552759505493419034844037886207223779e-08
  1.1703344634137277181246213503238103798093456639976e-08
  5.8516723170686386908097901008341396943900085051757e-09
  2.9258361585343193579282304690689559020175857150074e-09
  1.4629180792671596805295321618659637103742615227834e-09
*/
```

The output can be copied as text and readily integrated into a given source code. Alternatively, the output can be written to a text or even be used within a self-written automatic code generator as this example.

A computer algebra system can be used to verify the results obtained from Boost.Math and Boost.Multiprecision. For example, the Wolfram Mathematica computer algebra system can obtain a similar table with the command:

```
Table[N[Sin[Pi / (2^n)], 50], {n, 1, 31, 1}]
```

The Wolfram Alpha computational knowledge engine can also be used to generate this table. The same command can be pasted into the compute box.

The full source of this example is at fft_sines_table.cpp

Using with GCC's __float128 datatype

At present support for GCC's native `__float128` datatype is extremely limited: the numeric constants will all work with that type, and that's about it. If you want to use the distributions or special functions then you will need to provide your own wrapper header that:

- Provides `std::numeric_limits<__float128>` support.
- Provides overloads of the standard library math functions for type `__float128` and which forward to the libquadmath equivalents.

Ultimately these facilities should be provided by GCC and `libstdc++`.

Using With MPFR or GMP - High-Precision Floating-Point Library

The special functions and tools in this library can be used with MPFR (an arbitrary precision number type based on the GNU Multiple Precision Arithmetic Library), either via the bindings in boost/math/bindings/mpfr.hpp, or via boost/math/bindings/mpreal.hpp.

New projects are recommended to use Boost.Multiprecision with GMP/MPFR backend instead.

In order to use these bindings you will need to have installed MPFR plus its dependency the GMP library. You will also need one of the two supported C++ wrappers for MPFR: gmpfrxx (or mpfr_class), or mpfr-C++ (mpreal).

Unfortunately neither `mpfr_class` nor `mpreal` quite satisfy our conceptual requirements, so there is a very thin set of additional interfaces and some helper traits defined in boost/math/bindings/mpfr.hpp and boost/math/bindings/mpreal.hpp that you should use in place of including 'gmpfrxx.h' or 'mpreal.h' directly. The classes `mpfr_class` or `mpreal` are then usable unchanged once this header is included, so for example `mpfr_class`'s performance-enhancing expression templates are preserved and fully supported by this library:

```
#include <boost/math/bindings/mpfr.hpp>
#include <boost/math/special_functions/gamma.hpp>

int main()
{
   mpfr_class::set_dprec(500); // 500 bit precision
   //
   // Note that the argument to tgamma is
   // an expression template - that's just fine here.
   //
   mpfr_class v = boost::math::tgamma(sqrt(mpfr_class(2)));
   std::cout << std::setprecision(50) << v << std::endl;
}
```

Alternatively use with `mpreal` would look like:

```
#include <boost/math/bindings/mpreal.hpp>
#include <boost/math/special_functions/gamma.hpp>

int main()
{
   mpfr::mpreal::set_precision(500); // 500 bit precision
   mpfr::mpreal v = boost::math::tgamma(sqrt(mpfr::mpreal(2)));
   std::cout << std::setprecision(50) << v << std::endl;
}
```

For those functions that are based upon the Lanczos approximation, the bindings defines a series of approximations with up to 61 terms and accuracy up to approximately 3e-113. This therefore sets the upper limit for accuracy to the majority of functions defined this library when used with either `mpfr_class` or `mpreal`.

There is a concept checking test program for mpfr support here and here.

Using e_float Library

Boost.Multiprecision was a development from the e_float (TOMS Algorithm 910) library by Christopher Kormanyos.

e_float can still be used with Boost.Math library via the header:

```
<boost/math/bindings/e_float.hpp>
```

And the type `boost::math::ef::e_float`: this type is a thin wrapper class around ::e_float which provides the necessary syntactic sugar to make everything "just work".

There is also a concept checking test program for e_float support here.

New projects are recommended to use Boost.Multiprecision with `cpp_float` backend instead.

Using NTL Library

NTL::RR (an arbitrarily-fixed precision floating-point number type), can be used via the bindings in boost/math/bindings/rr.hpp. For details, see NTL: A Library for doing Number Theory by Victor Shoup.

New projects are recommended to use Boost.Multiprecision instead.

Unfortunately `NTL::RR` doesn't quite satisfy our conceptual requirements, so there is a very thin wrapper class `boost::math::ntl::RR` defined in boost/math/bindings/rr.hpp that you should use in place of `NTL::RR`. The class is intended to be a drop-in replacement for the "real" NTL::RR that adds some syntactic sugar to keep this library happy, plus some of the standard library functions not implemented in NTL.

For those functions that are based upon the Lanczos approximation, the bindings defines a series of approximations with up to 61 terms and accuracy up to approximately 3e-113. This therefore sets the upper limit for accuracy to the majority of functions defined this library when used with `NTL::RR`.

There is a concept checking test program for NTL support here.

Using without expression templates for Boost.Test and others

As noted in the Boost.Multiprecision documentation, certain program constructs will not compile when using expression templates. One example that many users may encounter is Boost.Test (1.54 and earlier) when using macro BOOST_CHECK_CLOSE and BOOST_CHECK_CLOSE_FRACTION.

If, for example, you wish to use any multiprecision type like `cpp_dec_float_50` in place of `double` to give more precision, you will need to override the default `boost::multiprecision::et_on` with `boost::multiprecision::et_off`.

```
#include <boost/multiprecision/cpp_dec_float.hpp>
```

To define a 50 decimal digit type using `cpp_dec_float`, you must pass two template parameters to `boost::multipreci-sion::number`.

It may be more legible to use a two-staged type definition such as this:

```
typedef boost::multiprecision::cpp_dec_float<50> mp_backend;
typedef boost::multiprecision::number<mp_backend, boost::multipreci↵
sion::et_off> cpp_dec_float_50_noet;
```

Here, we first define `mp_backend` as `cpp_dec_float` with 50 digits. The second step passes this backend to `boost::multi-precision::number` with `boost::multiprecision::et_off`, an enumerated type.

```
typedef boost::multiprecision::number<boost::multiprecision::cpp_dec_float<50>, boost::multi↵
precision::et_off>
cpp_dec_float_50_noet;
```

You can reduce typing with a `using` directive `using namespace boost::multiprecision;` if desired, as shown below.

```
using namespace boost::multiprecision;
```

Now `cpp_dec_float_50_noet` or `cpp_dec_float_50_et` can be used as a direct replacement for built-in types like `double` etc.

```
BOOST_AUTO_TEST_CASE(cpp_float_test_check_close_noet)
{ // No expression templates/
  typedef number<cpp_dec_float<50>, et_off> cpp_dec_float_50_noet;

  std::cout.precision(std::numeric_limits<cpp_dec_float_50_noet>::digits10); // All signific↵
ant digits.
  std::cout << std::showpoint << std::endl; // Show trailing zeros.

  cpp_dec_float_50_noet a ("1.0");
  cpp_dec_float_50_noet b ("1.0");
 b += std::numeric_limits<cpp_dec_float_50_noet>::epsilon(); // Increment least significant ↵
decimal digit.

  cpp_dec_float_50_noet eps = std::numeric_limits<cpp_dec_float_50_noet>::epsilon();

  std::cout <<"a = " << a << ",\nb = " << b << ",\neps = " << eps << std::endl;

  BOOST_CHECK_CLOSE(a, b, eps * 100); // Expected to pass (because tolerance is as percent).
 BOOST_CHECK_CLOSE_FRACTION(a, b, eps); // Expected to pass (because tolerance is as fraction).



} // BOOST_AUTO_TEST_CASE(cpp_float_test_check_close)

BOOST_AUTO_TEST_CASE(cpp_float_test_check_close_et)
{ // Using expression templates.
  typedef number<cpp_dec_float<50>, et_on> cpp_dec_float_50_et;

 std::cout.precision(std::numeric_limits<cpp_dec_float_50_et>::digits10); // All significant ↵
digits.
  std::cout << std::showpoint << std::endl; // Show trailing zeros.

  cpp_dec_float_50_et a("1.0");
  cpp_dec_float_50_et b("1.0");
  b += std::numeric_limits<cpp_dec_float_50_et>::epsilon(); // Increment least significant ↵
decimal digit.

  cpp_dec_float_50_et eps = std::numeric_limits<cpp_dec_float_50_et>::epsilon();

  std::cout << "a = " << a << ",\nb = " << b << ",\neps = " << eps << std::endl;

  BOOST_CHECK_CLOSE(a, b, eps * 100); // Expected to pass (because tolerance is as percent).
 BOOST_CHECK_CLOSE_FRACTION(a, b, eps); // Expected to pass (because tolerance is as fraction).
```

Using `cpp_dec_float_50` with the default expression template use switched on, the compiler error message for `BOOST_CHECK_CLOSE_FRACTION(a, b, eps); would be:

```
// failure floating_point_comparison.hpp(59): error C2440: 'static_cast' :
// cannot convert from 'int' to 'boost::multiprecision::detail::expres↵
sion<tag,Arg1,Arg2,Arg3,Arg4>'
```

A full example code is at test_cpp_float_close_fraction.cpp

Conceptual Requirements for Real Number Types

The functions and statistical distributions in this library can be used with any type *RealType* that meets the conceptual requirements given below. All the built-in floating-point types like `double` will meet these requirements. (Built-in types are also called fundamental (built-in) types).

User-defined types that meet the conceptual requirements can also be used. For example, with a thin wrapper class one of the types provided with NTL (RR) can be used. But now that Boost.Multiprecision library is available, this has become the preferred real-number type, typically cpp_dec_float or cpp_bin_float.

Submissions of binding to other extended precision types would also still be welcome.

The guiding principal behind these requirements is that a *RealType* behaves just like a built-in floating-point type.

Basic Arithmetic Requirements

These requirements are common to all of the functions in this library.

In the following table *r* is an object of type `RealType`, *cr* and *cr2* are objects of type `const RealType`, and *ca* is an object of type `const arithmetic-type` (arithmetic types include all the built in integers and floating point types).

Expression	Result Type	Notes
`RealType(cr)`	RealType	RealType is copy constructible.
`RealType(ca)`	RealType	RealType is copy constructible from the arithmetic types.
`r = cr`	RealType&	Assignment operator.
`r = ca`	RealType&	Assignment operator from the arithmetic types.
`r += cr`	RealType&	Adds cr to r.
`r += ca`	RealType&	Adds ar to r.
`r -= cr`	RealType&	Subtracts cr from r.
`r -= ca`	RealType&	Subtracts ca from r.
`r *= cr`	RealType&	Multiplies r by cr.
`r *= ca`	RealType&	Multiplies r by ca.
`r /= cr`	RealType&	Divides r by cr.
`r /= ca`	RealType&	Divides r by ca.
`-r`	RealType	Unary Negation.
`+r`	RealType&	Identity Operation.
`cr + cr2`	RealType	Binary Addition
`cr + ca`	RealType	Binary Addition
`ca + cr`	RealType	Binary Addition
`cr - cr2`	RealType	Binary Subtraction
`cr - ca`	RealType	Binary Subtraction
`ca - cr`	RealType	Binary Subtraction
`cr * cr2`	RealType	Binary Multiplication
`cr * ca`	RealType	Binary Multiplication
`ca * cr`	RealType	Binary Multiplication
`cr / cr2`	RealType	Binary Subtraction
`cr / ca`	RealType	Binary Subtraction
`ca / cr`	RealType	Binary Subtraction
`cr == cr2`	bool	Equality Comparison

Expression	Result Type	Notes
`cr == ca`	bool	Equality Comparison
`ca == cr`	bool	Equality Comparison
`cr != cr2`	bool	Inequality Comparison
`cr != ca`	bool	Inequality Comparison
`ca != cr`	bool	Inequality Comparison
`cr <= cr2`	bool	Less than equal to.
`cr <= ca`	bool	Less than equal to.
`ca <= cr`	bool	Less than equal to.
`cr >= cr2`	bool	Greater than equal to.
`cr >= ca`	bool	Greater than equal to.
`ca >= cr`	bool	Greater than equal to.
`cr < cr2`	bool	Less than comparison.
`cr < ca`	bool	Less than comparison.
`ca < cr`	bool	Less than comparison.
`cr > cr2`	bool	Greater than comparison.
`cr > ca`	bool	Greater than comparison.
`ca > cr`	bool	Greater than comparison.
`boost::math::tools::digits<RealType>()`	int	The number of digits in the significand of RealType.
`boost::math::tools::max_value<RealType>()`	RealType	The largest representable number by type RealType.
`boost::math::tools::min_value<RealType>()`	RealType	The smallest representable number by type RealType.
`boost::math::tools::log_max_value<RealType>()`	RealType	The natural logarithm of the largest representable number by type RealType.
`boost::math::tools::log_min_value<RealType>()`	RealType	The natural logarithm of the smallest representable number by type RealType.
`boost::math::tools::epsilon<RealType>()`	RealType	The machine epsilon of RealType.

Note that:

1. The functions `log_max_value` and `log_min_value` can be synthesised from the others, and so no explicit specialisation is required.

2. The function `epsilon` can be synthesised from the others, so no explicit specialisation is required provided the precision of RealType does not vary at runtime (see the header boost/math/bindings/rr.hpp for an example where the precision does vary at runtime).

3. The functions `digits`, `max_value` and `min_value`, all get synthesised automatically from `std::numeric_limits`. However, if `numeric_limits` is not specialised for type RealType, then you will get a compiler error when code tries to use these functions, *unless* you explicitly specialise them. For example if the precision of RealType varies at runtime, then `numeric_limits` support may not be appropriate, see boost/math/bindings/rr.hpp for examples.

Warning

If `std::numeric_limits<>` is **not specialized** for type *RealType* then the default float precision of 6 decimal digits will be used by other Boost programs including:

Boost.Test: giving misleading error messages like

"difference between {9.79796} and {9.79796} exceeds 5.42101e-19%".

Boost.LexicalCast and Boost.Serialization when converting the number to a string, causing potentially serious loss of accuracy on output.

Although it might seem obvious that RealType should require `std::numeric_limits` to be specialized, this is not sensible for `NTL::RR` and similar classes where the **number of digits is a runtime parameter** (whereas for `numeric_limits` everything has to be fixed at compile time).

Standard Library Support Requirements

Many (though not all) of the functions in this library make calls to standard library functions, the following table summarises the requirements. Note that most of the functions in this library will only call a small subset of the functions listed here, so if in doubt whether a user-defined type has enough standard library support to be useable the best advise is to try it and see!

In the following table *r* is an object of type `RealType`, *cr1* and *cr2* are objects of type `const RealType`, and *i* is an object of type `int`.

Expression	Result Type
`fabs(cr1)`	RealType
`abs(cr1)`	RealType
`ceil(cr1)`	RealType
`floor(cr1)`	RealType
`exp(cr1)`	RealType
`pow(cr1, cr2)`	RealType
`sqrt(cr1)`	RealType
`log(cr1)`	RealType
`frexp(cr1, &i)`	RealType
`ldexp(cr1, i)`	RealType
`cos(cr1)`	RealType
`sin(cr1)`	RealType
`asin(cr1)`	RealType
`tan(cr1)`	RealType
`atan(cr1)`	RealType
`fmod(cr1)`	RealType
`round(cr1)`	RealType
`iround(cr1)`	int
`trunc(cr1)`	RealType
`itrunc(cr1)`	int

Note that the table above lists only those standard library functions known to be used (or likely to be used in the near future) by this library. The following functions: `acos`, `atan2`, `fmod`, `cosh`, `sinh`, `tanh`, `log10`, `lround`, `llround`, `ltrunc`, `lltrunc` and `modf` are not currently used, but may be if further special functions are added.

Note that the `round`, `trunc` and `modf` functions are not part of the current C++ standard: they are part of the additions added to C99 which will likely be in the next C++ standard. There are Boost versions of these provided as a backup, and the functions are always called unqualified so that argument-dependent-lookup can take place.

In addition, for efficient and accurate results, a Lanczos approximation is highly desirable. You may be able to adapt an existing approximation from boost/math/special_functions/lanczos.hpp or boost/math/bindings/detail/big_lanczos.hpp: in the former case you will need change `static_cast`'s to `lexical_cast`'s, and the constants to *strings* (in order to ensure the coefficients aren't truncated to `long double`) and then specialise `lanczos_traits` for type T. Otherwise you may have to hack libs/math/tools/lanczos_generator.cpp to find a suitable approximation for your RealType. The code will still compile if you don't do this, but both accuracy and efficiency will be greatly compromised in any function that makes use of the gamma/beta/erf family of functions.

Conceptual Requirements for Distribution Types

A *DistributionType* is a type that implements the following conceptual requirements, and encapsulates a statistical distribution.

Please note that this documentation should not be used as a substitute for the reference documentation, and tutorial of the statistical distributions.

In the following table, *d* is an object of type `DistributionType`, *cd* is an object of type `const DistributionType` and *cr* is an object of a type convertible to `RealType`.

Expression	Result Type	Notes
DistributionType::value_type	RealType	The real-number type *RealType* upon which the distribution operates.
DistributionType::policy_type	RealType	The Policy to use when evaluating functions that depend on this distribution.
d = cd	Distribution&	Distribution types are assignable.
Distribution(cd)	Distribution	Distribution types are copy constructible.
pdf(cd, cr)	RealType	Returns the PDF of the distribution.
cdf(cd, cr)	RealType	Returns the CDF of the distribution.
cdf(complement(cd, cr))	RealType	Returns the complement of the CDF of the distribution, the same as: `1-cdf(cd, cr)`
quantile(cd, cr)	RealType	Returns the quantile (or percentile) of the distribution.
quantile(complement(cd, cr))	RealType	Returns the quantile (or percentile) of the distribution, starting from the complement of the probability, the same as: `quantile(cd, 1-cr)`
chf(cd, cr)	RealType	Returns the cumulative hazard function of the distribution.
hazard(cd, cr)	RealType	Returns the hazard function of the distribution.
kurtosis(cd)	RealType	Returns the kurtosis of the distribution.
kurtosis_excess(cd)	RealType	Returns the kurtosis excess of the distribution.
mean(cd)	RealType	Returns the mean of the distribution.
mode(cd)	RealType	Returns the mode of the distribution.
skewness(cd)	RealType	Returns the skewness of the distribution.
standard_deviation(cd)	RealType	Returns the standard deviation of the distribution.
variance(cd)	RealType	Returns the variance of the distribution.

Conceptual Archetypes for Reals and Distributions

There are a few concept archetypes available:

- Real concept for floating-point types.
- Distribution concept for statistical distributions.

Real concept

`std_real_concept` is an archetype for theReal types, including the built-in float, double, long double.

```
#include <boost/concepts/std_real_concept.hpp>
```

```
namespace boost{
namespace math{
namespace concepts
{
  class std_real_concept;
}
}} // namespaces
```

The main purpose in providing this type is to verify that standard library functions are found via a using declaration - bringing those functions into the current scope - and not just because they happen to be in global scope.

In order to ensure that a call to say `pow` can be found either via argument dependent lookup, or failing that then in the std namespace: all calls to standard library functions are unqualified, with the std:: versions found via a `using` declaration to make them visible in the current scope. Unfortunately it's all to easy to forget the `using` declaration, and call the double version of the function that happens to be in the global scope by mistake.

For example if the code calls ::pow rather than std::pow, the code will cleanly compile, but truncation of long doubles to double will cause a significant loss of precision. In contrast a template instantiated with std_real_concept will **only** compile if the all the standard library functions used have been brought into the current scope with a using declaration.

Testing the real concept

There is a test program libs/math/test/std_real_concept_check.cpp that instantiates every template in this library with type `std_real_concept` to verify its usage of standard library functions.

```
#include <boost/math/concepts/real_concept.hpp>
```

```
namespace boost{
namespace math{
namespace concepts{

class real_concept;

}}} // namespaces
```

`real_concept` is an archetype for user defined real types, it declares its standard library functions in its own namespace: these will only be found if they are called unqualified allowing argument dependent lookup to locate them. In addition this type is useable at runtime: this allows code that would not otherwise be exercised by the built-in floating point types to be tested. There is no std::numeric_limits<> support for this type, since numeric_limits is not a conceptual requirement for RealTypes.

NTL RR is an example of a type meeting the requirements that this type models, but note that use of a thin wrapper class is required: refer to "Using With NTL - a High-Precision Floating-Point Library".

There is no specific test case for type `real_concept`, instead, since this type is usable at runtime, each individual test case as well as testing `float`, `double` and `long double`, also tests `real_concept`.

Distribution Concept

Distribution Concept models statistical distributions.

```
#include <boost/math/concepts/distribution.hpp>
```

```
namespace boost{
namespace math{
namespace concepts
{
  template <class RealType>
  class distribution_archetype;

  template <class Distribution>
  struct DistributionConcept;

}}} // namespaces
```

The class template `distribution_archetype` is a model of the Distribution concept.

The class template `DistributionConcept` is a concept checking class for distribution types.

Testing the distribution concept

The test program distribution_concept_check.cpp is responsible for using `DistributionConcept` to verify that all the distributions in this library conform to the Distribution concept.

The class template `DistributionConcept` verifies the existence (but not proper function) of the non-member accessors required by the Distribution concept. These are checked by calls like

v = pdf(dist, x); // (Result v is ignored).

And in addition, those that accept two arguments do the right thing when the arguments are of different types (the result type is always the same as the distribution's value_type). (This is implemented by some additional forwarding-functions in derived_accessors.hpp, so that there is no need for any code changes. Likewise boilerplate versions of the hazard/chf/coefficient_of_variation functions are implemented in there too.)

Policies: Controlling Precision, Error Handling etc

Policy Overview

Policies are a powerful fine-grain mechanism that allow you to customise the behaviour of this library according to your needs. There is more information available in the policy tutorial and the policy reference.

Generally speaking, unless you find that the default policy behaviour when encountering 'bad' argument values does not meet your needs, you should not need to worry about policies.

Policies are a compile-time mechanism that allow you to change error-handling or calculation precision either program wide, or at the call site.

Although the policy mechanism itself is rather complicated, in practice it is easy to use, and very flexible.

Using policies you can control:

- How results from 'bad' arguments are handled, including those that cannot be fully evaluated.
- How accuracy is controlled by internal promotion to use more precise types.
- What working precision should be used to calculate results.
- What to do when a mathematically undefined function is used: Should this raise a run-time or compile-time error?
- Whether discrete functions, like the binomial, should return real or only integral values, and how they are rounded.
- How many iterations a special function is permitted to perform in a series evaluation or root finding algorithm before it gives up and raises an evaluation_error.

You can control policies:

- Using macros to change any default policy: the is the preferred method for installation wide policies.
- At your chosen namespace scope for distributions and/or functions: this is the preferred method for project, namespace, or translation unit scope policies.
- In an ad-hoc manner by passing a specific policy to a special function, or to a statistical distribution.

Policy Tutorial

So Just What is a Policy Anyway?

A policy is a compile-time mechanism for customising the behaviour of a special function, or a statistical distribution. With Policies you can control:

- What action to take when an error occurs.
- What happens when you call a function that is mathematically undefined (for example, if you ask for the mean of a Cauchy distribution).
- What happens when you ask for a quantile of a discrete distribution.
- Whether the library is allowed to internally promote `float` to `double` and `double` to `long double` in order to improve precision.
- What precision to use when calculating the result.

Some of these policies could arguably be run-time variables, but then we couldn't use compile-time dispatch internally to select the best evaluation method for the given policies.

For this reason a Policy is a *type*: in fact it's an instance of the class template `boost::math::policies::policy<>`. This class is just a compile-time-container of user-selected policies (sometimes called a type-list).

Over a dozen policy defaults are provided, so most of the time you can ignore the policy framework, but you can overwrite the defaults with your own policies to give detailed control, for example:

```
using namespace boost::math::policies;

// Define a policy that sets ::errno on overflow,
// and does not promote double to long double internally,
// and only aims for precision of only 3 decimal digits,
// to an error-handling policy, usually to trade precision for speed:

typedef policy
<
  domain_error<errno_on_error>,
  promote_double<false>,
  digits10<3>
> my_policy;
```

Policies Have Sensible Defaults

Most of the time you can just ignore the policy framework.

**The defaults for the various policies are as follows, if these work OK for you then you can stop reading now!*

Domain Error	Throws a `std::domain_error` exception.
Pole Error	Occurs when a function is evaluated at a pole: throws a `std::domain_error` exception.
Overflow Error	Throws a `std::overflow_error` exception.
Underflow	Ignores the underflow, and returns zero.
Denormalised Result	Ignores the fact that the result is denormalised, and returns it.
Rounding Error	Throws a `boost::math::rounding_error` exception.

Internal Evaluation Error	Throws a `boost::math::evaluation_error` exception.
Indeterminate Result Error	Returns a result that depends on the function where the error occurred.
Promotion of float to double	Does occur by default - gives full float precision results.
Promotion of double to long double	Does occur by default if long double offers more precision than double.
Precision of Approximation Used	By default uses an approximation that will result in the lowest level of error for the type of the result.
Behaviour of Discrete Quantiles	The quantile function will by default return an integer result that has been *rounded outwards*. That is to say lower quantiles (where the probability is less than 0.5) are rounded downward, and upper quantiles (where the probability is greater than 0.5) are rounded upwards. This behaviour ensures that if an X% quantile is requested, then *at least* the requested coverage will be present in the central region, and *no more than* the requested coverage will be present in the tails. This behaviour can be changed so that the quantile functions are rounded differently, or even return a real-valued result using Policies. It is strongly recommended that you read the tutorial Understanding Quantiles of Discrete Distributions before using the quantile function on a discrete distribution. The reference docs describe how to change the rounding policy for these distributions.

What's more, if you define your own policy type, then it automatically inherits the defaults for any policies not explicitly set, so given:

```
using namespace boost::math::policies;
//
// Define a policy that sets ::errno on overflow, and does
// not promote double to long double internally:

typedef policy
<
  domain_error<errno_on_error>,
  promote_double<false>
> my_policy;
```

then `my_policy` defines a policy where only the overflow error handling and `double`-promotion policies differ from the defaults.

We can also add a desired precision, for example, 9 bits or 3 decimal digits, to an error-handling policy, usually to trade precision for speed:

```
typedef policy<domain_error<errno_on_error>, digit2<9> > my_policy;
```

Or if you want to further modify an existing user policy, use `normalise`:

```
using boost::math::policies::normalise;

typedef normalise<my_policy, digits2<9>>::type my_policy_9; // errno on error, and limited pre↵
cision.
```

So How are Policies Used Anyway?

The details follow later, but basically policies can be set by either:

- Defining some macros that change the default behaviour: **this is the recommended method for setting installation-wide policies**.

- By instantiating a statistical distribution object with an explicit policy: this is mainly reserved for ad hoc policy changes.
- By passing a policy to a special function as an optional final argument: this is mainly reserved for ad hoc policy changes.
- By using some helper macros to define a set of functions or distributions in the current namespace that use a specific policy: **this is the recommended method for setting policies on a project- or translation-unit-wide basis**.

The following sections introduce these methods in more detail.

Changing the Policy Defaults

The default policies used by the library are changed by the usual configuration macro method.

For example, passing `-DBOOST_MATH_DOMAIN_ERROR_POLICY=errno_on_error` to your compiler will cause domain errors to set `::errno` and return a NaN rather than the usual default behaviour of throwing a `std::domain_error` exception.

Tip

For Microsoft Visual Studio,you can add to the Project Property Page, C/C++, Preprocessor, Preprocessor definitions like:

```
BOOST_MATH_ASSERT_UNDEFINED_POLICY=0
BOOST_MATH_OVERFLOW_ERROR_POLICY=errno_on_error
```

This may be helpful to avoid complications with pre-compiled headers that may mean that the equivalent definitions in source code:

```
#define BOOST_MATH_ASSERT_UNDEFINED_POLICY false
#define BOOST_MATH_OVERFLOW_ERROR_POLICY errno_on_error
```

- may be ignored*.

The compiler command line shows:

```
/D "BOOST_MATH_ASSERT_UNDEFINED_POLICY=0"
/D "BOOST_MATH_OVERFLOW_ERROR_POLICY=errno_on_error"
```

There is however a very important caveat to this:

Important

Default policies changed by setting configuration macros must be changed uniformly in every translation unit in the program.

Failure to follow this rule may result in violations of the "One Definition Rule (ODR)" and result in unpredictable program behaviour.

That means there are only two safe ways to use these macros:

- Edit them in boost/math/tools/user.hpp, so that the defaults are set on an installation-wide basis. Unfortunately this may not be convenient if you are using a pre-installed Boost distribution (on Linux for example).
- Set the defines in your project's Makefile or build environment, so that they are set uniformly across all translation units.

What you should **not** do is:

- Set the defines in the source file using `#define` as doing so almost certainly will break your program, unless you're absolutely certain that the program is restricted to a single translation unit.

And, yes, you will find examples in our test programs where we break this rule: but only because we know there will always be a single translation unit only: *don't say that you weren't warned!*

The following example demonstrates the effect of setting the macro BOOST_MATH_DOMAIN_ERROR_POLICY when an invalid argument is encountered. For the purposes of this example, we'll pass a negative degrees of freedom parameter to the student's t distribution.

Since we know that this is a single file program we could just add:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error
```

to the top of the source file to change the default policy to one that simply returns a NaN when a domain error occurs. Alternatively we could use:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

To ensure the `::errno` is set when a domain error occurs as well as returning a NaN.

This is safe provided the program consists of a single translation unit *and* we place the define *before* any #includes. Note that should we add the define after the includes then it will have no effect! A warning such as:

```
warning C4005: 'BOOST_MATH_OVERFLOW_ERROR_POLICY' : macro redefinition
```

is a certain sign that it will *not* have the desired effect.

We'll begin our sample program with the needed includes:

```
   #define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error

// Boost
#include <boost/math/distributions/students_t.hpp>
   using boost::math::students_t;  // Probability of students_t(df, t).

// std
#include <iostream>
   using std::cout;
   using std::endl;

#include <stdexcept>


#include <cstddef>
   // using ::errno
```

Next we'll define the program's main() to call the student's t distribution with an invalid degrees of freedom parameter, the program is set up to handle either an exception or a NaN:

```
int main()
{
   cout << "Example error handling using Student's t function. " << endl;
   cout << "BOOST_MATH_DOMAIN_ERROR_POLICY is set to: "
      << BOOST_STRINGIZE(BOOST_MATH_DOMAIN_ERROR_POLICY) << endl;

   double degrees_of_freedom = -1; // A bad argument!
   double t = 10;

   try
   {
      errno = 0; // Clear/reset.
      students_t dist(degrees_of_freedom); // exception is thrown here if enabled.
      double p = cdf(dist, t);
      // Test for error reported by other means:
      if((boost::math::isnan)(p))
      {
         cout << "cdf returned a NaN!" << endl;
         if (errno != 0)
         { // So errno has been set.
           cout << "errno is set to: " << errno << endl;
         }
      }
      else
         cout << "Probability of Student's t is " << p << endl;
   }
   catch(const std::exception& e)
   {
      std::cout <<
         "\n""Message from thrown exception was:\n   " << e.what() << std::endl;
   }
   return 0;
} // int main()
```

Here's what the program output looks like with a default build (one that **does throw exceptions**):

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: throw_on_error

Message from thrown exception was:
   Error in function boost::math::students_t_distribution<double>::students_t_distribution:
   Degrees of freedom argument is -1, but must be > 0 !
```

Alternatively let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error
```

Now the program output is:

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: ignore_error
cdf returned a NaN!
```

And finally let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

Which gives the output show errno:

```
Example error handling using Student's t function.
BOOST_MATH_DOMAIN_ERROR_POLICY is set to: errno_on_error
cdf returned a NaN!
errno is set to: 33
```

Setting Policies for Distributions on an Ad Hoc Basis

All of the statistical distributions in this library are class templates that accept two template parameters: real type (float, double ...) and policy (how to handle exceptional events), both with sensible defaults, for example:

```
namespace boost{ namespace math{

template <class RealType = double, class Policy = policies::policy<> >
class fisher_f_distribution;

typedef fisher_f_distribution<> fisher_f;

}}
```

This policy gets used by all the accessor functions that accept a distribution as an argument, and forwarded to all the functions called by these. So if you use the shorthand-typedef for the distribution, then you get `double` precision arithmetic and all the default policies.

However, say for example we wanted to evaluate the quantile of the binomial distribution at float precision, without internal promotion to double, and with the result rounded to the *nearest* integer, then here's how it can be done:

```
#include <boost/math/distributions/binomial.hpp>
using boost::math::binomial_distribution;

// Begin by defining a policy type, that gives the behaviour we want:

//using namespace boost::math::policies; or explicitly
using boost::math::policies::policy;

using boost::math::policies::promote_float;
using boost::math::policies::discrete_quantile;
using boost::math::policies::integer_round_nearest;

typedef policy<
   promote_float<false>, // Do not promote to double.
   discrete_quantile<integer_round_nearest> // Round result to nearest integer.
> mypolicy;
//
// Then define a new distribution that uses it:
typedef boost::math::binomial_distribution<float, mypolicy> mybinom;

//  And now use it to get the quantile:

int main()
{
   cout << "quantile(mybinom(200, 0.25), 0.05) is: " <<
      quantile(mybinom(200, 0.25), 0.05) << endl;
}
```

Which outputs:

```
quantile is: 40
```

Changing the Policy on an Ad Hoc Basis for the Special Functions

All of the special functions in this library come in two overloaded forms, one with a final "policy" parameter, and one without. For example:

```
namespace boost{ namespace math{

template <class RealType, class Policy>
RealType tgamma(RealType, const Policy&);

template <class RealType>
RealType tgamma(RealType);

}} // namespaces
```

Normally, the second version is just a forwarding wrapper to the first like this:

```
template <class RealType>
inline RealType tgamma(RealType x)
{
   return tgamma(x, policies::policy<>());
}
```

So calling a special function with a specific policy is just a matter of defining the policy type to use and passing it as the final parameter. For example, suppose we want `tgamma` to behave in a C-compatible fashion and set `::errno` when an error occurs, and never throw an exception:

```
#include <boost/math/special_functions/gamma.hpp>
using boost::math::tgamma;

// Define the policy to use:
using namespace boost::math::policies; // may be convenient, or

using boost::math::policies::policy;
// Types of error whose action can be altered by policies:.
using boost::math::policies::evaluation_error;
using boost::math::policies::domain_error;
using boost::math::policies::overflow_error;
using boost::math::policies::domain_error;
using boost::math::policies::pole_error;
// Actions on error (in enum error_policy_type):
using boost::math::policies::errno_on_error;
using boost::math::policies::ignore_error;
using boost::math::policies::throw_on_error;
using boost::math::policies::user_error;

typedef policy<
   domain_error<errno_on_error>,
   pole_error<errno_on_error>,
   overflow_error<errno_on_error>,
   evaluation_error<errno_on_error>
> c_policy;
//
// Now use the policy when calling tgamma:

// http://msdn.microsoft.com/en-us/library/t3ayayh1.aspx
// Microsoft errno declared in STDLIB.H as "extern int errno;"

int main()
{
   errno = 0; // Reset.
   cout << "Result of tgamma(30000) is: "
      << tgamma(30000, c_policy()) << endl; // Too big parameter
   cout << "errno = " << errno << endl; // errno 34 Numerical result out of range.
   cout << "Result of tgamma(-10) is: "
      << boost::math::tgamma(-10, c_policy()) << endl; // Negative parameter.
   cout << "errno = " << errno << endl; // error 33 Numerical argument out of domain.
} // int main()
```

which outputs:

```
Result of tgamma(30000) is: 1.#INF
errno = 34
Result of tgamma(-10) is: 1.#QNAN
errno = 33
```

Alternatively, for ad hoc use, we can use the `make_policy` helper function to create a policy for us: this usage is more verbose, so is probably only preferred when a policy is going to be used once only:

```
#include <boost/math/special_functions/gamma.hpp>
using boost::math::tgamma;

int main()
{
   // using namespace boost::math::policies; // or
   using boost::math::policies::errno_on_error;
   using boost::math::policies::make_policy;
   using boost::math::policies::pole_error;
   using boost::math::policies::domain_error;
   using boost::math::policies::overflow_error;
   using boost::math::policies::evaluation_error;

   errno = 0;
   std::cout << "Result of tgamma(30000) is: "
      << boost::math::tgamma(
         30000,
         make_policy(
            domain_error<errno_on_error>(),
            pole_error<errno_on_error>(),
            overflow_error<errno_on_error>(),
            evaluation_error<errno_on_error>()
         )
      ) << std::endl;
   // Check errno was set:
   std::cout << "errno = " << errno << std::endl;
   // and again with evaluation at a pole:
   std::cout << "Result of tgamma(-10) is: "
      << boost::math::tgamma(
         -10,
         make_policy(
            domain_error<errno_on_error>(),
            pole_error<errno_on_error>(),
            overflow_error<errno_on_error>(),
            evaluation_error<errno_on_error>()
         )
      ) << std::endl;
   // Check errno was set:
   std::cout << "errno = " << errno << std::endl;
}
```

Setting Policies at Namespace or Translation Unit Scope

Sometimes what you want to do is just change a set of policies within the current scope: **the one thing you should not do in this situation is use the configuration macros**, as this can lead to "One Definition Rule" violations. Instead this library provides a pair of macros especially for this purpose.

Let's consider the special functions first: we can declare a set of forwarding functions that all use a specific policy using the macro BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(*Policy*). This macro should be used either inside a unique namespace set aside for the purpose (for example, a C namespace for a C-style policy), or an unnamed namespace if you just want the functions visible in global scope for the current file only.

Suppose we want `C::foo()` to behave in a C-compatible way and set `::errno` on error rather than throwing any exceptions.

We'll begin by including the needed header for our function:

```
#include <boost/math/special_functions.hpp>
//using boost::math::tgamma; // Not needed because using C::tgamma.
```

Open up the "C" namespace that we'll use for our functions, and define the policy type we want: in this case a C-style one that sets ::errno and returns a standard value, rather than throwing exceptions.

Any policies we don't specify here will inherit the defaults.

```
namespace C
{ // To hold our C-style policy.
  //using namespace boost::math::policies; or explicitly:
  using boost::math::policies::policy;

  using boost::math::policies::domain_error;
  using boost::math::policies::pole_error;
  using boost::math::policies::overflow_error;
  using boost::math::policies::evaluation_error;
  using boost::math::policies::errno_on_error;

  typedef policy<
     domain_error<errno_on_error>,
     pole_error<errno_on_error>,
     overflow_error<errno_on_error>,
     evaluation_error<errno_on_error>
  > c_policy;
```

All we need do now is invoke the BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS macro passing our policy type c_policy as the single argument:

```
BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(c_policy)

} // close namespace C
```

We now have a set of forwarding functions defined in namespace C that all look something like this:

```
template <class RealType>
inline typename boost::math::tools::promote_args<RT>::type
   tgamma(RT z)
{
   return boost::math::tgamma(z, c_policy());
}
```

So that when we call `C::tgamma(z)`, we really end up calling `boost::math::tgamma(z, C::c_policy())`:

```
int main()
{
   errno = 0;
   cout << "Result of tgamma(30000) is: "
      << C::tgamma(30000) << endl; // Note using C::tgamma
   cout << "errno = " << errno << endl; // errno = 34
   cout << "Result of tgamma(-10) is: "
      << C::tgamma(-10) << endl;
   cout << "errno = " << errno << endl; // errno = 33, overwriting previous value of 34.
}
```

Which outputs:

```
Result of C::tgamma(30000) is: 1.#INF
errno = 34
Result of C::tgamma(-10) is: 1.#QNAN
errno = 33
```

This mechanism is particularly useful when we want to define a project-wide policy, and don't want to modify the Boost source, or to set project wide build macros (possibly fragile and easy to forget).

The same mechanism works well at file scope as well, by using an unnamed namespace, we can ensure that these declarations don't conflict with any alternate policies present in other translation units:

```
#include <boost/math/special_functions.hpp>
// using boost::math::tgamma; // Would create an ambiguity between
// 'double boost::math::tgamma<int>(T)' and
// 'double 'anonymous-namespace'::tgamma<int>(RT)'.

namespace mymath
{ // unnamed

using namespace boost::math::policies;

typedef policy<
   domain_error<errno_on_error>,
   pole_error<errno_on_error>,
   overflow_error<errno_on_error>,
   evaluation_error<errno_on_error>
> c_policy;

BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(c_policy)
```

So that when we call `mymath::tgamma(z)`, we really end up calling `boost::math::tgamma(z, anonymous-namespace::c_policy())`.

```
} // close unnamed namespace

int main()
{
   errno = 0;
   cout << "Result of tgamma(30000) is: "
      << mymath::tgamma(30000) << endl;
      // tgamma in unnamed namespace in this translation unit (file) only.
   cout << "errno = " << errno << endl;
   cout << "Result of tgamma(-10) is: "
      << mymath::tgamma(-10) << endl;
   cout << "errno = " << errno << endl;
   // Default tgamma policy would throw an exception, and abort.
}
```

Handling policies for the statistical distributions is very similar except that now the macro BOOST_MATH_DECLARE_DISTRIBUTIONS accepts two parameters: the floating point type to use, and the policy type to apply. For example:

```
BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)
```

Results a set of typedefs being defined like this:

```
typedef boost::math::normal_distribution<double, my_policy> normal;
```

The name of each typedef is the same as the name of the distribution class template, but without the "_distribution" suffix.

Suppose we want a set of distributions to behave as follows:

- Return infinity on overflow, rather than throwing an exception.
- Don't perform any promotion from double to long double internally.
- Return the closest integer result from the quantiles of discrete distributions.

We'll begin by including the needed header for all the distributions:

```
#include <boost/math/distributions.hpp>
```

Open up an appropriate namespace, calling it `my_distributions`, for our distributions, and define the policy type we want. Any policies we don't specify here will inherit the defaults:

```
namespace my_distributions
{
  using namespace boost::math::policies;
  // using boost::math::policies::errno_on_error; // etc.

  typedef policy<
     // return infinity and set errno rather than throw:
     overflow_error<errno_on_error>,
     // Don't promote double -> long double internally:
     promote_double<false>,
     // Return the closest integer result for discrete quantiles:
     discrete_quantile<integer_round_nearest>
  > my_policy;
```

All we need do now is invoke the BOOST_MATH_DECLARE_DISTRIBUTIONS macro passing the floating point type `double` and policy types `my_policy` as arguments:

```
BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)

} // close namespace my_namespace
```

We now have a set of typedefs defined in namespace my_distributions that all look something like this:

```
typedef boost::math::normal_distribution<double, my_policy> normal;
typedef boost::math::cauchy_distribution<double, my_policy> cauchy;
typedef boost::math::gamma_distribution<double, my_policy> gamma;
// etc
```

So that when we use my_distributions::normal we really end up using `boost::math::normal_distribution<double, my_policy>`:

```
int main()
{
   // Construct distribution with something we know will overflow
  // (using double rather than if promoted to long double):
   my_distributions::normal norm(10, 2);

   errno = 0;
   cout << "Result of quantile(norm, 0) is: "
      << quantile(norm, 0) << endl; // -infinity.
   cout << "errno = " << errno << endl;
   errno = 0;
   cout << "Result of quantile(norm, 1) is: "
      << quantile(norm, 1) << endl; // +infinity.
   cout << "errno = " << errno << endl;

   // Now try a discrete distribution.
   my_distributions::binomial binom(20, 0.25);
   cout << "Result of quantile(binom, 0.05) is: "
      << quantile(binom, 0.05) << endl; // To check we get integer results.
   cout << "Result of quantile(complement(binom, 0.05)) is: "
      << quantile(complement(binom, 0.05)) << endl;
}
```

Which outputs:

```
Result of quantile(norm, 0) is: -1.#INF
errno = 34
Result of quantile(norm, 1) is: 1.#INF
errno = 34
Result of quantile(binom, 0.05) is: 1
Result of quantile(complement(binom, 0.05)) is: 8
```

This mechanism is particularly useful when we want to define a project-wide policy, and don't want to modify the Boost source or set project wide build macros (possibly fragile and easy to forget).

Note

There is an important limitation to note: you can *not use the macros BOOST_MATH_DECLARE_DISTRIBUTIONS and BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS *in the same namespace**, as doing so creates ambiguities between functions and distributions of the same name.

As before, the same mechanism works well at file scope as well: by using an unnamed namespace, we can ensure that these declarations don't conflict with any alternate policies present in other translation units:

```
#include <boost/math/distributions.hpp> // All distributions.
// using boost::math::normal; // Would create an ambguity between
// boost::math::normal_distribution<RealType> boost::math::normal and
// 'anonymous-namespace'::normal'.

namespace
{ // anonymous or unnnamed (rather than named as in policy_eg_6.cpp).

  using namespace boost::math::policies;
   // using boost::math::policies::errno_on_error; // etc.
  typedef policy<
     // return infinity and set errno rather than throw:
     overflow_error<errno_on_error>,
     // Don't promote double -> long double internally:
     promote_double<false>,
     // Return the closest integer result for discrete quantiles:
     discrete_quantile<integer_round_nearest>
  > my_policy;

  BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)

} // close namespace my_namespace

int main()
{
   // Construct distribution with something we know will overflow.
   normal norm(10, 2); // using 'anonymous-namespace'::normal
   errno = 0;
   cout << "Result of quantile(norm, 0) is: "
      << quantile(norm, 0) << endl;
   cout << "errno = " << errno << endl;
   errno = 0;
   cout << "Result of quantile(norm, 1) is: "
      << quantile(norm, 1) << endl;
   cout << "errno = " << errno << endl;
   //
   // Now try a discrete distribution:
   binomial binom(20, 0.25);
   cout << "Result of quantile(binom, 0.05) is: "
      << quantile(binom, 0.05) << endl;
   cout << "Result of quantile(complement(binom, 0.05)) is: "
      << quantile(complement(binom, 0.05)) << endl;
}
```

Calling User Defined Error Handlers

Suppose we want our own user-defined error handlers rather than the any of the default ones supplied by the library to be used. If we set the policy for a specific type of error to `user_error` then the library will call a user-supplied error handler. These are forward declared, but not defined in boost/math/policies/error_handling.hpp like this:

```
namespace boost{ namespace math{ namespace policies{

template <class T>
T user_domain_error(const char* function, const char* message, const T& val);
template <class T>
T user_pole_error(const char* function, const char* message, const T& val);
template <class T>
T user_overflow_error(const char* function, const char* message, const T& val);
template <class T>
T user_underflow_error(const char* function, const char* message, const T& val);
template <class T>
T user_denorm_error(const char* function, const char* message, const T& val);
template <class T>
T user_evaluation_error(const char* function, const char* message, const T& val);
template <class T, class TargetType>
T user_rounding_error(const char* function, const char* message, const T& val, const Target↵
Type& t);
template <class T>
T user_indeterminate_result_error(const char* function, const char* message, const T& val);

}}} // namespaces
```

So out first job is to include the header we want to use, and then provide definitions for our user-defined error handlers that we want to use. We only provide our special domain and pole error handlers; other errors like overflow and underflow use the default.

```
#include <boost/math/special_functions.hpp>

namespace boost{ namespace math
{
  namespace policies
  {
    template <class T>
    T user_domain_error(const char* function, const char* message, const T& val)
    { // Ignoring function, message and val for this example, perhaps unhelpfully.
       cerr << "Domain Error!" << endl;
       return std::numeric_limits<T>::quiet_NaN();
    }

    template <class T>
    T user_pole_error(const char* function, const char* message, const T& val)
    { // Ignoring function, message and val for this example, perhaps unhelpfully.
       cerr << "Pole Error!" << endl;
       return std::numeric_limits<T>::quiet_NaN();
    }
  } // namespace policies
}} // namespace boost{ namespace math
```

Now we'll need to define a suitable policy that will call these handlers, and define some forwarding functions that make use of the policy:

```
namespace mymath{

using namespace boost::math::policies;

typedef policy<
   domain_error<user_error>,
   pole_error<user_error>
> user_error_policy;

BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(user_error_policy)

} // close unnamed namespace
```

We now have a set of forwarding functions defined in namespace mymath that all look something like this:

```
template <class RealType>
inline typename boost::math::tools::promote_args<RT>::type
   tgamma(RT z)
{
   return boost::math::tgamma(z, user_error_policy());
}
```

So that when we call `mymath::tgamma(z)` we really end up calling `boost::math::tgamma(z, user_error_policy())`, and any errors will get directed to our own error handlers.

```
int main()
{
   cout << "Result of erf_inv(-10) is: "
      << mymath::erf_inv(-10) << endl;
   cout << "Result of tgamma(-10) is: "
      << mymath::tgamma(-10) << endl;
}
```

Which outputs:

```
  Domain Error!
  Pole Error!
  Result of erf_inv(-10) is: 1.#QNAN
  Result of tgamma(-10) is: 1.#QNAN
```

The previous example was all well and good, but the custom error handlers didn't really do much of any use. In this example we'll implement all the custom handlers and show how the information provided to them can be used to generate nice formatted error messages.

Each error handler has the general form:

```
template <class T>
T user_error_type(
   const char* function,
   const char* message,
   const T& val);
```

and accepts three arguments:

const char* function	The name of the function that raised the error, this string contains one or more %1% format specifiers that should be replaced by the name of real type T, like float or double.

const char* message	A message associated with the error, normally this contains a %1% format specifier that should be replaced with the value of *value*: however note that overflow and underflow messages do not contain this %1% specifier (since the value of *value* is immaterial in these cases).
const T& value	The value that caused the error: either an argument to the function if this is a domain or pole error, the tentative result if this is a denorm or evaluation error, or zero or infinity for underflow or overflow errors.

As before we'll include the headers we need first:

```
#include <boost/math/special_functions.hpp>
```

Next we'll implement our own error handlers for each type of error, starting with domain errors:

```
namespace boost{ namespace math{
namespace policies
{

template <class T>
T user_domain_error(const char* function, const char* message, const T& val)
{
```

We'll begin with a bit of defensive programming in case function or message are empty:

```
if(function == 0)
    function = "Unknown function with arguments of type %1%";
if(message == 0)
    message = "Cause unknown with bad argument %1%";
```

Next we'll format the name of the function with the name of type T, perhaps double:

```
std::string msg("Error in function ");
msg += (boost::format(function) % typeid(T).name()).str();
```

Then likewise format the error message with the value of parameter *val*, making sure we output all the potentially significant digits of *val*:

```
msg += ": \n";
int prec = 2 + (std::numeric_limits<T>::digits * 30103UL) / 100000UL;
// int prec = std::numeric_limits<T>::max_digits10; //  For C++0X Standard Library
msg += (boost::format(message) % boost::io::group(std::setprecision(prec), val)).str();
```

Now we just have to do something with the message, we could throw an exception, but for the purposes of this example we'll just dump the message to std::cerr:

```
std::cerr << msg << std::endl;
```

Finally the only sensible value we can return from a domain error is a NaN:

```
   return std::numeric_limits<T>::quiet_NaN();
}
```

Pole errors are essentially a special case of domain errors, so in this example we'll just return the result of a domain error:

```
template <class T>
T user_pole_error(const char* function, const char* message, const T& val)
{
   return user_domain_error(function, message, val);
}
```

Overflow errors are very similar to domain errors, except that there's no %1% format specifier in the *message* parameter:

```
template <class T>
T user_overflow_error(const char* function, const char* message, const T& val)
{
   if(function == 0)
       function = "Unknown function with arguments of type %1%";
   if(message == 0)
       message = "Result of function is too large to represent";

   std::string msg("Error in function ");
   msg += (boost::format(function) % typeid(T).name()).str();

   msg += ": \n";
   msg += message;

   std::cerr << msg << std::endl;

   // Value passed to the function is an infinity, just return it:
   return val;
}
```

Underflow errors are much the same as overflow:

```
template <class T>
T user_underflow_error(const char* function, const char* message, const T& val)
{
   if(function == 0)
       function = "Unknown function with arguments of type %1%";
   if(message == 0)
       message = "Result of function is too small to represent";

   std::string msg("Error in function ");
   msg += (boost::format(function) % typeid(T).name()).str();

   msg += ": \n";
   msg += message;

   std::cerr << msg << std::endl;

   // Value passed to the function is zero, just return it:
   return val;
}
```

Denormalised results are much the same as underflow:

```
template <class T>
T user_denorm_error(const char* function, const char* message, const T& val)
{
   if(function == 0)
       function = "Unknown function with arguments of type %1%";
   if(message == 0)
       message = "Result of function is denormalised";

   std::string msg("Error in function ");
   msg += (boost::format(function) % typeid(T).name()).str();

   msg += ": \n";
   msg += message;

   std::cerr << msg << std::endl;

   // Value passed to the function is denormalised, just return it:
   return val;
}
```

Which leaves us with evaluation errors: these occur when an internal error occurs that prevents the function being fully evaluated. The parameter *val* contains the closest approximation to the result found so far:

```
template <class T>
T user_evaluation_error(const char* function, const char* message, const T& val)
{
   if(function == 0)
       function = "Unknown function with arguments of type %1%";
   if(message == 0)
       message = "An internal evaluation error occurred with "
                  "the best value calculated so far of %1%";

   std::string msg("Error in function ");
   msg += (boost::format(function) % typeid(T).name()).str();

   msg += ": \n";
   int prec = 2 + (std::numeric_limits<T>::digits * 30103UL) / 100000UL;
   // int prec = std::numeric_limits<T>::max_digits10; // For C++0X Standard Library
   msg += (boost::format(message) % boost::io::group(std::setprecision(prec), val)).str();

   std::cerr << msg << std::endl;

   // What do we return here?  This is generally a fatal error, that should never occur,
   // so we just return a NaN for the purposes of the example:
   return std::numeric_limits<T>::quiet_NaN();
}

} // policies
}} // boost::math
```

Now we'll need to define a suitable policy that will call these handlers, and define some forwarding functions that make use of the policy:

```
namespace mymath
{ // unnamed.

using namespace boost::math::policies;

typedef policy<
   domain_error<user_error>,
   pole_error<user_error>,
   overflow_error<user_error>,
   underflow_error<user_error>,
   denorm_error<user_error>,
   evaluation_error<user_error>
> user_error_policy;

BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(user_error_policy)

} // unnamed namespace
```

We now have a set of forwarding functions, defined in namespace mymath, that all look something like this:

```
template <class RealType>
inline typename boost::math::tools::promote_args<RT>::type
   tgamma(RT z)
{
   return boost::math::tgamma(z, user_error_policy());
}
```

So that when we call `mymath::tgamma(z)` we really end up calling `boost::math::tgamma(z, user_error_policy())`, and any errors will get directed to our own error handlers:

```
int main()
{
   // Raise a domain error:
   cout << "Result of erf_inv(-10) is: "
      << mymath::erf_inv(-10) << std::endl << endl;
   // Raise a pole error:
   cout << "Result of tgamma(-10) is: "
      << mymath::tgamma(-10) << std::endl << endl;
   // Raise an overflow error:
   cout << "Result of tgamma(3000) is: "
      << mymath::tgamma(3000) << std::endl << endl;
   // Raise an underflow error:
   cout << "Result of tgamma(-190.5) is: "
      << mymath::tgamma(-190.5) << std::endl << endl;
   // Unfortunately we can't predicably raise a denormalised
   // result, nor can we raise an evaluation error in this example
   // since these should never really occur!
} // int main()
```

Which outputs:

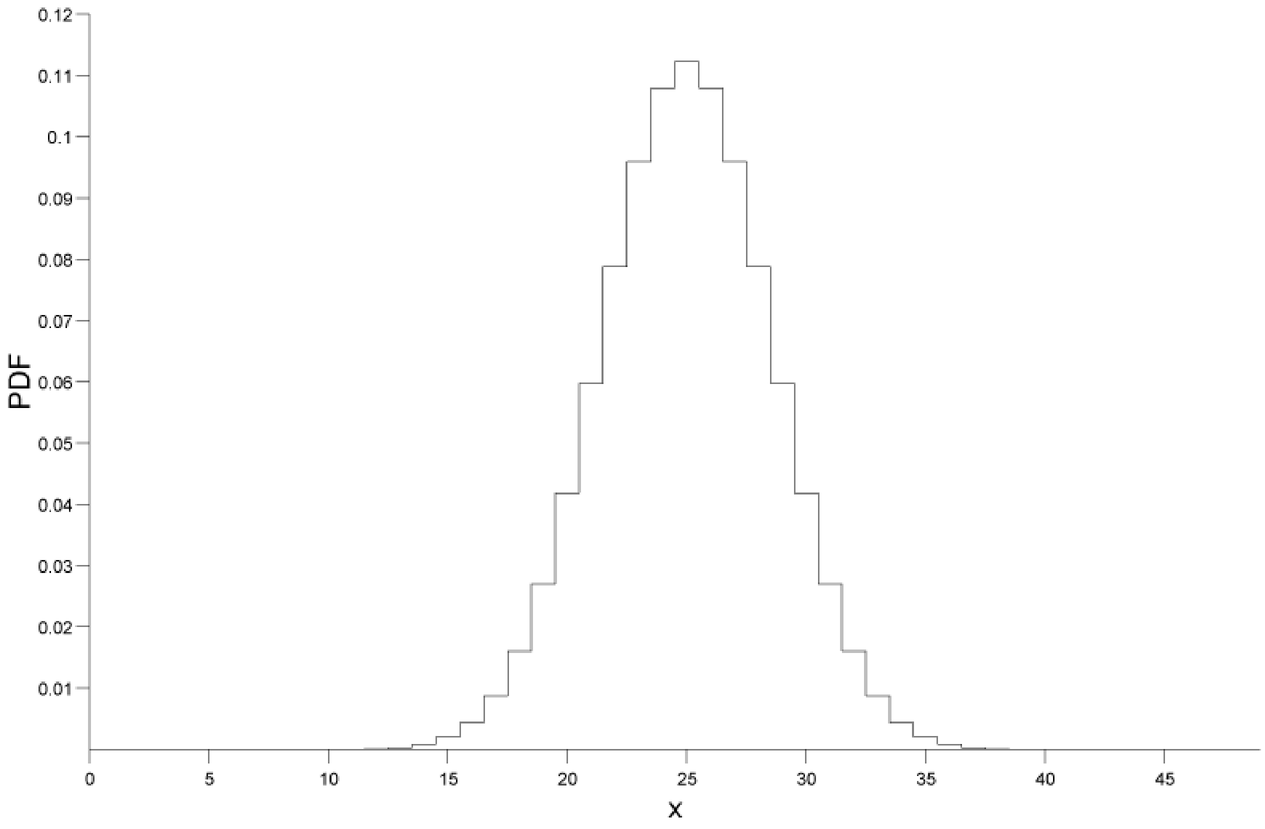
```
Error in function boost::math::erf_inv<double>(double, double):
Argument outside range [-1, 1] in inverse erf function (got p=-10).
Result of erf_inv(-10) is: 1.#QNAN

Error in function boost::math::tgamma<long double>(long double):
Evaluation of tgamma at a negative integer -10.
Result of tgamma(-10) is: 1.#QNAN

Error in function boost::math::tgamma<long double>(long double):
Result of tgamma is too large to represent.
Error in function boost::math::tgamma<double>(double):
Result of function is too large to represent
Result of tgamma(3000) is: 1.#INF

Error in function boost::math::tgamma<long double>(long double):
Result of tgamma is too large to represent.
Error in function boost::math::tgamma<long double>(long double):
Result of tgamma is too small to represent.
Result of tgamma(-190.5) is: 0
```

Notice how some of the calls result in an error handler being called more than once, or for more than one handler to be called: this is an artefact of the fact that many functions are implemented in terms of one or more sub-routines each of which may have it's own error handling. For example `tgamma(-190.5)` is implemented in terms of `tgamma(190.5)` - which overflows - the reflection formula for `tgamma` then notices that it is dividing by infinity and so underflows.

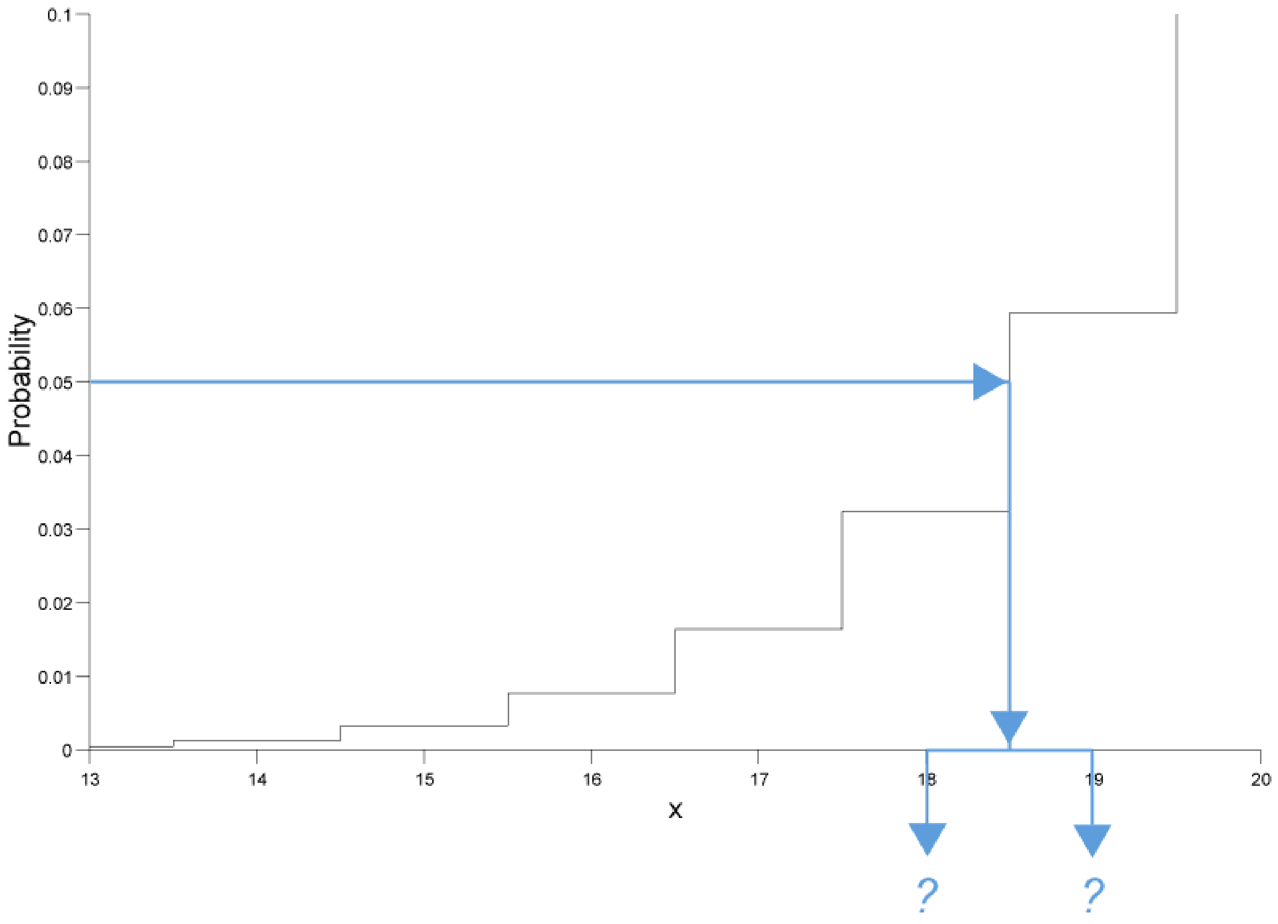
Understanding Quantiles of Discrete Distributions

Discrete distributions present us with a problem when calculating the quantile: we are starting from a continuous real-valued variable - the probability - but the result (the value of the random variable) should really be discrete.

Consider for example a Binomial distribution, with a sample size of 50, and a success fraction of 0.5. There are a variety of ways we can plot a discrete distribution, but if we plot the PDF as a step-function then it looks something like this:



Now lets suppose that the user asks for a the quantile that corresponds to a probability of 0.05, if we zoom in on the CDF for that region here's what we see:



As can be seen there is no random variable that corresponds to a probability of exactly 0.05, so we're left with two choices as shown in the figure:

- We could round the result down to 18.
- We could round the result up to 19.

In fact there's actually a third choice as well: we could "pretend" that the distribution was continuous and return a real valued result: in this case we would calculate a result of approximately 18.701 (this accurately reflects the fact that the result is nearer to 19 than 18).

By using policies we can offer any of the above as options, but that still leaves the question: *What is actually the right thing to do?*

And in particular: *What policy should we use by default?*

In coming to an answer we should realise that:

- Calculating an integer result is often much faster than calculating a real-valued result: in fact in our tests it was up to 20 times faster.
- Normally people calculate quantiles so that they can perform a test of some kind: *"If the random variable is less than N then we can reject our null-hypothesis with 90% confidence."*

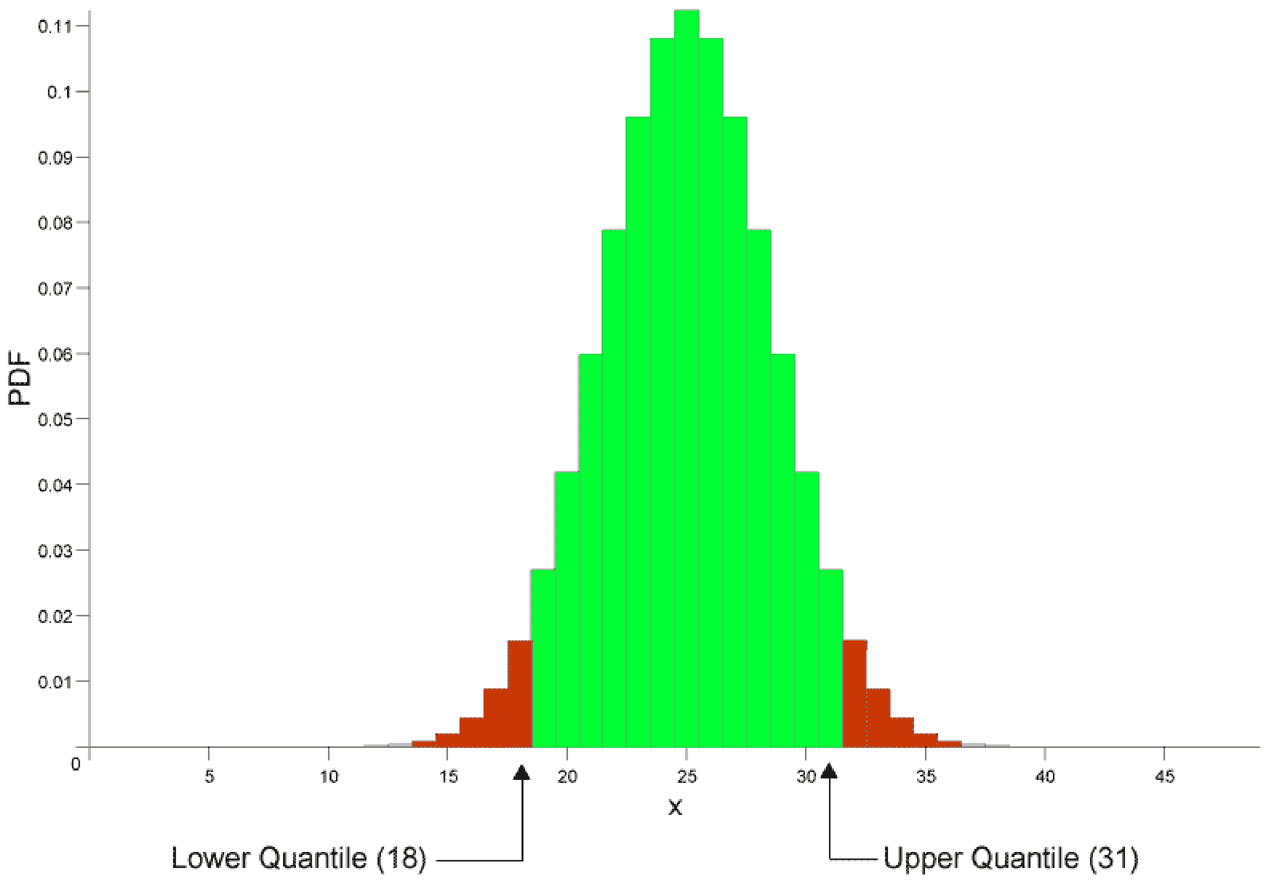
So there is a genuine benefit to calculating an integer result as well as it being "the right thing to do" from a philosophical point of view. What's more if someone asks for a quantile at 0.05, then we can normally assume that they are asking for ***at least** 95% of the probability to the right of the value chosen, and **no more than** 5% of the probability to the left of the value chosen.*

In the above binomial example we would therefore round the result down to 18.

The converse applies to upper-quantiles: If the probability is greater than 0.5 we would want to round the quantile up, *so that **at least** the requested probability is to the left of the value returned, and **no more than** 1 - the requested probability is to the right of the value returned.*

Likewise for two-sided intervals, we would round lower quantiles down, and upper quantiles up. This ensures that we have *at least the requested probability in the central region* and *no more than 1 minus the requested probability in the tail areas.*

For example, taking our 50 sample binomial distribution with a success fraction of 0.5, if we wanted a two sided 90% confidence interval, then we would ask for the 0.05 and 0.95 quantiles with the results *rounded outwards* so that *at least 90% of the probability* is in the central area:



So far so good, but there is in fact a trap waiting for the unwary here:

```
quantile(binomial(50, 0.5), 0.05);
```

returns 18 as the result, which is what we would expect from the graph above, and indeed there is no x greater than 18 for which:

```
cdf(binomial(50, 0.5), x) <= 0.05;
```

However:

```
quantile(binomial(50, 0.5), 0.95);
```

returns 31, and indeed while there is no x less than 31 for which:

```
cdf(binomial(50, 0.5), x) >= 0.95;
```

We might naively expect that for this symmetrical distribution the result would be 32 (since 32 = 50 - 18), but we need to remember that the cdf of the binomial is *inclusive* of the random variable. So while the left tail area *includes* the quantile returned, the right tail area always excludes an upper quantile value: since that "belongs" to the central area.

Look at the graph above to see what's going on here: the lower quantile of 18 belongs to the left tail, so any value <= 18 is in the left tail. The upper quantile of 31 on the other hand belongs to the central area, so the tail area actually starts at 32, so any value > 31 is in the right tail.

Therefore if U and L are the upper and lower quantiles respectively, then a random variable X is in the tail area - where we would reject the null hypothesis if:

```
X <= L || X > U
```

And the a variable X is inside the central region if:

```
L < X <= U
```

The moral here is to *always be very careful with your comparisons when dealing with a discrete distribution*, and if in doubt, *base your comparisons on CDF's instead*.

Other Rounding Policies are Available

As you would expect from a section on policies, you won't be surprised to know that other rounding options are available:

integer_round_outwards	This is the default policy as described above: lower quantiles are rounded down (probability < 0.5), and upper quantiles (probability > 0.5) are rounded up. This gives *no more than* the requested probability in the tails, and *at least* the requested probability in the central area.
integer_round_inwards	This is the exact opposite of the default policy: lower quantiles are rounded up (probability < 0.5), and upper quantiles (probability > 0.5) are rounded down. This gives *at least* the requested probability in the tails, and *no more than* the requested probability in the central area.
integer_round_down	This policy will always round the result down no matter whether it is an upper or lower quantile
integer_round_up	This policy will always round the result up no matter whether it is an upper or lower quantile
integer_round_nearest	This policy will always round the result to the nearest integer no matter whether it is an upper or lower quantile
real	This policy will return a real valued result for the quantile of a discrete distribution: this is generally much slower than finding an integer result but does allow for more sophisticated rounding policies.

To understand how the rounding policies for the discrete distributions can be used, we'll use the 50-sample binomial distribution with a success fraction of 0.5 once again, and calculate all the possible quantiles at 0.05 and 0.95.

Begin by including the needed headers (and some using statements for conciseness):

```
#include <iostream>
using std::cout; using std::endl;
using std::left; using std::fixed; using std::right; using std::scientific;
#include <iomanip>
using std::setw;
using std::setprecision;

#include <boost/math/distributions/binomial.hpp>
```

Next we'll bring the needed declarations into scope, and define distribution types for all the available rounding policies:

```
// Avoid
// using namespace std; // and
// using namespace boost::math;
// to avoid potential ambiguity of names, like binomial.
// using namespace boost::math::policies; is small risk, but
// the necessary items are brought into scope thus:

using boost::math::binomial_distribution;
using boost::math::policies::policy;
using boost::math::policies::discrete_quantile;

using boost::math::policies::integer_round_outwards;
using boost::math::policies::integer_round_down;
using boost::math::policies::integer_round_up;
using boost::math::policies::integer_round_nearest;
using boost::math::policies::integer_round_inwards;
using boost::math::policies::real;

using boost::math::binomial_distribution; // Not std::binomial_distribution.

typedef binomial_distribution<
            double,
            policy<discrete_quantile<integer_round_outwards> > >
        binom_round_outwards;

typedef binomial_distribution<
            double,
            policy<discrete_quantile<integer_round_inwards> > >
        binom_round_inwards;

typedef binomial_distribution<
            double,
            policy<discrete_quantile<integer_round_down> > >
        binom_round_down;

typedef binomial_distribution<
            double,
            policy<discrete_quantile<integer_round_up> > >
        binom_round_up;

typedef binomial_distribution<
            double,
            policy<discrete_quantile<integer_round_nearest> > >
        binom_round_nearest;

typedef binomial_distribution<
            double,
            policy<discrete_quantile<real> > >
        binom_real_quantile;
```

Now let's set to work calling those quantiles:

```
int main()
{
   cout <<
      "Testing rounding policies for a 50 sample binomial distribution,\n"
      "with a success fraction of 0.5.\n\n"
      "Lower quantiles are calculated at p = 0.05\n\n"
      "Upper quantiles at p = 0.95.\n\n";

   cout << setw(25) << right
      << "Policy"<< setw(18) << right
      << "Lower Quantile" << setw(18) << right
      << "Upper Quantile" << endl;

   // Test integer_round_outwards:
   cout << setw(25) << right
      << "integer_round_outwards"
      << setw(18) << right
      << quantile(binom_round_outwards(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_round_outwards(50, 0.5), 0.95)
      << endl;

   // Test integer_round_inwards:
   cout << setw(25) << right
      << "integer_round_inwards"
      << setw(18) << right
      << quantile(binom_round_inwards(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_round_inwards(50, 0.5), 0.95)
      << endl;

   // Test integer_round_down:
   cout << setw(25) << right
      << "integer_round_down"
      << setw(18) << right
      << quantile(binom_round_down(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_round_down(50, 0.5), 0.95)
      << endl;

   // Test integer_round_up:
   cout << setw(25) << right
      << "integer_round_up"
      << setw(18) << right
      << quantile(binom_round_up(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_round_up(50, 0.5), 0.95)
      << endl;

   // Test integer_round_nearest:
   cout << setw(25) << right
      << "integer_round_nearest"
      << setw(18) << right
      << quantile(binom_round_nearest(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_round_nearest(50, 0.5), 0.95)
      << endl;

   // Test real:
   cout << setw(25) << right
      << "real"
      << setw(18) << right
      << quantile(binom_real_quantile(50, 0.5), 0.05)
      << setw(18) << right
      << quantile(binom_real_quantile(50, 0.5), 0.95)
      << endl;
} // int main()
```

Which produces the program output:

```
policy_eg_10.vcxproj -> J:\Cpp\MathToolkit\test\Math_test\Release\policy_eg_10.exe
Testing rounding policies for a 50 sample binomial distribution,
with a success fraction of 0.5.

Lower quantiles are calculated at p = 0.05

Upper quantiles at p = 0.95.

                   Policy    Lower Quantile    Upper Quantile
   integer_round_outwards                18                31
    integer_round_inwards                19                30
       integer_round_down                18                30
         integer_round_up                19                31
    integer_round_nearest                19                30
                     real            18.701            30.299
```

Policy Reference

Error Handling Policies

There are two orthogonal aspects to error handling:

- What to do (if anything) with the error.
- What kind of error is being raised.

Available Actions When an Error is Raised

What to do with the error is encapsulated by an enumerated type:

```
namespace boost { namespace math { namespace policies {

enum error_policy_type
{
   throw_on_error = 0, // throw an exception.
   errno_on_error = 1, // set ::errno & return 0, NaN, infinity or best guess.
   ignore_error = 2, // return 0, NaN, infinity or best guess.
   user_error = 3  // call a user-defined error handler.
};

}}} // namespaces
```

The various enumerated values have the following meanings:

throw_on_error

Will throw one of the following exceptions, depending upon the type of the error:

Error Type	Exception
Domain Error	std::domain_error
Pole Error	std::domain_error
Overflow Error	std::overflow_error
Underflow Error	std::underflow_error
Denorm Error	std::underflow_error
Evaluation Error	boost::math::evaluation_error
Indeterminate Result Error	std::domain_error

errno_on_error

Will set global `::errno` `::errno` to one of the following values depending upon the error type (often EDOM = 33 and ERANGE = 34), and then return the same value as if the error had been ignored:

Error Type	errno value
Domain Error	EDOM
Pole Error	EDOM
Overflow Error	ERANGE
Underflow Error	ERANGE
Denorm Error	ERANGE
Evaluation Error	EDOM
Indeterminate Result Error	EDOM

ignore_error

Will return one of the values below depending on the error type (`::errno` is NOT changed)::

Error Type	Returned Value
Domain Error	std::numeric_limits<T>::quiet_NaN()
Pole Error	std::numeric_limits<T>::quiet_NaN()
Overflow Error	std::numeric_limits<T>::infinity()
Underflow Error	0
Denorm Error	The denormalised value.
Evaluation Error	The best guess (perhaps NaN) as to the result: which may be significantly in error.
Indeterminate Result Error	Depends on the function where the error occurred

user_error

Will call a user defined error handler: these are forward declared in boost/math/policies/error_handling.hpp, but the actual definitions must be provided by the user:

```
namespace boost{ namespace math{ namespace policies{

template <class T>
T user_domain_error(const char* function, const char* message, const T& val);

template <class T>
T user_pole_error(const char* function, const char* message, const T& val);

template <class T>
T user_overflow_error(const char* function, const char* message, const T& val);

template <class T>
T user_underflow_error(const char* function, const char* message, const T& val);

template <class T>
T user_denorm_error(const char* function, const char* message, const T& val);

template <class T>
T user_rounding_error(const char* function, const char* message, const T& val);

template <class T>
T user_evaluation_error(const char* function, const char* message, const T& val);

template <class T>
T user_indeterminate_result_error(const char* function, const char* message, const T& val);

}}} // namespaces
```

Note that the strings *function* and *message* may contain "%1%" format specifiers designed to be used in conjunction with Boost.Format. If these strings are to be presented to the program's end-user then the "%1%" format specifier should be replaced with the name of type T in the *function* string, and if there is a %1% specifier in the *message* string then it should be replaced with the value of *val*.

There is more information on user-defined error handlers in the tutorial here.

Kinds of Error Raised

There are six kinds of error reported by this library, which are summarised in the following table:

Error Type	Policy Class	Description
Domain Error	boost::math::policies::domain_error<*action*>	Raised when more or more arguments are outside the defined range of the function. Defaults to `boost::math::policies::domain_error<throw_on_error>` When the action is set to *throw_on_error* then throws `std::domain_error`
Pole Error	boost::math::policies::pole_error<*action*>	Raised when more or more arguments would cause the function to be evaluated at a pole. Defaults to `boost::math::policies::pole_error<throw_on_error>` When the action is *throw_on_error* then throw a `std::domain_error`
Overflow Error	boost::math::policies::overflow_error<*action*>	Raised when the result of the function is outside the representable range of the floating point type used. Defaults to `boost::math::policies::overflow_error<throw_on_error>`. When the action is *throw_on_error* then throws a `std::overflow_error`.
Underflow Error	boost::math::policies::underflow_error<*action*>	Raised when the result of the function is too small to be represented in the floating point type used. Defaults to `boost::math::policies::underflow_error<ignore_error>` When the specified action is *throw_on_error* then throws a `std::underflow_error`
Denorm Error	boost::math::policies::denorm_error<*action*>	Raised when the result of the function is a denormalised value. Defaults to `boost::math::policies::denorm_error<ignore_error>` When the action is *throw_on_error* then throws a `std::underflow_error`

Error Type	**Policy Class**	**Description**
Rounding Error	boost::math::policies::rounding_error<*action*>	Raised When one of the rounding functions round, trunc or modf is called with an argument that has no integer representation, or is too large to be represented in the result type Defaults to `boost::math::policies::rounding_error<throw_on_error>` When the action is *throw_on_error* then throws `boost::math::rounding_error`
Evaluation Error	boost::math::policies::evaluation_error<*action*>	Raised when the result of the function is well defined and finite, but we were unable to compute it. Typically this occurs when an iterative method fails to converge. Of course ideally this error should never be raised: feel free to report it as a bug if it is! Defaults to `boost::math::policies::evaluation_error<throw_on_error>` When the action is *throw_on_error* then throws `boost::math::evaluation_error`
Indeterminate Result Error	boost::math::policies::indeterminate_result_error<*action*>	Raised when the result of a function is not defined for the values that were passed to it. Defaults to `boost::math::policies::indeterminate_result_error<ignore_error>` When the action is *throw_on_error* then throws `std::domain_error`

Examples

Suppose we want a call to `tgamma` to behave in a C-compatible way and set global `::errno` rather than throw an exception, we can achieve this at the call site using:

```
#include <boost/math/special_functions/gamma.hpp>
using boost::math::tgamma;

//using namespace boost::math::policies; may also be convenient.
using boost::math::policies::policy;
using boost::math::policies::evaluation_error;
using boost::math::policies::domain_error;
using boost::math::policies::overflow_error;
using boost::math::policies::domain_error;
using boost::math::policies::pole_error;
using boost::math::policies::errno_on_error;

// Define a policy:
typedef policy<
  domain_error<errno_on_error>,
  pole_error<errno_on_error>,
  overflow_error<errno_on_error>,
  evaluation_error<errno_on_error>
> my_policy;

double my_value = 0.; //

// Call the function applying my_policy:
double t1 = tgamma(my_value, my_policy());

// Alternatively (and equivalently) we could use helpful function
// make_policy and define everything at the call site:
double t2 = tgamma(my_value,
  make_policy(
    domain_error<errno_on_error>(),
    pole_error<errno_on_error>(),
    overflow_error<errno_on_error>(),
    evaluation_error<errno_on_error>() )
  );
```

Suppose we want a statistical distribution to return infinities, rather than throw exceptions, then we can use:

```
#include <boost/math/distributions/normal.hpp>
using boost::math::normal_distribution;

using namespace boost::math::policies;

// Define a specific policy:
typedef policy<
      overflow_error<ignore_error>
      > my_policy;

// Define the distribution, using my_policy:
typedef normal_distribution<double, my_policy> my_norm;

// Construct a my_norm distribution, using default mean and standard deviation,
// and get a 0.05 or 5% quantile:
double q = quantile(my_norm(), 0.05); // = -1.64485
```

Internal Floating-point Promotion Policies

Normally when evaluating a function at say `float` precision, maximal accuracy is assured by conducting the calculation at `double` precision internally, and then rounding the result. There are two policies that control whether internal promotion to a higher precision floating-point type takes place, or not:

Policy	Meaning
boost::math::policies::promote_float<B>	Indicates whether float arguments should be promoted to double precision internally: defaults to boost::math::policies::promote_float<true>
boost::math::policies::promote_double<B>	Indicates whether double arguments should be promoted to long double precision internally: defaults to boost::math::policies::promote_double<true>

Examples

Suppose we want tgamma to be evaluated without internal promotion to long double, then we could use:

```
#include <boost/math/special_functions/gamma.hpp>

using namespace boost::math::policies;
using boost::math::tgamma;

// Define a new policy *not* internally promoting RealType to double:
typedef policy<
      promote_double<false>
      > my_policy;

// Call the function, applying the new policy:
double t1 = tgamma(some_value, my_policy());

// Alternatively we could use helper function make_policy,
// and concisely define everything at the call site:
double t2 = tgamma(some_value, make_policy(promote_double<false>()));
```

Alternatively, suppose we want a distribution to perform calculations without promoting float to double, then we could use:

```
#include <boost/math/distributions/normal.hpp>
using boost::math::normal_distribution;

using namespace boost::math::policies;

// Define a policy:
typedef policy<
      promote_float<false>
      > my_policy;

// Define the new normal distribution using my_policy:
typedef normal_distribution<float, my_policy> my_norm;

// Get a quantile:
float q = quantile(my_norm(), 0.05f);
```

Mathematically Undefined Function Policies

There are some functions that are generic (they are present for all the statistical distributions supported) but which may be mathematically undefined for certain distributions, but defined for others.

For example, the Cauchy distribution does not have a meaningful mean, so what should

```
mean(cauchy<>());
```

return, and should such an expression even compile at all?

The default behaviour is for all such functions to not compile at all - in fact they will raise a static assertion - but by changing the policy we can have them return the result of a domain error instead (which may well throw an exception, depending on the error handling policy).

This behaviour is controlled by the `assert_undefined<>` policy:

```
namespace boost{ namespace math{ namespace policies {

template <bool b>
class assert_undefined;

}}} //namespaces
```

For example:

```
#include <boost/math/distributions/cauchy.hpp>

using namespace boost::math::policies;
using namespace boost::math;

// This will not compile, cauchy has no mean!
double m1 = mean(cauchy());

// This will compile, but raises a domain error!
double m2 = mean(cauchy_distribution<double, policy<assert_undefined<false> > >());
```

`policy<assert_undefined<false>` behaviour can also be obtained by defining the macro

```
#define BOOST_MATH_ASSERT_UNDEFINED_POLICY false
```

at the head of the file - see Using Macros to Change the Policy Defaults.

Discrete Quantile Policies

If a statistical distribution is *discrete* then the random variable can only have integer values - this leaves us with a problem when calculating quantiles - we can either ignore the discreteness of the distribution and return a real value, or we can round to an integer. As it happens, computing integer values can be substantially faster than calculating a real value, so there are definite advantages to returning an integer, but we do then need to decide how best to round the result. The `discrete_quantile` policy defines how discrete quantiles work, and how integer results are rounded:

```
enum discrete_quantile_policy_type
{
   real,
   integer_round_outwards, // default
   integer_round_inwards,
   integer_round_down,
   integer_round_up,
   integer_round_nearest
};

template <discrete_quantile_policy_type>
struct discrete_quantile;
```

The values that `discrete_quantile` can take have the following meanings:

real

Ignores the discreteness of the distribution, and returns a real-valued result. For example:

```
#include <boost/math/distributions/negative_binomial.hpp>
using boost::math::negative_binomial_distribution;

using namespace boost::math::policies;

typedef negative_binomial_distribution<
      double,
      policy<discrete_quantile<real> >
   > dist_type;

// Lower 5% quantile:
double x = quantile(dist_type(20, 0.3), 0.05);
// Upper 95% quantile:
double y = quantile(complement(dist_type(20, 0.3), 0.05));
```

Results in `x = 27.3898` and `y = 68.1584`.

integer_round_outwards

This is the default policy: an integer value is returned so that:

- Lower quantiles (where the probability is less than 0.5) are rounded down.
- Upper quantiles (where the probability is greater than 0.5) are rounded up.

This is normally the safest rounding policy, since it ensures that both one and two sided intervals are guaranteed to have *at least* the requested coverage. For example:

```
#include <boost/math/distributions/negative_binomial.hpp>
using boost::math::negative_binomial;

// Use the default rounding policy integer_round_outwards.
// Lower quantile rounded down:
double x = quantile(negative_binomial(20, 0.3), 0.05); // rounded up 27 from 27.3898
// Upper quantile rounded up:
double y = quantile(complement(negative_binomial(20, 0.3), 0.05)); // rounded down to 69 from ↵
68.1584
```

Results in `x = 27` (rounded down from 27.3898) and `y = 69` (rounded up from 68.1584).

The variables x and y are now defined so that:

```
cdf(negative_binomial(20), x) <= 0.05
cdf(negative_binomial(20), y) >= 0.95
```

In other words we guarantee *at least 90% coverage in the central region overall*, and also *no more than 5% coverage in each tail*.

integer_round_inwards

This is the opposite of *integer_round_outwards*: an integer value is returned so that:

- Lower quantiles (where the probability is less than 0.5) are rounded *up*.
- Upper quantiles (where the probability is greater than 0.5) are rounded *down*.

For example:

```
#include <boost/math/distributions/negative_binomial.hpp>
using boost::math::negative_binomial_distribution;

using namespace boost::math::policies;

typedef negative_binomial_distribution<
      double,
      policy<discrete_quantile<integer_round_inwards> >
   > dist_type;

// Lower quantile rounded up:
double x = quantile(dist_type(20, 0.3), 0.05); // 28 rounded up from 27.3898
// Upper quantile rounded down:
double y = quantile(complement(dist_type(20, 0.3), 0.05)); // 68 rounded down from 68.1584
```

Results in `x = 28` (rounded up from 27.3898) and `y = 68` (rounded down from 68.1584).

The variables x and y are now defined so that:

```
cdf(negative_binomial(20), x) >= 0.05
cdf(negative_binomial(20), y) <= 0.95
```

In other words we guarantee *at no more than 90% coverage in the central region overall*, and also *at least 5% coverage in each tail*.

integer_round_down

Always rounds down to an integer value, no matter whether it's an upper or a lower quantile.

integer_round_up

Always rounds up to an integer value, no matter whether it's an upper or a lower quantile.

integer_round_nearest

Always rounds to the nearest integer value, no matter whether it's an upper or a lower quantile. This will produce the requested coverage *in the average case*, but for any specific example may results in either significantly more or less coverage than the requested amount. For example:

For example:

```
#include <boost/math/distributions/negative_binomial.hpp>
using boost::math::negative_binomial_distribution;

using namespace boost::math::policies;

typedef negative_binomial_distribution<
      double,
      policy<discrete_quantile<integer_round_nearest> >
   > dist_type;

// Lower quantile rounded (down) to nearest:
double x = quantile(dist_type(20, 0.3), 0.05); // 27
// Upper quantile rounded (down) to nearest:
double y = quantile(complement(dist_type(20, 0.3), 0.05)); // 68
```

Results in `x = 27` (rounded from 27.3898) and `y = 68` (rounded from 68.1584).

Precision Policies

There are two equivalent policies that effect the *working precision* used to calculate results, these policies both default to 0 - meaning calculate to the maximum precision available in the type being used - but can be set to other values to cause lower levels of precision to be used. One might want to trade precision for evaluation speed.

```
namespace boost{ namespace math{ namespace policies{

template <int N>
digits10;

template <int N>
digits2;

}}} // namespaces
```

As you would expect, *digits10* specifies the number of decimal digits to use, and *digits2* the number of binary digits. Internally, whichever is used, the precision is always converted to *binary digits*.

These policies are specified at compile-time, because many of the special functions use compile-time-dispatch to select which approximation to use based on the precision requested and the numeric type being used.

For example we could calculate `tgamma` to approximately 5 decimal digits using:

```
#include <boost/math/special_functions/gamma.hpp>
using boost::math::tgamma;
using boost::math::policies::policy;
using boost::math::policies::digits10;

typedef policy<digits10<5> > my_pol_5; // Define a new, non-default, policy
// to calculate tgamma to accuracy of approximately 5 decimal digits.
```

Or again using helper function `make_policy`:

```
#include <boost/math/special_functions/gamma.hpp>
using boost::math::tgamma;

using namespace boost::math::policies;

double t = tgamma(12, policy<digits10<5> >());  // Concise make_policy.
```

And for a quantile of a distribution to approximately 25-bit precision:

```
#include <boost/math/distributions/normal.hpp>
using boost::math::normal_distribution;

using namespace boost::math::policies;

const int bits = 25; // approximate precision.

double q = quantile(
      normal_distribution<double, policy<digits2<bits> > >(),
      0.05); // 5% quantile.
```

Iteration Limits Policies

There are two policies that effect the iterative algorithms used to implement the special functions in this library:

```
template <unsigned long limit = BOOST_MATH_MAX_SERIES_ITERATION_POLICY>
class max_series_iterations;

template <unsigned long limit = BOOST_MATH_MAX_ROOT_ITERATION_POLICY>
class max_root_iterations;
```

The class `max_series_iterations` determines the maximum number of iterations permitted in a series evaluation, before the special function gives up and returns the result of evaluation_error.

The class `max_root_iterations` determines the maximum number of iterations permitted in a root-finding algorithm before the special function gives up and returns the result of evaluation_error.

Using Macros to Change the Policy Defaults

You can use the various macros below to change any (or all) of the policies.

You can make a local change by placing a macro definition **before** a function or distribution #include.

Caution

There is a danger of One-Definition-Rule violations if you add ad-hoc macros to more than one source files: these must be set the same in **every translation unit**.

Caution

If you place it after the #include it will have no effect, (and it will affect only any other following #includes). This is probably not what you intend!

If you want to alter the defaults for any or all of the policies for **all** functions and distributions, installation-wide, then you can do so by defining various macros in boost/math/tools/user.hpp.

BOOST_MATH_DOMAIN_ERROR_POLICY

Defines what happens when a domain error occurs, if not defined then defaults to `throw_on_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_POLE_ERROR_POLICY

Defines what happens when a pole error occurs, if not defined then defaults to `throw_on_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_OVERFLOW_ERROR_POLICY

Defines what happens when an overflow error occurs, if not defined then defaults to `throw_on_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_ROUNDING_ERROR_POLICY

Defines what happens when a rounding error occurs, if not defined then defaults to `throw_on_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_EVALUATION_ERROR_POLICY

Defines what happens when an internal evaluation error occurs, if not defined then defaults to `throw_on_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_UNDERFLOW_ERROR_POLICY

Defines what happens when an overflow error occurs, if not defined then defaults to `ignore_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_DENORM_ERROR_POLICY

Defines what happens when a denormalisation error occurs, if not defined then defaults to `ignore_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_INDETERMINATE_RESULT_ERROR_POLICY

Defines what happens when the result is indeterminate, but where there is none the less a convention for the result. If not defined then defaults to `ignore_error`, but can be set to any of the enumerated actions for error handing: `throw_on_error`, `errno_on_error`, `ignore_error` or `user_error`.

BOOST_MATH_DIGITS10_POLICY

Defines how many decimal digits to use in internal computations: defaults to `0` - meaning use all available digits - but can be set to some other decimal value. Since setting this is likely to have a substantial impact on accuracy, it's not generally recommended that you change this from the default.

BOOST_MATH_PROMOTE_FLOAT_POLICY

Determines whether `float` types get promoted to `double` internally to ensure maximum precision in the result, defaults to `true`, but can be set to `false` to turn promotion of `float`'s off.

BOOST_MATH_PROMOTE_DOUBLE_POLICY

Determines whether `double` types get promoted to `long double` internally to ensure maximum precision in the result, defaults to `true`, but can be set to `false` to turn promotion of `double`'s off.

BOOST_MATH_DISCRETE_QUANTILE_POLICY

Determines how discrete quantiles return their results: either as an integer, or as a real value, can be set to one of the enumerated values: `real`, `integer_round_outwards`, `integer_round_inwards`, `integer_round_down`, `integer_round_up`, `integer_round_nearest`. Defaults to `integer_round_outwards`.

BOOST_MATH_ASSERT_UNDEFINED_POLICY

Determines whether functions that are mathematically undefined for a specific distribution compile or raise a static (i.e. compile-time) assertion. Defaults to `true`: meaning that any mathematically undefined function will not compile. When set to `false` then the function will compile but return the result of a domain error: this can be useful for some generic code, that needs to work with all distributions and determine at runtime whether or not a particular property is well defined.

BOOST_MATH_MAX_SERIES_ITERATION_POLICY

Determines how many series iterations a special function is permitted to perform before it gives up and returns an evaluation_error: Defaults to 1000000.

BOOST_MATH_MAX_ROOT_ITERATION_POLICY

Determines how many root-finding iterations a special function is permitted to perform before it gives up and returns an evaluation_error: Defaults to 200.

Example

Suppose we want overflow errors to set `::errno` and return an infinity, discrete quantiles to return a real-valued result (rather than round to integer), and for mathematically undefined functions to compile, but return a domain error. Then we could add the following to boost/math/tools/user.hpp:

```
#define BOOST_MATH_OVERFLOW_ERROR_POLICY errno_on_error
#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real
#define BOOST_MATH_ASSERT_UNDEFINED_POLICY false
```

or we could place these definitions **before**

```
#include <boost/math/distributions/normal.hpp>
  using boost::math::normal_distribution;
```

in a source .cpp file.

Setting Polices at Namespace Scope

Sometimes what you really want to do is bring all the special functions, or all the distributions into a specific namespace-scope, along with a specific policy to use with them. There are two macros defined to assist with that:

```
BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(Policy)
```

and:

```
BOOST_MATH_DECLARE_DISTRIBUTIONS(Type, Policy)
```

You can use either of these macros after including any special function or distribution header. For example:

```
#include <boost/math/special_functions/gamma.hpp>
//using boost::math::tgamma;
// Need not declare using boost::math::tgamma here,
// because will define tgamma in myspace using macro below.

namespace myspace
{
  using namespace boost::math::policies;

  // Define a policy that does not throw on overflow:
  typedef policy<overflow_error<errno_on_error> > my_policy;

  // Define the special functions in this scope to use the policy:
  BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(my_policy)
}

// Now we can use myspace::tgamma etc.
// They will automatically use "my_policy":
//
double t = myspace::tgamma(30.0); // Will *not* throw on overflow,
// despite the large value of factorial 30 = 265252859812191058636308480000000
// unlike default policy boost::math::tgamma;
```

In this example, using BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS results in a set of thin inline forwarding functions being defined:

```
template <class T>
inline T tgamma(T a){ return ::boost::math::tgamma(a, mypolicy()); }

template <class T>
inline T lgamma(T a) ( return ::boost::math::lgamma(a, mypolicy()); }
```

and so on. Note that while a forwarding function is defined for all the special functions, however, unless you include the specific header for the special function you use (or boost/math/special_functions.hpp to include everything), you will get linker errors from functions that are forward declared, but not defined.

We can do the same thing with the distributions, but this time we need to specify the floating-point type to use:

```
#include <boost/math/distributions/cauchy.hpp>

namespace myspace
{ // using namespace boost::math::policies; // May be convenient in myspace.

  // Define a policy called my_policy to use.
  using boost::math::policies::policy;

// In this case we want all the distribution accessor functions to compile,
// even if they are mathematically undefined, so
// make the policy assert_undefined.
  using boost::math::policies::assert_undefined;

typedef policy<assert_undefined<false> > my_policy;

// Finally apply this policy to type double.
BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)
} // namespace myspace

// Now we can use myspace::cauchy etc, which will use policy
// myspace::mypolicy:
//
// This compiles but throws a domain error exception at runtime.
// Caution! If you omit the try'n'catch blocks,
// it will just silently terminate, giving no clues as to why!
// So try'n'catch blocks are very strongly recommended.

void test_cauchy()
{
   try
   {
      double d = mean(myspace::cauchy());  // Cauchy does not have a mean!
      (void) d;
   }
   catch(const std::domain_error& e)
   {
      cout << e.what() << endl;
   }
}
```

In this example the result of BOOST_MATH_DECLARE_DISTRIBUTIONS is to declare a typedef for each distribution like this:

```
typedef boost::math::cauchy_distribution<double, my_policy> cauchy;
tyepdef boost::math::gamma_distribution<double, my_policy> gamma;
```

and so on. The name given to each typedef is the name of the distribution with the "_distribution" suffix removed.

Policy Class Reference

There's very little to say here, the `policy` class is just a rag-bag compile-time container for a collection of policies:

```
#include <boost/math/policies/policy.hpp>
```

```
namespace boost{
namespace math{
namespace policies

template <class A1 = default_policy,
          class A2 = default_policy,
          class A3 = default_policy,
          class A4 = default_policy,
          class A5 = default_policy,
          class A6 = default_policy,
          class A7 = default_policy,
          class A8 = default_policy,
          class A9 = default_policy,
          class A10 = default_policy,
          class A11 = default_policy,
          class A12 = default_policy,
          class A13 = default_policy>
struct policy
{
public:
   typedef computed-from-template-arguments domain_error_type;
   typedef computed-from-template-arguments pole_error_type;
   typedef computed-from-template-arguments overflow_error_type;
   typedef computed-from-template-arguments underflow_error_type;
   typedef computed-from-template-arguments denorm_error_type;
   typedef computed-from-template-arguments rounding_error_type;
   typedef computed-from-template-arguments evaluation_error_type;
   typedef computed-from-template-arguments indeterminate_result_error_type;
   typedef computed-from-template-arguments precision_type;
   typedef computed-from-template-arguments promote_float_type;
   typedef computed-from-template-arguments promote_double_type;
   typedef computed-from-template-arguments discrete_quantile_type;
   typedef computed-from-template-arguments assert_undefined_type;
};

template <...argument list...>
typename normalise<policy<>, A1>::type make_policy(...argument list..);

template <class Policy,
          class A1 = default_policy,
          class A2 = default_policy,
          class A3 = default_policy,
          class A4 = default_policy,
          class A5 = default_policy,
          class A6 = default_policy,
          class A7 = default_policy,
          class A8 = default_policy,
          class A9 = default_policy,
          class A10 = default_policy,
          class A11 = default_policy,
          class A12 = default_policy,
          class A13 = default_policy>
struct normalise
{
   typedef computed-from-template-arguments type;
};
```

The member typedefs of class `policy` are intended for internal use but are documented briefly here for the sake of completeness.

```
policy<...>::domain_error_type
```

Specifies how domain errors are handled, will be an instance of `boost::math::policies::domain_error<>` with the template argument to `domain_error` one of the `error_policy_type` enumerated values.

```
policy<...>::pole_error_type
```

Specifies how pole-errors are handled, will be an instance of `boost::math::policies::pole_error<>` with the template argument to `pole_error` one of the `error_policy_type` enumerated values.

```
policy<...>::overflow_error_type
```

Specifies how overflow errors are handled, will be an instance of `boost::math::policies::overflow_error<>` with the template argument to `overflow_error` one of the `error_policy_type` enumerated values.

```
policy<...>::underflow_error_type
```

Specifies how underflow errors are handled, will be an instance of `boost::math::policies::underflow_error<>` with the template argument to `underflow_error` one of the `error_policy_type` enumerated values.

```
policy<...>::denorm_error_type
```

Specifies how denorm errors are handled, will be an instance of `boost::math::policies::denorm_error<>` with the template argument to `denorm_error` one of the `error_policy_type` enumerated values.

```
policy<...>::rounding_error_type
```

Specifies how rounding errors are handled, will be an instance of `boost::math::policies::rounding_error<>` with the template argument to `rounding_error` one of the `error_policy_type` enumerated values.

```
policy<...>::evaluation_error_type
```

Specifies how evaluation errors are handled, will be an instance of `boost::math::policies::evaluation_error<>` with the template argument to `evaluation_error` one of the `error_policy_type` enumerated values.

```
policy<...>::indeterminate_error_type
```

Specifies how indeterminate result errors are handled, will be an instance of `boost::math::policies::indeterminate_result_error<>` with the template argument to `indeterminate_result_error` one of the `error_policy_type` enumerated values.

```
policy<...>::precision_type
```

Specifies the internal precision to use in binary digits (uses zero to represent whatever the default precision is). Will be an instance of `boost::math::policies::digits2<N>` which in turn inherits from `boost::mpl::int_<N>`.

```
policy<...>::promote_float_type
```

Specifies whether or not to promote `float` arguments to `double` precision internally. Will be an instance of `boost::math::policies::promote_float<B>` which in turn inherits from `boost::mpl::bool_<B>`.

```
policy<...>::promote_double_type
```

Specifies whether or not to promote `double` arguments to `long double` precision internally. Will be an instance of `boost::math::policies::promote_float<B>` which in turn inherits from `boost::mpl::bool_<B>`.

```
policy<...>::discrete_quantile_type
```

Specifies how discrete quantiles are evaluated, will be an instance of `boost::math::policies::discrete_quantile<>` instantiated with one of the `discrete_quantile_policy_type` enumerated type.

```
policy<...>::assert_undefined_type
```

Specifies whether mathematically-undefined properties are asserted as compile-time errors, or treated as runtime errors instead. Will be an instance of `boost::math::policies::assert_undefined<B>` which in turn inherits from `boost::math::mpl::bool_<B>`.

```
template <...argument list...>
typename normalise<policy<>, A1>::type make_policy(...argument list..);
```

`make_policy` is a helper function that converts a list of policies into a normalised `policy` class.

```
template <class Policy,
          class A1 = default_policy,
          class A2 = default_policy,
          class A3 = default_policy,
          class A4 = default_policy,
          class A5 = default_policy,
          class A6 = default_policy,
          class A7 = default_policy,
          class A8 = default_policy,
          class A9 = default_policy,
          class A10 = default_policy,
          class A11 = default_policy,
          class A12 = default_policy,
          class A13 = default_policy>
struct normalise
{
   typedef computed-from-template-arguments type;
};
```

The `normalise` class template converts one instantiation of the `policy` class into a normalised form. This is used internally to reduce code bloat: so that instantiating a special function on `policy<A,B>` or `policy<B,A>` actually both generate the same code internally.

Further more, `normalise` can be used to combine a policy with one or more policies: for example many of the special functions will use this to set policies which they don't make use of to their default values, before forwarding to the actual implementation. In this way code bloat is reduced, since the actual implementation depends only on the policy types that they actually use.

Performance

Performance Overview

By and large the performance of this library should be acceptable for most needs. However, often the library has to make a choice whether to be accurate or fast and by default it chooses accuracy over speed. If you would rather have fast rather than fully accurate routines, then refer to the performance section for information and examples on how to achieve this.

In terms of the algorithms used, this library aims to use the same "best of breed" algorithms as many other libraries: the principle difference is that this library is implemented in C++ - taking advantage of all the abstraction mechanisms that C++ offers - where as most traditional numeric libraries are implemented in C or FORTRAN. Traditionally languages such as C or FORTRAN are perceived as easier to optimise than more complex languages like C++, so in a sense this library provides a good test of current compiler technology, and the "abstraction penalty" - if any - of C++ compared to other languages.

The three most important things you can do to ensure the best performance from this library are:

1. Turn on your compilers optimisations: the difference between "release" and "debug" builds can easily be a factor of 20.
2. Pick your compiler carefully: performance differences of up to 8 fold have been found between some Windows compilers for example.
3. Disable internal use of `long double`, this will reduce accuracy but typically yield a 2x speedup on modern x64 hardware/compilers.

The performance section contains more information on the performance of this library, what you can do to fine tune it, and how this library compares to some other open source alternatives.

Interpreting these Results

In all of the following tables, the best performing result in each row, is assigned a relative value of "1" and shown in bold, so a score of "2" means *"twice as slow as the best performing result"*. Actual timings in nano-seconds per function call are also shown in parenthesis. To make the results easier to read, they are color-coded as follows: the best result and everything within 20% of it is green, anything that's more than twice as slow as the best result is red, and results in between are blue.

Result were obtained on a system with an Intel core i7 4710MQ with 16Gb RAM and running either Windows 8.1 or Xubuntu Linux.

Caution

As usual with performance results these should be taken with a large pinch of salt: relative performance is known to shift quite a bit depending upon the architecture of the particular test system used. Further more, our performance results were obtained using our own test data: these test values are designed to provide good coverage of our code and test all the appropriate corner cases. They do not necessarily represent "typical" usage: whatever that may be!

Getting the Best Performance from this Library: Compiler and Compiler Options

By far the most important thing you can do when using this library is turn on your compiler's optimisation options. As the following table shows the penalty for using the library in debug mode can be quite large. In addition switching to 64-bit code has a small but noticeable improvement in performance, as does switching to a different compiler (Intel C++ 15 in this example).

Table 125. Compiler Option Comparison on Windows x64

Function	cl /Od (x86 build)	cl /arch:sse2 /Ox (x86 build)	cl /Ox (x64 build)	icl /Ox (x64 build)
boost::math::cbrt	18.29 (256ns)	4.29 (60ns)	3.14 (44ns)	1.00 (14ns)
boost::math::cyl_bessel_j (integer orders)	10.03 (742ns)	1.77 (131ns)	1.36 (101ns)	1.00 (74ns)
boost::math::ibeta_inv	4.32 (6583ns)	1.29 (1963ns)	1.28 (1957ns)	1.00 (1523ns)

Trading Accuracy for Performance

There are a number of Policies that can be used to trade accuracy for performance:

- Internal promotion: by default functions with `float` arguments are evaluated at `double` precision internally to ensure full precision in the result. Similarly `double` precision functions are evaluated at `long double` precision internally by default. Changing these defaults can have a significant speed advantage at the expense of accuracy, note also that evaluating using `float` internally may result in numerical instability for some of the more complex algorithms, we suggest you use this option with care.
- Target accuracy: just because you choose to evaluate at `double` precision doesn't mean you necessarily want to target full 16-digit accuracy, if you wish you can change the default (full machine precision) to whatever is "good enough" for your particular use case.

For example, suppose you want to evaluate `double` precision functions at `double` precision internally, you can change the global default by passing `-DBOOST_MATH_PROMOTE_DOUBLE_POLICY=false` on the command line, or at the point of call via something like this:

```
double val = boost::math::erf(my_argu↵
ment, boost::math::policies::make_policy(boost::math::policies::promote_double<false>()));
```

However, an easier option might be:

```
#include <boost/math/special_functions.hpp> // Or any individual special function header

namespace math{

namespace precise{
//
// Define a Policy for accurate evaluation - this is the same as the default, unless
// someone has changed the global defaults.
//
typedef boost::math::policies::policy<> accurate_policy;
//
// Invoke BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS to declare
// functions that use the above policy.  Note no trailing
// ";" required on the macro call:
//
BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(accurate_policy)


}

namespace fast{
//
// Define a Policy for fast evaluation:
//
using namespace boost::math::polcies;
typedef policy<promote_double<false> > fast_policy;
//
// Invoke BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS:
//
BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(fast_policy)

}

}
```

And now one can call:

```
math::accurate::tgamma(x);
```

For the "accurate" version of tgamma, and:

```
math::fast::tgamma(x);
```

For the faster version.

Had we wished to change the target precision (to 9 decimal places) as well as the evaluation type used, we might have done:

```
namespace math{
namespace fast{
//
// Define a Policy for fast evaluation:
//
using namespace boost::math::polcies;
typedef policy<promote_double<false>, digits10<9> > fast_policy;
//
// Invoke BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS:
//
BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS(fast_policy)

}
}
```

One can do a similar thing with the distribution classes:

```
#include <boost/math/distributions.hpp> // or any individual distribution header

namespace math{ namespace fast{
//
// Define a policy for fastest possible evaluation:
//
using namespace boost::math::polcies;
typedef policy<promote_float<false> > fast_float_policy;
//
// Invoke BOOST_MATH_DECLARE_DISTRIBUTIONS
//
BOOST_MATH_DECLARE_DISTRIBUTIONS(float, fast_float_policy)

}} // namespaces

//
// And use:
//
float p_val = cdf(math::fast::normal(1.0f, 3.0f), 0.25f);
```

Here's how these options change the relative performance of the distributions on Linux:

[table_Distribution_performance_comparison_for_different_performance_options_with_GNU_C_version_5_1_0_on_linux]

Cost of High-Precision Non-built-in Floating-point

Using user-defined floating-point like Boost.Multiprecision has a very high run-time cost.

To give some flavour of this:

Table 126. Linpack Benchmark

floating-point type	speed Mflops
double	2727
__float128	35
multiprecision::float128	35
multiprecision::cpp_bin_float_quad	6

Performance Tuning Macros

There are a small number of performance tuning options that are determined by configuration macros. These should be set in boost/math/tools/user.hpp; or else reported to the Boost-development mailing list so that the appropriate option for a given compiler and OS platform can be set automatically in our configuration setup.

Macro	**Meaning**
BOOST_MATH_POLY_METHOD	Determines how polynomials and most rational functions are evaluated. Define to one of the values 0, 1, 2 or 3: see below for the meaning of these values.
BOOST_MATH_RATIONAL_METHOD	Determines how symmetrical rational functions are evaluated: mostly this only effects how the Lanczos approximation is evaluated, and how the `evaluate_rational` function behaves. Define to one of the values 0, 1, 2 or 3: see below for the meaning of these values.
BOOST_MATH_MAX_POLY_ORDER	The maximum order of polynomial or rational function that will be evaluated by a method other than 0 (a simple "for" loop).
BOOST_MATH_INT_TABLE_TYPE(RT, IT)	Many of the coefficients to the polynomials and rational functions used by this library are integers. Normally these are stored as tables as integers, but if mixed integer / floating point arithmetic is much slower than regular floating point arithmetic then they can be stored as tables of floating point values instead. If mixed arithmetic is slow then add: 1. define BOOST_MATH_INT_TABLE_TYPE(RT, IT) RT to boost/math/tools/user.hpp, otherwise the default of: 1. define BOOST_MATH_INT_TABLE_TYPE(RT, IT) IT Set in boost/math/config.hpp is fine, and may well result in smaller code.

The values to which `BOOST_MATH_POLY_METHOD` and `BOOST_MATH_RATIONAL_METHOD` may be set are as follows:

Value	Effect
0	The polynomial or rational function is evaluated using Horner's method, and a simple for-loop. Note that if the order of the polynomial or rational function is a runtime parameter, or the order is greater than the value of `BOOST_MATH_MAX_POLY_ORDER`, then this method is always used, irrespective of the value of `BOOST_MATH_POLY_METHOD` or `BOOST_MATH_RATIONAL_METHOD`.
1	The polynomial or rational function is evaluated without the use of a loop, and using Horner's method. This only occurs if the order of the polynomial is known at compile time and is less than or equal to `BOOST_MATH_MAX_POLY_ORDER`.
2	The polynomial or rational function is evaluated without the use of a loop, and using a second order Horner's method. In theory this permits two operations to occur in parallel for polynomials, and four in parallel for rational functions. This only occurs if the order of the polynomial is known at compile time and is less than or equal to `BOOST_MATH_MAX_POLY_ORDER`.
3	The polynomial or rational function is evaluated without the use of a loop, and using a second order Horner's method. In theory this permits two operations to occur in parallel for polynomials, and four in parallel for rational functions. This differs from method "2" in that the code is carefully ordered to make the parallelisation more obvious to the compiler: rather than relying on the compiler's optimiser to spot the parallelisation opportunities. This only occurs if the order of the polynomial is known at compile time and is less than or equal to `BOOST_MATH_MAX_POLY_ORDER`.

The performance test suite generates a report for your particular compiler showing which method is likely to work best, the following tables show the results for MSVC-14.0 and GCC-5.1.0 (Linux). There's not much to choose between the various methods, but generally loop-unrolled methods perform better. Interestingly, ordering the code to try and "second guess" possible optimizations seems not to be such a good idea (method 3 below).

Table 127. Polynomial Method Comparison with Microsoft Visual C++ version 14.0 on Windows x64

Function	Method 0 (Double Coefficients)	Method 0 (Integer Coefficients)	Method 1 (Double Coefficients)	Method 1 (Integer Coefficients)	Method 2 (Double Coefficients)	Method 2 (Integer Coefficients)	Method 3 (Double Coefficients)	Method 3 (Integer Coefficients)
Order 2	-	-	1.00 (9ns)	1.00 (9ns)	1.00 (9ns)	1.00 (9ns)	1.00 (9ns)	1.00 (9ns)
Order 3	2.08 (25ns)	2.75 (33ns)	1.08 (13ns)	1.08 (13ns)	1.08 (13ns)	1.08 (13ns)	1.08 (13ns)	1.00 (12ns)
Order 4	2.06 (35ns)	2.71 (46ns)	1.06 (18ns)	1.00 (17ns)	1.06 (18ns)	1.06 (18ns)	1.00 (17ns)	1.00 (17ns)
Order 5	1.32 (29ns)	2.00 (44ns)	1.00 (22ns)	1.00 (22ns)	1.05 (23ns)	1.05 (23ns)	1.05 (23ns)	1.05 (23ns)
Order 6	1.38 (36ns)	2.04 (53ns)	1.08 (28ns)	1.00 (26ns)	1.08 (28ns)	1.08 (28ns)	1.35 (35ns)	1.38 (36ns)
Order 7	1.43 (43ns)	2.13 (64ns)	1.03 (31ns)	1.00 (30ns)	1.10 (33ns)	1.03 (31ns)	1.10 (33ns)	1.13 (34ns)
Order 8	1.65 (61ns)	2.22 (82ns)	1.00 (37ns)	1.08 (40ns)	1.14 (42ns)	1.05 (39ns)	1.08 (40ns)	1.11 (41ns)
Order 9	1.39 (57ns)	2.05 (84ns)	1.17 (48ns)	1.17 (48ns)	1.00 (41ns)	1.05 (43ns)	1.15 (47ns)	1.12 (46ns)
Order 10	1.37 (63ns)	2.20 (101ns)	1.22 (56ns)	1.24 (57ns)	1.00 (46ns)	1.00 (46ns)	1.17 (54ns)	1.17 (54ns)
Order 11	1.59 (78ns)	2.24 (110ns)	1.37 (67ns)	1.29 (63ns)	1.22 (60ns)	1.00 (49ns)	1.22 (60ns)	1.22 (60ns)
Order 12	1.46 (83ns)	2.16 (123ns)	1.28 (73ns)	1.26 (72ns)	1.02 (58ns)	1.00 (57ns)	1.07 (61ns)	1.05 (60ns)
Order 13	1.61 (90ns)	2.55 (143ns)	1.32 (74ns)	1.39 (78ns)	1.04 (58ns)	1.00 (56ns)	1.11 (62ns)	1.07 (60ns)
Order 14	1.61 (106ns)	2.23 (147ns)	1.45 (96ns)	1.45 (96ns)	1.02 (67ns)	1.02 (67ns)	1.00 (66ns)	1.09 (72ns)
Order 15	1.49 (119ns)	2.10 (168ns)	1.35 (108ns)	1.35 (108ns)	1.00 (80ns)	1.00 (80ns)	1.00 (80ns)	1.02 (82ns)
Order 16	1.54 (129ns)	1.99 (167ns)	1.49 (125ns)	1.45 (122ns)	1.07 (90ns)	1.00 (84ns)	1.08 (91ns)	1.02 (86ns)
Order 17	1.51 (133ns)	2.02 (178ns)	1.57 (138ns)	1.50 (132ns)	1.02 (90ns)	1.00 (88ns)	1.07 (94ns)	1.06 (93ns)
Order 18	1.53 (148ns)	2.16 (210ns)	1.49 (145ns)	1.57 (152ns)	1.11 (108ns)	1.09 (106ns)	1.00 (97ns)	1.08 (105ns)

Function	Method 0 (Double Coefficients)	Method 0 (Integer Coefficients)	Method 1 (Double Coefficients)	Method 1 (Integer Coefficients)	Method 2 (Double Coefficients)	Method 2 (Integer Coefficients)	Method 3 (Double Coefficients)	Method 3 (Integer Coefficients)
Order 19	1.90 (194ns)	2.27 (232ns)	1.62 (165ns)	1.62 (165ns)	1.08 (110ns)	1.00 (102ns)	1.17 (119ns)	1.19 (121ns)
Order 20	1.65 (206ns)	2.08 (260ns)	1.45 (181ns)	1.44 (180ns)	1.00 (125ns)	1.00 (125ns)	1.01 (126ns)	1.03 (129ns)

Table 128. Rational Method Comparison with Microsoft Visual C++ version 14.0 on Windows x64

Function	Method 0 (Double Coefficients)	Method 0 (Integer Coefficients)	Method 1 (Double Coefficients)	Method 1 (Integer Coefficients)	Method 2 (Double Coefficients)	Method 2 (Integer Coefficients)	Method 3 (Double Coefficients)	Method 3 (Integer Coefficients)
Order 2	-	-	2.12 (89ns)	1.95 (82ns)	1.00 (42ns)	1.00 (42ns)	1.00 (42ns)	1.00 (42ns)
Order 3	2.10 (88ns)	2.10 (88ns)	2.05 (86ns)	2.10 (88ns)	1.05 (44ns)	1.00 (42ns)	1.00 (42ns)	1.00 (42ns)
Order 4	2.12 (89ns)	2.21 (93ns)	1.98 (83ns)	2.10 (88ns)	1.02 (43ns)	1.02 (43ns)	1.02 (43ns)	1.00 (42ns)
Order 5	1.07 (90ns)	1.15 (97ns)	1.08 (91ns)	1.00 (84ns)	1.45 (122ns)	1.46 (123ns)	1.45 (122ns)	1.45 (122ns)
Order 6	1.16 (102ns)	1.58 (139ns)	1.00 (88ns)	1.03 (91ns)	1.44 (127ns)	1.44 (127ns)	1.41 (124ns)	1.38 (121ns)
Order 7	1.29 (121ns)	1.44 (135ns)	1.01 (95ns)	1.00 (94ns)	1.38 (130ns)	1.36 (128ns)	1.33 (125ns)	1.36 (128ns)
Order 8	1.33 (134ns)	1.52 (154ns)	1.00 (101ns)	1.08 (109ns)	1.38 (139ns)	1.31 (132ns)	1.39 (140ns)	1.37 (138ns)
Order 9	1.18 (141ns)	1.45 (172ns)	1.00 (119ns)	1.08 (128ns)	1.13 (135ns)	1.26 (150ns)	1.26 (150ns)	1.27 (151ns)
Order 10	1.29 (180ns)	1.28 (178ns)	1.05 (146ns)	1.00 (139ns)	1.06 (147ns)	1.06 (147ns)	1.18 (164ns)	1.17 (163ns)
Order 11	1.28 (187ns)	1.28 (187ns)	1.06 (155ns)	1.05 (154ns)	1.03 (151ns)	1.00 (146ns)	1.19 (174ns)	1.47 (215ns)
Order 12	1.22 (197ns)	1.38 (223ns)	1.04 (168ns)	1.04 (169ns)	1.00 (162ns)	1.04 (169ns)	1.22 (198ns)	1.52 (246ns)
Order 13	1.23 (209ns)	1.29 (220ns)	1.15 (196ns)	1.10 (187ns)	1.00 (170ns)	1.15 (196ns)	1.22 (208ns)	1.61 (273ns)
Order 14	1.28 (242ns)	1.39 (262ns)	1.15 (218ns)	1.14 (216ns)	1.00 (189ns)	1.01 (191ns)	1.49 (282ns)	1.53 (290ns)
Order 15	1.28 (260ns)	1.34 (273ns)	1.12 (227ns)	1.15 (233ns)	1.00 (203ns)	1.00 (203ns)	1.38 (280ns)	1.47 (298ns)
Order 16	1.35 (288ns)	1.40 (300ns)	1.22 (261ns)	1.18 (252ns)	1.00 (214ns)	1.23 (264ns)	1.43 (305ns)	1.52 (325ns)
Order 17	1.16 (259ns)	1.47 (328ns)	1.15 (256ns)	1.35 (302ns)	1.00 (223ns)	1.22 (273ns)	1.50 (334ns)	1.52 (339ns)
Order 18	1.10 (273ns)	1.46 (363ns)	1.10 (273ns)	1.75 (434ns)	1.00 (248ns)	1.30 (322ns)	1.41 (349ns)	1.46 (363ns)

Function	Method 0 (Double Coefficients)	Method 0 (Integer Coefficients)	Method 1 (Double Coefficients)	Method 1 (Integer Coefficients)	Method 2 (Double Coefficients)	Method 2 (Integer Coefficients)	Method 3 (Double Coefficients)	Method 3 (Integer Coefficients)
Order 19	1.26 (330ns)	1.35 (352ns)	1.24 (324ns)	1.33 (348ns)	1.00 (261ns)	1.22 (319ns)	1.44 (377ns)	1.46 (381ns)
Order 20	1.24 (330ns)	1.60 (427ns)	1.22 (327ns)	1.56 (416ns)	1.00 (267ns)	1.19 (317ns)	1.57 (418ns)	1.56 (416ns)

[table_Polynomial_Method_Comparison_with_GNU_C_version_5_1_0_on_linux]

[table_Rational_Method_Comparison_with_GNU_C_version_5_1_0_on_linux]

Comparing Different Compilers

By running our performance test suite multiple times, we can compare the effect of different compilers: as might be expected, the differences are generally small compared to say disabling internal use of `long double`. However, there are still gains to be main, particularly from some of the commercial offerings:

Table 129. Compiler Comparison on Windows x64

Function	Microsoft Visual C++ version 14.0 boost 1.61	GNU C++ version 5.3.0 boost 1.61	GNU C++ version 5.3.0 boost 1.61 promote_double<false>	Intel C++ C++0x mode version 1600 boost 1.61
assoc_laguerre	1.16 (208ns)	1.21 (217ns)	1.06 (190ns)	1.00 (180ns)
assoc_legendre	1.57 (151ns)	1.80 (173ns)	1.16 (111ns)	1.00 (96ns)
beta	1.74 (176ns)	3.78 (382ns)	2.17 (219ns)	1.00 (101ns)
beta (incomplete)	1.44 (520ns)	3.03 (1096ns)	1.84 (666ns)	1.00 (362ns)
cbrt	3.21 (45ns)	3.43 (48ns)	3.07 (43ns)	1.00 (14ns)
cyl_bessel_i	1.11 (404ns)	2.78 (1016ns)	1.21 (442ns)	1.00 (365ns)
cyl_bessel_i (integer order)	1.29 (261ns)	3.16 (638ns)	1.22 (247ns)	1.00 (202ns)
cyl_bessel_j	1.19 (488ns)	2.06 (841ns)	1.25 (513ns)	1.00 (409ns)
cyl_bessel_j (integer order)	1.36 (103ns)	2.39 (182ns)	1.47 (112ns)	1.00 (76ns)
cyl_bessel_k	1.04 (747ns)	9.43 (6743ns)	1.03 (734ns)	1.00 (715ns)
cyl_bessel_k (integer order)	1.26 (419ns)	10.49 (3494ns)	1.10 (367ns)	1.00 (333ns)
cyl_neumann	16.76 (11212ns)	2.01 (1346ns)	1.00 (669ns)	15.35 (10266ns)
cyl_neumann (integer order)	1.00 (225ns)	1.79 (403ns)	1.24 (279ns)	1.02 (229ns)
digamma	1.20 (24ns)	3.75 (75ns)	2.15 (43ns)	1.00 (20ns)
ellint_1	1.87 (329ns)	1.99 (350ns)	1.32 (232ns)	1.00 (176ns)
ellint_1 (complete)	2.18 (61ns)	2.57 (72ns)	1.57 (44ns)	1.00 (28ns)

Function	**Microsoft Visual C++ version 14.0 boost 1.61**	**GNU C++ version 5.3.0 boost 1.61**	**GNU C++ version 5.3.0 boost 1.61 promote_double<false>**	**Intel C++ C++0x mode version 1600 boost 1.61**
ellint_2	1.78 (525ns)	2.17 (640ns)	1.52 (449ns)	1.00 (295ns)
ellint_2 (complete)	2.14 (62ns)	2.07 (60ns)	1.24 (36ns)	1.00 (29ns)
ellint_3	1.94 (1155ns)	2.91 (1733ns)	1.65 (986ns)	1.00 (596ns)
ellint_3 (complete)	2.05 (721ns)	3.07 (1079ns)	1.46 (514ns)	1.00 (352ns)
ellint_rc	1.58 (49ns)	2.68 (83ns)	1.55 (48ns)	1.00 (31ns)
ellint_rd	1.75 (332ns)	1.83 (348ns)	1.40 (266ns)	1.00 (190ns)
ellint_rf	1.69 (71ns)	2.40 (101ns)	1.57 (66ns)	1.00 (42ns)
ellint_rj	1.91 (344ns)	2.96 (532ns)	1.43 (257ns)	1.00 (180ns)
erf	1.77 (23ns)	2.92 (38ns)	1.92 (25ns)	1.00 (13ns)
erfc	1.29 (22ns)	3.41 (58ns)	2.06 (35ns)	1.00 (17ns)
expint	1.39 (39ns)	3.07 (86ns)	2.14 (60ns)	1.00 (28ns)
expint (En)	1.10 (117ns)	2.19 (232ns)	1.38 (146ns)	1.00 (106ns)
expm1	1.00 (10ns)	3.10 (31ns)	2.40 (24ns)	1.10 (11ns)
gamma_p	1.29 (192ns)	2.64 (393ns)	1.71 (255ns)	1.00 (149ns)
gamma_p_inv	1.36 (706ns)	2.47 (1288ns)	1.91 (997ns)	1.00 (521ns)
gamma_q	1.16 (179ns)	2.79 (429ns)	1.61 (248ns)	1.00 (154ns)
gamma_q_inv	1.37 (703ns)	2.38 (1225ns)	1.89 (971ns)	1.00 (515ns)

Function	Microsoft Visual C++ version 14.0 boost 1.61	GNU C++ version 5.3.0 boost 1.61	GNU C++ version 5.3.0 boost 1.61 promote_double<false>	Intel C++ C++0x mode version 1600 boost 1.61
ibeta	1.34 (520ns)	3.13 (1218ns)	1.83 (711ns)	1.00 (389ns)
ibeta_inv	1.51 (2193ns)	2.81 (4085ns)	2.27 (3303ns)	1.00 (1452ns)
ibetac	1.29 (518ns)	2.94 (1179ns)	1.73 (694ns)	1.00 (401ns)
ibetac_inv	1.41 (2045ns)	2.80 (4058ns)	1.78 (2572ns)	1.00 (1447ns)
jacobi_cn	1.36 (182ns)	3.72 (499ns)	2.28 (306ns)	1.00 (134ns)
jacobi_dn	1.82 (262ns)	3.68 (530ns)	2.27 (327ns)	1.00 (144ns)
jacobi_sn	1.31 (179ns)	3.73 (511ns)	2.43 (333ns)	1.00 (137ns)
laguerre	1.00 (121ns)	1.15 (139ns)	1.04 (126ns)	1.10 (133ns)
legendre	1.07 (364ns)	1.16 (396ns)	1.00 (340ns)	1.13 (383ns)
legendre Q	1.00 (427ns)	1.20 (512ns)	1.01 (430ns)	1.07 (455ns)
lgamma	1.00 (73ns)	3.07 (224ns)	1.82 (133ns)	1.05 (77ns)
log1p	1.00 (14ns)	2.00 (28ns)	1.14 (16ns)	1.00 (14ns)
polygamma	1.16 (3773ns)	1.02 (3320ns)	2.24 (7270ns)	1.00 (3246ns)
sph_bessel	1.17 (1005ns)	1.55 (1325ns)	1.09 (931ns)	1.00 (857ns)
sph_neumann	1.08 (1827ns)	2.07 (3483ns)	1.00 (1685ns)	1.01 (1702ns)
tgamma	1.31 (77ns)	4.03 (238ns)	2.49 (147ns)	1.00 (59ns)
tgamma (incomplete)	1.26 (276ns)	2.52 (552ns)	1.73 (379ns)	1.00 (219ns)

Function	**Microsoft Visual C++ version 14.0 boost 1.61**	**GNU C++ version 5.3.0 boost 1.61**	**GNU C++ version 5.3.0 boost 1.61 promote_double<false>**	**Intel C++ C++0x mode version 1600 boost 1.61**
trigamma	1.35 (23ns)	2.88 (49ns)	1.41 (24ns)	1.00 (17ns)
zeta	1.34 (121ns)	3.83 (345ns)	2.52 (227ns)	1.00 (90ns)

Table 130. Compiler Comparison on linux

Function	GNU C++ version 5.3.0 boost 1.61	GNU C++ version 5.3.0 boost 1.61 promote_double<false>
assoc_laguerre	1.36 (263ns)	1.00 (194ns)
assoc_legendre	2.55 (258ns)	1.00 (101ns)
beta	4.71 (734ns)	1.00 (156ns)
beta (incomplete)	3.12 (1796ns)	1.00 (575ns)
cbrt	2.53 (43ns)	1.00 (17ns)
cyl_bessel_i	3.47 (1410ns)	1.00 (406ns)
cyl_bessel_i (integer order)	3.38 (893ns)	1.00 (264ns)
cyl_bessel_j	2.69 (1071ns)	1.00 (398ns)
cyl_bessel_j (integer order)	2.59 (275ns)	1.00 (106ns)
cyl_bessel_k	6.86 (4589ns)	1.00 (669ns)
cyl_bessel_k (integer order)	10.17 (3673ns)	1.00 (361ns)
cyl_neumann	2.48 (1478ns)	1.00 (597ns)
cyl_neumann (integer order)	2.20 (484ns)	1.00 (220ns)
digamma	1.92 (75ns)	1.00 (39ns)
ellint_1	1.94 (358ns)	1.00 (185ns)
ellint_1 (complete)	1.86 (52ns)	1.00 (28ns)
ellint_2	2.32 (805ns)	1.00 (347ns)

Function	**GNU C++ version 5.3.0** **boost 1.61**	**GNU C++ version 5.3.0** **boost 1.61** **promote_double<false>**
ellint_2 (complete)	1.71 (60ns)	1.00 (35ns)
ellint_3	2.75 (2154ns)	1.00 (783ns)
ellint_3 (complete)	2.23 (1172ns)	1.00 (525ns)
ellint_rc	1.57 (77ns)	1.00 (49ns)
ellint_rd	2.23 (520ns)	1.00 (233ns)
ellint_rf	1.51 (95ns)	1.00 (63ns)
ellint_rj	2.33 (481ns)	1.00 (206ns)
erf	2.60 (39ns)	1.00 (15ns)
erfc	2.68 (59ns)	1.00 (22ns)
expint	2.47 (94ns)	1.00 (38ns)
expint (En)	1.86 (273ns)	1.00 (147ns)
expm1	1.00 (10ns)	1.10 (11ns)
gamma_p	2.42 (492ns)	1.00 (203ns)
gamma_p_inv	2.24 (1577ns)	1.00 (704ns)
gamma_q	2.53 (508ns)	1.00 (201ns)
gamma_q_inv	2.45 (1841ns)	1.00 (751ns)
ibeta	2.71 (1715ns)	1.00 (634ns)
ibeta_inv	2.58 (5742ns)	1.00 (2224ns)

Function	**GNU C++ version 5.3.0** **boost 1.61**	**GNU C++ version 5.3.0** **boost 1.61** **promote_double<false>**
ibetac	2.66 (1736ns)	1.00 (653ns)
ibetac_inv	2.44 (5451ns)	1.00 (2237ns)
jacobi_cn	2.77 (476ns)	1.00 (172ns)
jacobi_dn	2.80 (481ns)	1.00 (172ns)
jacobi_sn	2.86 (492ns)	1.00 (172ns)
laguerre	1.09 (139ns)	1.00 (128ns)
legendre	1.16 (399ns)	1.00 (345ns)
legendre Q	1.18 (496ns)	1.00 (422ns)
lgamma	2.20 (257ns)	1.00 (117ns)
log1p	1.00 (12ns)	1.00 (12ns)
polygamma	3.93 (2885ns)	1.00 (734ns)
sph_bessel	1.71 (1563ns)	1.00 (915ns)
sph_neumann	2.15 (3745ns)	1.00 (1744ns)
tgamma	3.69 (354ns)	1.00 (96ns)
tgamma (incomplete)	2.35 (744ns)	1.00 (316ns)
trigamma	1.64 (36ns)	1.00 (22ns)
zeta	2.71 (509ns)	1.00 (188ns)

Comparisons to Other Open Source Libraries

We've run our performance tests both for our own code, and against other open source implementations of the same functions. The results are presented below to give you a rough idea of how they all compare. In order to give a more-or-less level playing field our test data was screened against all the libraries being tested, and any unsupported domains removed, likewise for any test cases that gave large errors or unexpected non-finite values.

Caution

You should exercise extreme caution when interpreting these results, relative performance may vary by platform, the tests use data that gives good code coverage of *our* code, but which may skew the results towards the corner cases. Finally, remember that different libraries make different choices with regard to performance verses numerical stability.

The first results compare standard library functions to Boost equivalents with MSVC-14.0:

Table 131. Library Comparison with Microsoft Visual C++ version 14.0 on Windows x64

Function	boost 1.61	math.h
cbrt (85/85 tests selected)	1.00 (45ns)	1.44 (65ns)
cyl_bessel_j (integer order) (267/268 tests selected)	1.00 (103ns)	2.11 (217ns)
cyl_neumann (integer order) (428/428 tests selected)	1.57 (225ns)	1.00 (143ns)
erf (950/950 tests selected)	1.10 (23ns)	1.00 (21ns)
erfc (950/950 tests selected)	1.00 (22ns)	3.36 (74ns)
expm1 (80/80 tests selected)	1.00 (10ns)	1.10 (11ns)
lgamma (400/400 tests selected)	1.00 (73ns)	1.74 (127ns)
log1p (80/80 tests selected)	1.08 (14ns)	1.00 (13ns)
tgamma (400/400 tests selected)	1.00 (77ns)	12.12 (933ns)

On Linux with GCC, we can also compare to the TR1 functions, and to GSL and RMath:

[table_Library_Comparison_with_GNU_C_version_5_1_0_on_linux]

And finally we can compare the statistical distributions to GSL, RMath and DCDFLIB:

[table_Distribution_performance_comparison_with_GNU_C_version_5_1_0_on_linux]

The Performance Test Applications

Under *boost-path*/libs/math/reporting/performance you will find some reasonable comprehensive performance test applications for this library.

In order to generate the tables you will have seen in this documentation (or others for your specific compiler) you need to invoke `bjam` in this directory, using a C++11 capable compiler. Note that results extend/overwrite whatever is already present in *boost-path*/libs/math/reporting/performance/doc/performance_tables.qbk, you may want to delete this file before you begin so as to make a fresh start for your particular system.

The programs produce results in Boost's Quickbook format which is not terribly human readable. If you configure your user-config.jam to be able to build Docbook documentation, then you will also get a full summary of all the data in HTML format in *boost-path*/libs/math/reporting/performance/html/index.html. Assuming you're on a 'nix-like platform the procedure to do this is to first install the `xsltproc`, `Docbook DTD`, and `Bookbook XSL` packages. Then:

- Copy *boost-path*/tools/build/example/user-config.jam to your home directory.
- Add `using xsltproc ;` to the end of the file (note the space surrounding each token, including the final ";", this is important!) This assumes that `xsltproc` is in your path.
- Add `using boostbook : path-to-xsl-stylesheets : path-to-dtd ;` to the end of the file. The `path-to-dtd` should point to version 4.2.x of the Docbook DTD, while `path-to-xsl-stylesheets` should point to the folder containing the latest XSLT stylesheets. Both paths should use all forward slashes even on Windows.

At this point you should be able to run the tests and generate the HTML summary, if GSL, RMath or libstdc++ are present in the compilers path they will be automatically tested. For DCDFLIB you will need to place the C source in *boost-path*/libs/math/reporting/performance/third_party/dcdflib.

If you want to compare multiple compilers, or multiple options for one compiler, then you will need to invoke `bjam` multiple times, once for each compiler. Note that in order to test multiple configurations of the same compiler, each has to be given a unique name in the test program, otherwise they all edit the same table cells. Suppose you want to test GCC with and without the -ffast-math option, in this case bjam would be invoked first as:

```
bjam toolset=gcc -a cxxflags=-std=gnu++11
```

Which would run the tests using default optimization options (-O3), we can then run again using -ffast-math:

```
bjam toolset=gcc -a cxxflags='-std=gnu++11 -ffast-math' define=COMPILER_NAME='"GCC with -ffast-math"'
```

In the command line above, the -a flag forces a full rebuild, and the preprocessor define COMPILER_NAME needs to be set to a string literal describing the compiler configuration, hence the double quotes - one for the command line, one for the compiler.

Backgrounders

Additional Implementation Notes

The majority of the implementation notes are included with the documentation of each function or distribution. The notes here are of a more general nature, and reflect more the general implementation philosophy used.

Implementation philosophy

"First be right, then be fast."

There will always be potential compromises to be made between speed and accuracy. It may be possible to find faster methods, particularly for certain limited ranges of arguments, but for most applications of math functions and distributions, we judge that speed is rarely as important as accuracy.

So our priority is accuracy.

To permit evaluation of accuracy of the special functions, production of extremely accurate tables of test values has received considerable effort.

(It also required much CPU effort - there was some danger of molten plastic dripping from the bottom of JM's laptop, so instead, PAB's Dual-core desktop was kept 50% busy for **days** calculating some tables of test values!)

For a specific RealType, say `float` or `double`, it may be possible to find approximations for some functions that are simpler and thus faster, but less accurate (perhaps because there are no refining iterations, for example, when calculating inverse functions).

If these prove accurate enough to be "fit for his purpose", then a user may substitute his custom specialization.

For example, there are approximations dating back from times when computation was a **lot** more expensive:

H Goldberg and H Levine, Approximate formulas for percentage points and normalisation of t and chi squared, Ann. Math. Stat., 17(4), 216 - 225 (Dec 1946).

A H Carter, Approximations to percentage points of the z-distribution, Biometrika 34(2), 352 - 358 (Dec 1947).

These could still provide sufficient accuracy for some speed-critical applications.

Accuracy and Representation of Test Values

In order to be accurate enough for as many as possible real types, constant values are given to 50 decimal digits if available (though many sources proved only accurate near to 64-bit double precision). Values are specified as long double types by appending L, unless they are exactly representable, for example integers, or binary fractions like 0.125. This avoids the risk of loss of accuracy converting from double, the default type. Values are used after `static_cast<RealType>(1.2345L)` to provide the appropriate RealType for spot tests.

Functions that return constants values, like kurtosis for example, are written as

```
static_cast<RealType>(-3) / 5;
```

to provide the most accurate value that the compiler can compute for the real type. (The denominator is an integer and so will be promoted exactly).

So tests for one third, **not** exactly representable with radix two floating-point, (should) use, for example:

```
static_cast<RealType>(1) / 3;
```

If a function is very sensitive to changes in input, specifying an inexact value as input (such as 0.1) can throw the result off by a noticeable amount: 0.1f is "wrong" by ~1e-7 for example (because 0.1 has no exact binary representation). That is why exact binary values - halves, quarters, and eighths etc - are used in test code along with the occasional fraction `a/b` with `b` a power of two (in order to ensure that the result is an exactly representable binary value).

Tolerance of Tests

The tolerances need to be set to the maximum of:

- Some epsilon value.
- The accuracy of the data (often only near 64-bit double).

Otherwise when long double has more digits than the test data, then no amount of tweaking an epsilon based tolerance will work.

A common problem is when tolerances that are suitable for implementations like Microsoft VS.NET where double and long double are the same size: tests fail on other systems where long double is more accurate than double. Check first that the suffix L is present, and then that the tolerance is big enough.

Handling Unsuitable Arguments

In Errors in Mathematical Special Functions, J. Marraffino & M. Paterno it is proposed that signalling a domain error is mandatory when the argument would give an mathematically undefined result.

- Guideline 1

  > A mathematical function is said to be defined at a point a = (a1, a2, . . .) if the limits as x = (x1, x2, . . .) 'approaches a from all directions agree'. The defined value may be any number, or +infinity, or -infinity.

Put crudely, if the function goes to + infinity and then emerges 'round-the-back' with - infinity, it is NOT defined.

> The library function which approximates a mathematical function shall signal a domain error whenever evaluated with argument values for which the mathematical function is undefined.

- Guideline 2

  > The library function which approximates a mathematical function shall signal a domain error whenever evaluated with argument values for which the mathematical function obtains a non-real value.

This implementation is believed to follow these proposals and to assist compatibility with *ISO/IEC 9899:1999 Programming languages - C* and with the Draft Technical Report on C++ Library Extensions, 2005-06-24, section 5.2.1, paragraph 5. See also domain_error.

See policy reference for details of the error handling policies that should allow a user to comply with any of these recommendations, as well as other behaviour.

See error handling for a detailed explanation of the mechanism, and error_handling example and error_handling_example.cpp

Caution

If you enable throw but do NOT have try & catch block, then the program will terminate with an uncaught exception and probably abort. Therefore to get the benefit of helpful error messages, enabling **all** exceptions **and** using try&catch is recommended for all applications. However, for simplicity, this is not done for most examples.

Handling of Functions that are Not Mathematically defined

Functions that are not mathematically defined, like the Cauchy mean, fail to compile by default. A policy allows control of this.

If the policy is to permit undefined functions, then calling them throws a domain error, by default. But the error policy can be set to not throw, and to return NaN instead. For example,

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY ignore_error
```

appears before the first Boost include, then if the un-implemented function is called, mean(cauchy<>()) will return std::numeric_limits<T>::quiet_NaN().

Warning

If `std::numeric_limits<T>::has_quiet_NaN` is false (for example, if T is a User-defined type without NaN support), then an exception will always be thrown when a domain error occurs. Catching exceptions is therefore strongly recommended.

Median of distributions

There are many distributions for which we have been unable to find an analytic formula, and this has deterred us from implementing median functions, the mid-point in a list of values.

However a useful numerical approximation for distribution `dist` is available as usual as an accessor non-member function median using `median(dist)`, that may be evaluated (in the absence of an analytic formula) by calling

`quantile(dist, 0.5)` (this is the *mathematical* definition of course).

Mean, Median, and Skew, Paul T von Hippel

Descriptive Statistics,

and

Mathematica Basic Statistics. give more detail, in particular for discrete distributions.

Handling of Floating-Point Infinity

Some functions and distributions are well defined with + or - infinity as argument(s), but after some experiments with handling infinite arguments as special cases, we concluded that it was generally more useful to forbid this, and instead to return the result of domain_error.

Handling infinity as special cases is additionally complicated because, unlike built-in types on most - but not all - platforms, not all User-Defined Types are specialized to provide `std::numeric_limits<RealType>::infinity()` and would return zero rather than any representation of infinity.

The rationale is that non-finiteness may happen because of error or overflow in the users code, and it will be more helpful for this to be diagnosed promptly rather than just continuing. The code also became much more complicated, more error-prone, much more work to test, and much less readable.

However in a few cases, for example normal, where we felt it obvious, we have permitted argument(s) to be infinity, provided infinity is implemented for the `RealType` on that implementation, and it is supported and tested by the distribution.

The range for these distributions is set to infinity if supported by the platform, (by testing `std::numeric_limits<RealType>::has_infinity`) else the maximum value provided for the `RealType` by Boost.Math.

Testing for has_infinity is obviously important for arbitrary precision types where infinity makes much less sense than for IEEE754 floating-point.

So far we have not set `support()` function (only range) on the grounds that the PDF is uninteresting/zero for infinities.

Users who require special handling of infinity (or other specific value) can, of course, always intercept this before calling a distribution or function and return their own choice of value, or other behavior. This will often be simpler than trying to handle the aftermath of the error policy.

Overflow, underflow, denorm can be handled using error handling policies.

We have also tried to catch boundary cases where the mathematical specification would result in divide by zero or overflow and signalling these similarly. What happens at (and near), poles can be controlled through error handling policies.

Scale, Shape and Location

We considered adding location and scale to the list of functions, for example:

```
template <class RealType>
inline RealType scale(const triangular_distribution<RealType>& dist)
{
  RealType lower = dist.lower();
  RealType mode = dist.mode();
  RealType upper = dist.upper();
  RealType result;  // of checks.
  if(false == detail::check_triangular(BOOST_CURRENT_FUNCTION, lower, mode, upper, &result))
  {
    return result;
  }
  return (upper - lower);
}
```

but found that these concepts are not defined (or their definition too contentious) for too many distributions to be generally applicable. Because they are non-member functions, they can be added if required.

Notes on Implementation of Specific Functions & Distributions

- Default parameters for the Triangular Distribution. We are uncertain about the best default parameters. Some sources suggest that the Standard Triangular Distribution has lower = 0, mode = half and upper = 1. However as a approximation for the normal distribution, the most common usage, lower = -1, mode = 0 and upper = 1 would be more suitable.

Rational Approximations Used

Some of the special functions in this library are implemented via rational approximations. These are either taken from the literature, or devised by John Maddock using our Remez code.

Rational rather than Polynomial approximations are used to ensure accuracy: polynomial approximations are often wonderful up to a certain level of accuracy, but then quite often fail to provide much greater accuracy no matter how many more terms are added.

Our own approximations were devised either for added accuracy (to support 128-bit long doubles for example), or because literature methods were unavailable or under non-BSL compatible license. Our Remez code is known to produce good agreement with literature results in fairly simple "toy" cases. All approximations were checked for convergence and to ensure that they were not ill-conditioned (the coefficients can give a theoretically good solution, but the resulting rational function may be un-computable at fixed precision).

Recomputing using different Remez implementations may well produce differing coefficients: the problem is well known to be ill conditioned in general, and our Remez implementation often found a broad and ill-defined minima for many of these approximations (of course for simple "toy" examples like approximating `exp` the minima is well defined, and the coefficients should agree no matter whose Remez implementation is used). This should not in general effect the validity of the approximations: there's good literature supporting the idea that coefficients can be "in error" without necessarily adversely effecting the result. Note that "in error" has a special meaning in this context, see "Approximate construction of rational approximations and the effect of error autocorrection.", Grigori Litvinov, eprint arXiv:math/0101042. Therefore the coefficients still need to be accurately calculated, even if they can be in error compared to the "true" minimax solution.

Representation of Mathematical Constants

A macro BOOST_DEFINE_MATH_CONSTANT in constants.hpp is used to provide high accuracy constants to mathematical functions and distributions, since it is important to provide values uniformly for both built-in float, double and long double types, and for User Defined types in Boost.Multiprecision like cpp_dec_float. and others like NTL::quad_float and NTL::RR.

To permit calculations in this Math ToolKit and its tests, (and elsewhere) at about 100 decimal digits with NTL::RR type, it is obviously necessary to define constants to this accuracy.

However, some compilers do not accept decimal digits strings as long as this. So the constant is split into two parts, with the 1st containing at least long double precision, and the 2nd zero if not needed or known. The 3rd part permits an exponent to be provided if necessary (use zero if none) - the other two parameters may only contain decimal digits (and sign and decimal point), and may NOT include an exponent like 1.234E99 (nor a trailing F or L). The second digit string is only used if T is a User-Defined Type, when the constant is converted to a long string literal and lexical_casted to type T. (This is necessary because you can't use a numeric constant since even a long double might not have enough digits).

For example, pi is defined:

```
BOOST_DEFINE_MATH_CONSTANT(pi,
  3.141592653589793238462643383279502884197169399375105820974944,
  5923078164062862089986280348253421170679821480865132823066470938446095505,
  0)
```

And used thus:

```
using namespace boost::math::constants;

double diameter = 1.;
double radius = diameter * pi<double>();

or boost::math::constants::pi<NTL::RR>()
```

Note that it is necessary (if inconvenient) to specify the type explicitly.

So you cannot write

```
double p = boost::math::constants::pi<>();  // could not deduce template argument for 'T'
```

Neither can you write:

```
double p = boost::math::constants::pi; // Context does not allow for disambiguation of over↲
loaded function
double p = boost::math::constants::pi(); // Context does not allow for disambiguation of over↲
loaded function
```

Thread safety

Reporting of error by setting `errno` should be thread-safe already (otherwise none of the std lib math functions would be thread safe?). If you turn on reporting of errors via exceptions, `errno` gets left unused anyway.

For normal C++ usage, the Boost.Math `static const` constants are now thread-safe so for built-in real-number types: `float`, `double` and `long double` are all thread safe.

For User_defined types, for example, cpp_dec_float, the Boost.Math should also be thread-safe, (thought we are unsure how to rigorously prove this).

(Thread safety has received attention in the C++11 Standard revision, so hopefully all compilers will do the right thing here at some point.)

Sources of Test Data

We found a large number of sources of test data. We have assumed that these are *"known good"* if they agree with the results from our test and only consulted other sources for their *'vote'* in the case of serious disagreement. The accuracy, actual and claimed, vary very widely. Only Wolfram Mathematica functions provided a higher accuracy than C++ double (64-bit floating-point) and was regarded as the most-trusted source by far. The The R Project for Statistical Computing provided the widest range of distributions, but the usual Intel X86 distribution uses 64-but doubles, so our use was limited to the 15 to 17 decimal digit accuracy.

A useful index of sources is: Web-oriented Teaching Resources in Probability and Statistics

Statlet: Is a Javascript application that calculates and plots probability distributions, and provides the most complete range of distributions:

> Bernoulli, Binomial, discrete uniform, geometric, hypergeometric, negative binomial, Poisson, beta, Cauchy-Lorentz, chi-sequared, Erlang, exponential, extreme value, Fisher, gamma, Laplace, logistic, lognormal, normal, Parteo, Student's t, triangular, uniform, and Weibull.

It calculates pdf, cdf, survivor, log survivor, hazard, tail areas, & critical values for 5 tail values.

It is also the only independent source found for the Weibull distribution; unfortunately it appears to suffer from very poor accuracy in areas where the underlying special function is known to be difficult to implement.

Testing for Invalid Parameters to Functions and Constructors

After finding that some 'bad' parameters (like NaN) were not throwing a `domain_error` exception as they should, a function

`check_out_of_range` (in `test_out_of_range.hpp`) was devised by JM to check (using Boost.Test's BOOST_CHECK_THROW macro) that bad parameters passed to constructors and functions throw `domain_error` exceptions.

Usage is `check_out_of_range< DistributionType >(list-of-params);` Where list-of-params is a list of **valid** parameters from which the distribution can be constructed - ie the same number of args are passed to the function, as are passed to the distribution constructor.

The values of the parameters are not important, but must be **valid** to pass the constructor checks; the default values are suitable, but must be explicitly provided, for example:

```
check_out_of_range<extreme_value_distribution<RealType> >(1, 2);
```

Checks made are:

- Infinity or NaN (if available) passed in place of each of the valid params.
- Infinity or NaN (if available) as a random variable.
- Out-of-range random variable passed to pdf and cdf (ie outside of "range(DistributionType)").
- Out-of-range probability passed to quantile function and complement.

but does **not** check finite but out-of-range parameters to the constructor because these are specific to each distribution, for example:

```
BOOST_CHECK_THROW(pdf(pareto_distribution<RealType>(0, 1), 0), std::domain_error);
BOOST_CHECK_THROW(pdf(pareto_distribution<RealType>(1, 0), 0), std::domain_error);
```

checks `scale` and `shape` parameters are both > 0 by checking that `domain_error` exception is thrown if either are == 0.

(Use of `check_out_of_range` function may mean that some previous tests are now redundant).

It was also noted that if more than one parameter is bad, then only the first detected will be reported by the error message.

Creating and Managing the Equations

Equations that fit on a single line can most easily be produced by inline Quickbook code using templates for Unicode Greek and Unicode Math symbols. All Greek letter and small set of Math symbols is available at /boost-path/libs/math/doc/sf_and_dist/html4_symbols.qbk

Where equations need to use more than one line, real Math editors were used.

The primary source for the equations is now MathML: see the *.mml files in libs/math/doc/sf_and_dist/equations/.

These are most easily edited by a GUI editor such as Mathcast, please note that the equation editor supplied with Open Office currently mangles these files and should not currently be used.

Conversion to SVG was achieved using SVGMath and a command line such as:

```
$for file in *.mml; do
>/cygdrive/c/Python25/python.exe 'C:\download\open\SVGMath-0.3.1\math2svg.py' \
>>$file > $(basename $file .mml).svg
>done
```

See also the section on "Using Python to run Inkscape" and "Using inkscape to convert scalable vector SVG files to Portable Network graphic PNG".

Note that SVGMath requires that the mml files are **not** wrapped in an XHTML XML wrapper - this is added by Mathcast by default - one workaround is to copy an existing mml file and then edit it with Mathcast: the existing format should then be preserved. This is a bug in the XML parser used by SVGMath which the author is aware of.

If necessary the XHTML wrapper can be removed with:

```
cat filename | tr -d "\r\n" | sed -e 's/.*\(<math[^>]*>.*</math>\).*/\1/' > newfile
```

Setting up fonts for SVGMath is currently rather tricky, on a Windows XP system JM's font setup is the same as the sample config file provided with SVGMath but with:

```
    <!-- Double-struck -->
    <mathvariant name="double-struck" family="Mathematica7, Lucida Sans Unicode"/>
```

changed to:

```
    <!-- Double-struck -->
    <mathvariant name="double-struck" family="Lucida Sans Unicode"/>
```

Note that unlike the sample config file supplied with SVGMath, this does not make use of the Mathematica 7 font as this lacks sufficient Unicode information for it to be used with either SVGMath or XEP "as is".

Also note that the SVG files in the repository are almost certainly Windows-specific since they reference various Windows Fonts.

PNG files can be created from the SVGs using Batik and a command such as:

```
java -jar 'C:\download\open\batik-1.7\batik-rasterizer.jar' -dpi 120 *.svg
```

Or using Inkscape (File, Export bitmap, Drawing tab, bitmap size (default size, 100 dpi), Filename (default). png)

or Using Cygwin, a command such as:

```
for file in *.svg; do
  /cygdrive/c/progra~1/Inkscape/inkscape -d 120 -e $(cygpath -a -w $(basename $file .svg).png) ↵
$(cygpath -a -w $file);
done
```

Using BASH

```
# Convert single SVG to PNG file.
# /c/progra~1/Inkscape/inkscape -d 120 -e a.png a.svg
```

or to convert All files in folder SVG to PNG.

```
for file in *.svg; do
/c/progra~1/Inkscape/inkscape -d 120 -e $(basename $file .svg).png $file
done
```

Currently Inkscape seems to generate the better looking PNGs.

The PDF is generated into \pdf\math.pdf using a command from a shell or command window with current directory \math_toolkit\libs\math\doc\sf_and_dist, typically:

```
bjam -a pdf >math_pdf.log
```

Note that XEP will have to be configured to **use and embed** whatever fonts are used by the SVG equations (almost certainly editing the sample xep.xml provided by the XEP installation). If you fail to do this you will get XEP warnings in the log file like

```
[warning]could not find any font family matching "Times New Roman"; replaced by Helvetica
```

(html is the default so it is generated at libs\math\doc\html\index.html using command line >bjam -a > math_toolkit.docs.log).

```
<!-- Sample configuration for Windows TrueType fonts.  -->
```

is provided in the xep.xml downloaded, but the Windows TrueType fonts are commented out.

JM's XEP config file \xep\xep.xml has the following font configuration section added:

```
    <font-group xml:base="file:/C:/Windows/Fonts/" label="Windows TrueType" embed="true" sub↲
set="true">
      <font-family name="Arial">
        <font><font-data ttf="arial.ttf"/></font>
        <font style="oblique"><font-data ttf="ariali.ttf"/></font>
        <font weight="bold"><font-data ttf="arialbd.ttf"/></font>
        <font weight="bold" style="oblique"><font-data ttf="arialbi.ttf"/></font>
      </font-family>

      <font-family name="Times New Roman" ligatures="&#xFB01; &#xFB02;">
        <font><font-data ttf="times.ttf"/></font>
        <font style="italic"><font-data ttf="timesi.ttf"/></font>
        <font weight="bold"><font-data ttf="timesbd.ttf"/></font>
        <font weight="bold" style="italic"><font-data ttf="timesbi.ttf"/></font>
      </font-family>

      <font-family name="Courier New">
        <font><font-data ttf="cour.ttf"/></font>
        <font style="oblique"><font-data ttf="couri.ttf"/></font>
        <font weight="bold"><font-data ttf="courbd.ttf"/></font>
        <font weight="bold" style="oblique"><font-data ttf="courbi.ttf"/></font>
      </font-family>

      <font-family name="Tahoma" embed="true">
        <font><font-data ttf="tahoma.ttf"/></font>
        <font weight="bold"><font-data ttf="tahomabd.ttf"/></font>
      </font-family>

      <font-family name="Verdana" embed="true">
        <font><font-data ttf="verdana.ttf"/></font>
        <font style="oblique"><font-data ttf="verdanai.ttf"/></font>
        <font weight="bold"><font-data ttf="verdanab.ttf"/></font>
        <font weight="bold" style="oblique"><font-data ttf="verdanaz.ttf"/></font>
      </font-family>

     <font-family name="Palatino" embed="true" ligatures="&#xFB00; &#xFB01; &#xFB02; &#xFB03; ↲
&#xFB04;">
        <font><font-data ttf="pala.ttf"/></font>
        <font style="italic"><font-data ttf="palai.ttf"/></font>
        <font weight="bold"><font-data ttf="palab.ttf"/></font>
        <font weight="bold" style="italic"><font-data ttf="palabi.ttf"/></font>
      </font-family>

    <font-family name="Lucida Sans Unicode">
         <!-- <font><font-data ttf="lsansuni.ttf"><font> -->
         <!-- actually called l_10646.ttf on Windows 2000 and Vista Sp1 -->
         <font><font-data ttf="l_10646.ttf"></font>
    </font-family>
```

PAB had to alter his because the Lucida Sans Unicode font had a different name. Other changes are very likely to be required if you are not using Windows.

XZ authored his equations using the venerable Latex, JM converted these to MathML using mxlatex. This process is currently unreliable and required some manual intervention: consequently Latex source is not considered a viable route for the automatic production of SVG versions of equations.

Equations are embedded in the quickbook source using the *equation* template defined in math.qbk. This outputs Docbook XML that looks like:

```
<inlinemediaobject>
<imageobject role="html">
<imagedata fileref="../equations/myfile.png"></imagedata>
</imageobject>
<imageobject role="print">
<imagedata fileref="../equations/myfile.svg"></imagedata>
</imageobject>
</inlinemediaobject>
```

MathML is not currently present in the Docbook output, or in the generated HTML: this needs further investigation.

Producing Graphs

Graphs were produced in SVG format and then converted to PNG's using the same process as the equations.

The programs `/libs/math/doc/sf_and_dist/graphs/dist_graphs.cpp` and `/libs/math/doc/sf_and_dist/graphs/sf_graphs.cpp` generate the SVG's directly using the Google Summer of Code 2007 project of Jacob Voytko (whose work so far, considerably enhanced and now reasonably mature and usable, by Paul A. Bristow, is at .\boost-sandbox\SOC\2007\visualization).

Tutorial: How to Write a New Special Function

Implementation

In this section, we'll provide a "recipe" for adding a new special function to this library to make life easier for future authors wishing to contribute. We'll assume the function returns a single floating-point result, and takes two floating-point arguments. For the sake of exposition we'll give the function the name *`my_special`*.

Normally, the implementation of such a function is split into two layers - a public user layer, and an internal implementation layer that does the actual work. The implementation layer is declared inside a `detail` namespace and has a simple signature:

```
namespace boost { namespace math { namespace detail {

template <class T, class Policy>
T my_special_imp(const T& a, const T&b, const Policy& pol)
{
   /* Implementation goes here */
}

}}} // namespaces
```

We'll come back to what can go inside the implementation later, but first lets look at the user layer. This consists of two overloads of the function, with and without a Policy argument:

```
namespace boost{ namespace math{

template <class T, class U>
typename tools::promote_args<T, U>::type my_special(const T& a, const U& b);

template <class T, class U, class Policy>
typename tools::promote_args<T, U>::type my_special(const T& a, const U& b, const Policy& pol);

}} // namespaces
```

Note how each argument has a different template type - this allows for mixed type arguments - the return type is computed from a traits class and is the "common type" of all the arguments after any integer arguments have been promoted to type `double`.

The implementation of the non-policy overload is trivial:

```
namespace boost{ namespace math{

template <class T, class U>
inline typename tools::promote_args<T, U>::type my_special(const T& a, const U& b)
{
   // Simply forward with a default policy:
   return my_special(a, b, policies::policy<>());
}

}} // namespaces
```

The implementation of the other overload is somewhat more complex, as there's some meta-programming to do, but from a runtime perspective is still a one-line forwarding function. Here it is with comments explaining what each line does:

```
namespace boost{ namespace math{

template <class T, class U, class Policy>
inline typename tools::promote_args<T, U>::type my_spe↵
cial(const T& a, const U& b, const Policy& pol)
{
   //
   // We've found some standard library functions to misbehave if any FPU exception flags
   // are set prior to their call, this code will clear those flags, then reset them
   // on exit:
   //
   BOOST_FPU_EXCEPTION_GUARD
   //
   // The type of the result - the common type of T and U after
   // any integer types have been promoted to double:
   //
   typedef typename tools::promote_args<T, U>::type result_type;
   //
   // The type used for the calculation.  This may be a wider type than
   // the result in order to ensure full precision:
   //
   typedef typename policies::evaluation<result_type, Policy>::type value_type;
   //
   // The type of the policy to forward to the actual implementation.
   // We disable promotion of float and double as that's [possibly]
   // happened already in the line above.  Also reset to the default
   // any policies we don't use (reduces code bloat if we're called
   // multiple times with differing policies we don't actually use).
   // Also normalise the type, again to reduce code bloat in case we're
   // called multiple times with functionally identical policies that happen
   // to be different types.
   //
   typedef typename policies::normalise<
      Policy,
      policies::promote_float<false>,
      policies::promote_double<false>,
      policies::discrete_quantile<>,
      policies::assert_undefined<> >::type forwarding_policy;
   //
   // Whew.  Now we can make the actual call to the implementation.
   // Arguments are explicitly cast to the evaluation type, and the result
   // passed through checked_narrowing_cast which handles things like overflow
   // according to the policy passed:
   //
   return policies::checked_narrowing_cast<result_type, forwarding_policy>(
         detail::my_special_imp(
               static_cast<value_type>(a),
               static_cast<value_type>(x),
               forwarding_policy()),
         "boost::math::my_special<%1%>(%1%, %1%)");
}

}} // namespaces
```

We're now almost there, we just need to flesh out the details of the implementation layer:

```
namespace boost { namespace math { namespace detail {

template <class T, class Policy>
T my_special_imp(const T& a, const T&b, const Policy& pol)
{
   /* Implementation goes here */
}

}}} // namespaces
```

The following guidelines indicate what (other than basic arithmetic) can go in the implementation:

- Error conditions (for example bad arguments) should be handled by calling one of the policy based error handlers.
- Calls to standard library functions should be made unqualified (this allows argument dependent lookup to find standard library functions for user-defined floating point types such as those from Boost.Multiprecision). In addition, the macro `BOOST_MATH_STD_USING` should appear at the start of the function (note no semi-colon afterwards!) so that all the math functions in `namespace std` are visible in the current scope.
- Calls to other special functions should be made as fully qualified calls, and include the policy parameter as the last argument, for example `boost::math::tgamma(a, pol)`.
- Where possible, evaluation of series, continued fractions, polynomials, or root finding should use one of the boiler-plate functions. In any case, after any iterative method, you should verify that the number of iterations did not exceed the maximum specified in the Policy type, and if it did terminate as a result of exceeding the maximum, then the appropriate error handler should be called (see existing code for examples).
- Numeric constants such as π etc should be obtained via a call to the appropriate function, for example: `constants::pi<T>()`.
- Where tables of coefficients are used (for example for rational approximations), care should be taken to ensure these are initialized at program startup to ensure thread safety when using user-defined number types. See for example the use of `erf_initializer` in erf.hpp.

Here are some other useful internal functions:

function	Meaning
`policies::digits<T, Policy>()`	Returns number of binary digits in T (possible overridden by the policy).
`policies::get_max_series_iterations<Policy>()`	Maximum number of iterations for series evaluation.
`policies::get_max_root_iterations<Policy>()`	Maximum number of iterations for root finding.
`polices::get_epsilon<T, Policy>()`	Epsilon for type T, possibly overridden by the Policy.
`tools::digits<T>()`	Returns the number of binary digits in T.
`tools::max_value<T>()`	Equivalent to `std::numeric_limits<T>::max()`
`tools::min_value<T>()`	Equivalent to `std::numeric_limits<T>::min()`
`tools::log_max_value<T>()`	Equivalent to the natural logarithm of `std::numeric_limits<T>::max()`
`tools::log_min_value<T>()`	Equivalent to the natural logarithm of `std::numeric_limits<T>::min()`
`tools::epsilon<T>()`	Equivalent to `std::numeric_limits<T>::epsilon()`.
`tools::root_epsilon<T>()`	Equivalent to the square root of `std::numeric_limits<T>::epsilon()`.
`tools::forth_root_epsilon<T>()`	Equivalent to the forth root of `std::numeric_limits<T>::epsilon()`.

Testing

We work under the assumption that untested code doesn't work, so some tests for your new special function are in order, we'll divide these up in to 3 main categories:

Spot Tests

Spot tests consist of checking that the expected exception is generated when the inputs are in error (or otherwise generate undefined values), and checking any special values. We can check for expected exceptions with `BOOST_CHECK_THROW`, so for example if it's a domain error for the last parameter to be outside the range `[0,1]` then we might have:

```
BOOST_CHECK_THROW(my_special(0, -0.1), std::domain_error);
BOOST_CHECK_THROW(my_special(0, 1.1), std::domain_error);
```

When the function has known exact values (typically integer values) we can use `BOOST_CHECK_EQUAL`:

```
BOOST_CHECK_EQUAL(my_special(1.0, 0.0), 0);
BOOST_CHECK_EQUAL(my_special(1.0, 1.0), 1);
```

When the function has known values which are not exact (from a floating point perspective) then we can use `BOOST_CHECK_CLOSE_FRACTION`:

```
// Assumes 4 epsilon is as close as we can get to a true value of 2Pi:
BOOST_CHECK_CLOSE_FRACTION(my_special(0.5, 0.5), 2 * constants::pi<double>(), std::numeric_lim↵
its<double>::epsilon() * 4);
```

Independent Test Values

If the function is implemented by some other known good source (for example Mathematica or it's online versions functions.wolfram.com or www.wolframalpha.com then it's a good idea to sanity check our implementation by having at least one independendly generated value for each code branch our implementation may take. To slot these in nicely with our testing framework it's best to tabulate these like this:

```
// function values calculated on http://functions.wolfram.com/
static const boost::array<boost::array<T, 3>, 10> my_special_data = {{
    {{ SC_(0), SC_(0), SC_(1) }},
    {{ SC_(0), SC_(1), SC_(1.26606587775200833559824462521471753760767031135496220680814) }},
    /* More values here... */
}};
```

We'll see how to use this table and the meaning of the `SC_` macro later. One important point is to make sure that the input values have exact binary representations: so choose values such as 1.5, 1.25, 1.125 etc. This ensures that if `my_special` is unusually sensitive in one area, that we don't get apparently large errors just because the inputs are 0.5 ulp in error.

Random Test Values

We can generate a large number of test values to check both for future regressions, and for accumulated rounding or cancellation error in our implementation. Ideally we would use an independent implementation for this (for example my_special may be defined in directly terms of other special functions but not implemented that way for performance or accuracy reasons). Alternatively we may use our own implementation directly, but with any special cases (asymptotic expansions etc) disabled. We have a set of tools to generate test data directly, here's a typical example:

```
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/math/tools/test_data.hpp>
#include <boost/test/included/prg_exec_monitor.hpp>
#include <fstream>

using namespace boost::math::tools;
using namespace boost::math;
using namespace std;
using namespace boost::multiprecision;

template <class T>
T my_special(T a, T b)
{
   // Implementation of my_special here...
   return a + b;
}

int cpp_main(int argc, char*argv [])
{
   //
   // We'll use so many digits of precision that any
   // calculation errors will still leave us with
   // 40-50 good digits.  We'll only run this program
   // once so it doesn't matter too much how long this takes!
   //
   typedef number<cpp_dec_float<500> > bignum;

   parameter_info<bignum> arg1, arg2;
   test_data<bignum> data;

   bool cont;
   std::string line;

   if(argc < 1)
      return 1;

   do{
      //
      // User interface which prompts for
      // range of input parameters:
      //
      if(0 == get_user_parameter_info(arg1, "a"))
         return 1;
      if(0 == get_user_parameter_info(arg2, "b"))
         return 1;

      //
      // Get a pointer to the function and call
      // test_data::insert to actually generate
      // the values.
      //
      bignum (*fp)(bignum, bignum) = &my_special;
      data.insert(fp, arg2, arg1);

      std::cout << "Any more data [y/n]?";
      std::getline(std::cin, line);
      boost::algorithm::trim(line);
      cont = (line == "y");
   }while(cont);
   //
   // Just need to write the results to a file:
   //
   std::cout << "Enter name of test data file [default=my_special.ipp]";
```

```
    std::getline(std::cin, line);
    boost::algorithm::trim(line);
    if(line == "")
       line = "my_special.ipp";
    std::ofstream ofs(line.c_str());
    line.erase(line.find('.'));
    ofs << std::scientific << std::setprecision(50);
    write_code(ofs, data, line.c_str());

    return 0;
}
```

Typically several sets of data will be generated this way, including random values in some "normal" range, extreme values (very large or very small), and values close to any "interesting" behaviour of the function (singularities etc).

The Test File Header

We split the actual test file into 2 distinct parts: a header that contains the testing code as a series of function templates, and the actual .cpp test driver that decides which types are tested, and sets the "expected" error rates for those types. It's done this way because:

- We want to test with both built in floating point types, and with multiprecision types. However, both compile and runtimes with the latter can be too long for the folks who run the tests to realistically cope with, so it makes sense to split the test into (at least) 2 parts.
- The definition of the SC_ macro used in our tables of data may differ depending on what type we're testing (see below). Again this is largely a matter of managing compile times as large tables of user-defined-types can take a crazy amount of time to compile with some compilers.

The test header contains 2 functions:

```
template <class Real, class T>
void do_test(const T& data, const char* type_name, const char* test_name);

template <class T>
void test(T, const char* type_name);
```

Before implementing those, we'll include the headers we'll need, and provide a default definition for the SC_ macro:

```
// A couple of Boost.Test headers in case we need any BOOST_CHECK_* macros:
#include <boost/test/unit_test.hpp>
#include <boost/test/floating_point_comparison.hpp>
// Our function to test:
#include <boost/math/special_functions/my_special.hpp>
// We need boost::array for our test data, plus a few headers from
// libs/math/test that contain our testing machinary:
#include <boost/array.hpp>
#include "functor.hpp"
#include "handle_test_result.hpp"
#include "table_type.hpp"

#ifndef SC_
#define SC_(x) static_cast<typename table_type<T>::type>(BOOST_JOIN(x, L))
#endif
```

The easiest function to implement is the "test" function which is what we'll be calling from the test-driver program. It simply includes the files containing the tabular test data and calls `do_test` function for each table, along with a description of what's being tested:

```
template <class T>
void test(T, const char* type_name)
{
   //
   // The actual test data is rather verbose, so it's in a separate file
   //
   // The contents are as follows, each row of data contains
   // three items, input value a, input value b and my_special(a, b):
   //
#  include "my_special_1.ipp"

   do_test<T>(my_special_1, name, "MySpecial Function: Mathematica Values");

#  include "my_special_2.ipp"

   do_test<T>(my_special_2, name, "MySpecial Function: Random Values");

#  include "my_special_3.ipp"

   do_test<T>(my_special_3, name, "MySpecial Function: Very Small Values");
}
```

The function `do_test` takes each table of data and calculates values for each row of data, along with statistics for max and mean error etc, most of this is handled by some boilerplate code:

```
template <class Real, class T>
void do_test(const T& data, const char* type_name, const char* test_name)
{
   // Get the type of each row and each element in the rows:
   typedef typename T::value_type row_type;
   typedef Real                   value_type;

   // Get a pointer to our function, we have to use a workaround here
   // as some compilers require the template types to be explicitly
   // specified, while others don't much like it if it is!
   typedef value_type (*pg)(value_type, value_type);
#if defined(BOOST_MATH_NO_DEDUCED_FUNCTION_POINTERS)
   pg funcp = boost::math::my_special<value_type, value_type>;
#else
   pg funcp = boost::math::my_special;
#endif

   // Somewhere to hold our results:
   boost::math::tools::test_result<value_type> result;
   // And some pretty printing:
   std::cout << "Testing " << test_name << " with type " << type_name
      << "\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

   //
   // Test my_special against data:
   //
   result = boost::math::tools::test_hetero<Real>(
      /* First argument is the table */
      data,
      /* Next comes our function pointer, plus the indexes of it's arguments in the table */
      bind_func<Real>(funcp, 0, 1),
      /* Then the index of the result in the table - potentially we can test several
      related functions this way, each having the same input arguments, and different
      output values in different indexes in the table */
      extract_result<Real>(2));
   //
   // Finish off with some boilerplate to check the results were within the expected errors,
   // and pretty print the results:
   //
   handle_test_result(result, data[result.worst()], res↵
ult.worst(), type_name, "boost::math::my_special", test_name);
}
```

Now we just need to write the test driver program, at it's most basic it looks something like this:

```
#include <boost/math/special_functions/math_fwd.hpp>
#include <boost/math/tools/test.hpp>
#include <boost/math/tools/stats.hpp>
#include <boost/type_traits.hpp>
#include <boost/array.hpp>
#include "functor.hpp"

#include "handle_test_result.hpp"
#include "test_my_special.hpp"

BOOST_AUTO_TEST_CASE( test_main )
{
   //
   // Test each floating point type, plus real_concept.
   // We specify the name of each type by hand as typeid(T).name()
   // often gives an unreadable mangled name.
   //
   test(0.1F, "float");
   test(0.1, "double");
   //
   // Testing of long double and real_concept is protected
   // by some logic to disable these for unsupported
   // or problem compilers.
   //
#ifndef BOOST_MATH_NO_LONG_DOUBLE_MATH_FUNCTIONS
   test(0.1L, "long double");
#ifndef BOOST_MATH_NO_REAL_CONCEPT_TESTS
#if !BOOST_WORKAROUND(__BORLANDC__, BOOST_TESTED_AT(0x582))
   test(boost::math::concepts::real_concept(0.1), "real_concept");
#endif
#endif
#else
   std::cout << "<note>The long double tests have been disabled on this platform "
      "either because the long double overloads of the usual math functions are "
      "not available at all, or because they are too inaccurate for these tests "
      "to pass.</note>" << std::cout;
#endif
}
```

That's almost all there is too it - except that if the above program is run it's very likely that all the tests will fail as the default maximum allowable error is 1 epsilon. So we'll define a function (don't forget to call it from the start of the `test_main` above) to up the limits to something sensible, based both on the function we're calling and on the particular tests plus the platform and compiler:

```
void expected_results()
{
   //
   // Define the max and mean errors expected for
   // various compilers and platforms.
   //
   const char* largest_type;
#ifndef BOOST_MATH_NO_LONG_DOUBLE_MATH_FUNCTIONS
   if(boost::math::policies::di↵
gits<double, boost::math::policies::policy<> >() == boost::math::policies::di↵
gits<long double, boost::math::policies::policy<> >())
   {
      largest_type = "(long\\s+)?double|real_concept";
   }
   else
   {
      largest_type = "long double|real_concept";
   }
#else
   largest_type = "(long\\s+)?double";
#endif
   //
   // We call add_expected_result for each error rate we wish to adjust, these tell
   // handle_test_result what level of error is acceptable.  We can have as many calls
   // to add_expected_result as we need, each one establishes a rule for acceptable error
   // with rules set first given preference.
   //
   add_expected_result(
      /* First argument is a regular expression to match against the name of the compiler
         set in BOOST_COMPILER */
      ".*",
      /* Second argument is a regular expression to match against the name of the
         C++ standard library as set in BOOST_STDLIB */
      ".*",
      /* Third argument is a regular expression to match against the name of the
         platform as set in BOOST_PLATFORM */
      ".*",
      /* Forth argument is the name of the type being tested, normally we will
         only need to up the acceptable error rate for the widest floating
         point type being tested */
      largest_real,
      /* Fifth argument is a regular expression to match against
         the name of the group of data being tested */
      "MySpecial Function:.*Small.*",
      /* Sixth argument is a regular expression to match against the name
         of the function being tested */
      "boost::math::my_special",
      /* Seventh argument is the maximum allowable error expressed in units
         of machine epsilon passed as a long integer value */
      50,
      /* Eighth argument is the maximum allowable mean error expressed in units
         of machine epsilon passed as a long integer value */
      20);
}
```

Testing Multiprecision Types

Testing of multiprecision types is handled by the test drivers in libs/multiprecision/test/math, please refer to these for examples. Note that these tests are run only occationally as they take a lot of CPU cycles to build and run.

Improving Compile Times

As noted above, these test programs can take a while to build as we're instantiating a lot of templates for several different types, and our test runners are already stretched to the limit, and probably using outdated "spare" hardware. There are two things we can do to speed things up:

- Use a precompiled header.
- Use separate compilation of our special function templates.

We can make these changes by changing the list of includes from:

```
#include <boost/math/special_functions/math_fwd.hpp>
#include <boost/math/tools/test.hpp>
#include <boost/math/tools/stats.hpp>
#include <boost/type_traits.hpp>
#include <boost/array.hpp>
#include "functor.hpp"

#include "handle_test_result.hpp"
```

To just:

```
#include <pch_light.hpp>
```

And changing

```
#include <boost/math/special_functions/my_special.hpp>
```

To:

```
#include <boost/math/special_functions/math_fwd.hpp>
```

The Jamfile target that builds the test program will need the targets

```
test_instances//test_instances pch_light
```

adding to it's list of source dependencies (see the Jamfile for examples).

Finally the project in libs/math/test/test_instances will need modifying to instantiate function `my_special`.

These changes should be made last, when `my_special` is stable and the code is in Trunk.

Concept Checks

Our concept checks verify that your function's implementation makes no assumptions that aren't required by our Real number conceptual requirements. They also check for various common bugs and programming traps that we've fallen into over time. To add your function to these tests, edit libs/math/test/compile_test/instantiate.hpp to add calls to your function: there are 7 calls to each function, each with a different purpose. Search for something like "ibeta" or "gamm_p" and follow their examples.

Relative Error

Given an actual value *a* and a found value *v* the relative error can be calculated from:

$$\left| \frac{a - v}{a} \right|$$

However the test programs in the library use the symmetrical form:

$$\max\left(\left| \frac{a - v}{a} \right|, \left| \frac{a - v}{v} \right| \right)$$

which measures *relative difference* and happens to be less error prone in use since we don't have to worry which value is the "true" result, and which is the experimental one. It guarantees to return a value at least as large as the relative error.

Special care needs to be taken when one value is zero: we could either take the absolute error in this case (but that's cheating as the absolute error is likely to be very small), or we could assign a value of either 1 or infinity to the relative error in this special case. In the test cases for the special functions in this library, everything below a threshold is regarded as "effectively zero", otherwise the relative error is assigned the value of 1 if only one of the terms is zero. The threshold is currently set at `std::numeric_limits<>::min()`: in other words all denormalised numbers are regarded as a zero.

All the test programs calculate *quantized relative error*, whereas the graphs in this manual are produced with the *actual error*. The difference is as follows: in the test programs, the test data is rounded to the target real type under test when the program is compiled, so the error observed will then be a whole number of *units in the last place* either rounded up from the actual error, or rounded down (possibly to zero). In contrast the *true error* is obtained by extending the precision of the calculated value, and then comparing to the actual value: in this case the calculated error may be some fraction of *units in the last place*.

Note that throughout this manual and the test programs the relative error is usually quoted in units of epsilon. However, remember that *units in the last place* more accurately reflect the number of contaminated digits, and that relative error can *"wobble"* by a factor of 2 compared to *units in the last place*. In other words: two implementations of the same function, whose maximum relative errors differ by a factor of 2, can actually be accurate to the same number of binary digits. You have been warned!

The Impossibility of Zero Error

For many of the functions in this library, it is assumed that the error is "effectively zero" if the computation can be done with a number of guard digits. However it should be remembered that if the result is a *transcendental number* then as a point of principle we can never be sure that the result is accurate to more than 1 ulp. This is an example of what http://en.wikipedia.org/wiki/William_Kahan called http://en.wikipedia.org/wiki/Rounding#The_table-maker.27s_dilemma: consider what happens if the first guard digit is a one, and the remaining guard digits are all zero. Do we have a tie or not? Since the only thing we can tell about a transcendental number is that its digits have no particular pattern, we can never tell if we have a tie, no matter how many guard digits we have. Therefore, we can never be completely sure that the result has been rounded in the right direction. Of course, transcendental numbers that just happen to be a tie - for however many guard digits we have - are extremely rare, and get rarer the more guard digits we have, but even so....

Refer to the classic text What Every Computer Scientist Should Know About Floating-Point Arithmetic for more information.

The Lanczos Approximation

Motivation

Why base gamma and gamma-like functions on the Lanczos approximation?

First of all I should make clear that for the gamma function over real numbers (as opposed to complex ones) the Lanczos approximation (See Wikipedia or Mathworld) appears to offer no clear advantage over more traditional methods such as Stirling's approximation. Pugh carried out an extensive comparison of the various methods available and discovered that they were all very similar in terms of complexity and relative error. However, the Lanczos approximation does have a couple of properties that make it worthy of further consideration:

- The approximation has an easy to compute truncation error that holds for all *z > 0*. In practice that means we can use the same approximation for all *z > 0*, and be certain that no matter how large or small *z* is, the truncation error will *at worst* be bounded by some finite value.
- The approximation has a form that is particularly amenable to analytic manipulation, in particular ratios of gamma or gamma-like functions are particularly easy to compute without resorting to logarithms.

It is the combination of these two properties that make the approximation attractive: Stirling's approximation is highly accurate for large z, and has some of the same analytic properties as the Lanczos approximation, but can't easily be used across the whole range of z.

As the simplest example, consider the ratio of two gamma functions: one could compute the result via lgamma:

```
exp(lgamma(a) - lgamma(b));
```

However, even if lgamma is uniformly accurate to 0.5ulp, the worst case relative error in the above can easily be shown to be:

```
Erel > a * log(a)/2 + b * log(b)/2
```

For small *a* and *b* that's not a problem, but to put the relationship another way: *each time a and b increase in magnitude by a factor of 10, at least one decimal digit of precision will be lost.*

In contrast, by analytically combining like power terms in a ratio of Lanczos approximation's, these errors can be virtually eliminated for small *a* and *b*, and kept under control for very large (or very small for that matter) *a* and *b*. Of course, computing large powers is itself a notoriously hard problem, but even so, analytic combinations of Lanczos approximations can make the difference between obtaining a valid result, or simply garbage. Refer to the implementation notes for the beta function for an example of this method in practice. The incomplete gamma_p gamma and beta functions use similar analytic combinations of power terms, to combine gamma and beta functions divided by large powers into single (simpler) expressions.

The Approximation

The Lanczos Approximation to the Gamma Function is given by:

$$\Gamma(z+1) \quad = \quad \sqrt{2\pi}\,(z+g+0.5)^{z+0.5}e^{-(z+g+0.5)}S_g(z)$$

Where $S_g(z)$ is an infinite sum, that is convergent for all z > 0, and *g* is an arbitrary parameter that controls the "shape" of the terms in the sum which is given by:

$$S_g(z) \quad = \quad \left[\frac{1}{2}a_0 + a_1\frac{z}{z+1} + a_2\frac{z(z-1)}{(z+1)(z+2)} + \ \ldots\right]$$

With individual coefficients defined in closed form by:

$$a_k = (-1)^k \sqrt{\frac{2}{\pi}} e^g k \sum_{j=0}^{k} (-1)^j \frac{(k+j-1)!}{(k-j)!j!} \left(\frac{e}{j+g+\frac{1}{2}} \right)^{j+\frac{1}{2}}$$

However, evaluation of the sum in that form can lead to numerical instability in the computation of the ratios of rising and falling factorials (effectively we're multiplying by a series of numbers very close to 1, so roundoff errors can accumulate quite rapidly).

The Lanczos approximation is therefore often written in partial fraction form with the leading constants absorbed by the coefficients in the sum:

$$\Gamma(z) = \frac{(z+g-0.5)^{z-0.5}}{e^{z+g-0.5}} L_g(z);$$

where:

$$L_g(z) = C_0 + \sum_{k=1}^{N-1} \frac{C_N}{z+k-1}$$

Again parameter *g* is an arbitrarily chosen constant, and *N* is an arbitrarily chosen number of terms to evaluate in the "Lanczos sum" part.

Note

Some authors choose to define the sum from k=1 to N, and hence end up with N+1 coefficients. This happens to confuse both the following discussion and the code (since C++ deals with half open array ranges, rather than the closed range of the sum). This convention is consistent with Godfrey, but not Pugh, so take care when referring to the literature in this field.

Computing the Coefficients

The coefficients C0..CN-1 need to be computed from *N* and *g* at high precision, and then stored as part of the program. Calculation of the coefficients is performed via the method of Godfrey; let the constants be contained in a column vector P, then:

P = D B C F

where B is an NxN matrix:

$$B_{i,j} = \begin{cases} 1 & if \quad i = 0 \\ -1^{j-i} X & if \quad i > 0 \quad j \geq i \\ 0 & otherwise \end{cases} \quad ; \quad X = \binom{i+j-1}{j-i}$$

D is an NxN matrix:

$$D_{ij} = \begin{cases} 0 & if \quad i \neq j \\ 1 & if \quad i = j = 0 \\ -1 & if \quad i = j = 1 \\ \frac{D_{i-1,i-1} 2(2i-1)}{i-1} & otherwise \end{cases}$$

C is an NxN matrix:

$$C_{i,j} = \begin{cases} \frac{1}{2} & if \quad i = j = 0 \\ 0 & if \quad j > i \\ -1^{i-j} S & otherwise \end{cases} \quad ; \quad S = \sum_{k=0}^{i} \binom{2i}{2k} \binom{k}{k+j-i}$$

and F is an N element column vector:

$$F_i = \frac{(2i)!e^{i+g+0.5}}{i!2^{2i-1}(i+g+0.5)^{i+0.5}}$$

Note than the matrices B, D and C contain all integer terms and depend only on *N*, this product should be computed first, and then multiplied by *F* as the last step.

Choosing the Right Parameters

The trick is to choose *N* and *g* to give the desired level of accuracy: choosing a small value for *g* leads to a strictly convergent series, but one which converges only slowly. Choosing a larger value of *g* causes the terms in the series to be large and/or divergent for about the first *g-1* terms, and to then suddenly converge with a "crunch".

Pugh has determined the optimal value of *g* for *N* in the range *1 <= N <= 60*: unfortunately in practice choosing these values leads to cancellation errors in the Lanczos sum as the largest term in the (alternating) series is approximately 1000 times larger than the result. These optimal values appear not to be useful in practice unless the evaluation can be done with a number of guard digits *and* the coefficients are stored at higher precision than that desired in the result. These values are best reserved for say, computing to float precision with double precision arithmetic.

Table 132. Optimal choices for N and g when computing with guard digits (source: Pugh)

Significand Size	N	g	Max Error
24	6	5.581	9.51e-12
53	13	13.144565	9.2213e-23

The alternative described by Godfrey is to perform an exhaustive search of the *N* and *g* parameter space to determine the optimal combination for a given *p* digit floating-point type. Repeating this work found a good approximation for double precision arithmetic (close to the one Godfrey found), but failed to find really good approximations for 80 or 128-bit long doubles. Further it was observed that the approximations obtained tended to optimised for the small values of z (1 < z < 200) used to test the implementation against the factorials. Computing ratios of gamma functions with large arguments were observed to suffer from error resulting from the truncation of the Lancozos series.

Pugh identified all the locations where the theoretical error of the approximation were at a minimum, but unfortunately has published only the largest of these minima. However, he makes the observation that the minima coincide closely with the location where the first neglected term (a_N) in the Lanczos series $S_g(z)$ changes sign. These locations are quite easy to locate, albeit with considerable computer time. These "sweet spots" need only be computed once, tabulated, and then searched when required for an approximation that delivers the required precision for some fixed precision type.

Unfortunately, following this path failed to find a really good approximation for 128-bit long doubles, and those found for 64 and 80-bit reals required an excessive number of terms. There are two competing issues here: high precision requires a large value of *g*, but avoiding cancellation errors in the evaluation requires a small *g*.

At this point note that the Lanczos sum can be converted into rational form (a ratio of two polynomials, obtained from the partial-fraction form using polynomial arithmetic), and doing so changes the coefficients so that *they are all positive*. That means that the sum in rational form can be evaluated without cancellation error, albeit with double the number of coefficients for a given N. Repeating the search of the "sweet spots", this time evaluating the Lanczos sum in rational form, and testing only those "sweet spots" whose theoretical error is less than the machine epsilon for the type being tested, yielded good approximations for all the types tested. The optimal values found were quite close to the best cases reported by Pugh (just slightly larger *N* and slightly smaller *g* for a given precision than Pugh reports), and even though converting to rational form doubles the number of stored coefficients, it should be noted that half of them are integers (and therefore require less storage space) and the approximations require a smaller *N* than would otherwise be required, so fewer floating point operations may be required overall.

The following table shows the optimal values for *N* and *g* when computing at fixed precision. These should be taken as work in progress: there are no values for 106-bit significand machines (Darwin long doubles & NTL quad_float), and further optimisation of the values of *g* may be possible. Errors given in the table are estimates of the error due to truncation of the Lanczos infinite series to *N* terms. They are calculated from the sum of the first five neglected terms - and are known to be rather pessimistic estimates - although it is noticeable that the best combinations of *N* and *g* occurred when the estimated truncation error almost exactly matches the machine epsilon for the type in question.

Table 133. Optimum value for N and g when computing at fixed precision

Significand Size	Platform/Compiler Used	N	g	Max Truncation Error
24	Win32, VC++ 7.1	6	1.428456135094165802001953125	9.41e-007
53	Win32, VC++ 7.1	13	6.024680040776729583740234375	3.23e-016
64	Suse Linux 9 IA64, gcc-3.3.3	17	12.2252227365970611572265625	2.34e-024
116	HP Tru64 Unix 5.1B / Alpha, Compaq C++ V7.1-006	24	20.3209821879863739013671875	4.75e-035

Finally note that the Lanczos approximation can be written as follows by removing a factor of exp(g) from the denominator, and then dividing all the coefficients by exp(g):

$$\Gamma(z) = \left(\frac{z+g-0.5}{e}\right)^{z-0.5} L_{g,e}(z);$$

This form is more convenient for calculating lgamma, but for the gamma function the division by *e* turns a possibly exact quality into an inexact value: this reduces accuracy in the common case that the input is exact, and so isn't used for the gamma function.

References

1. Paul Godfrey, "A note on the computation of the convergent Lanczos complex Gamma approximation".
2. Glendon Ralph Pugh, "An Analysis of the Lanczos Gamma Approximation", PhD Thesis November 2004.
3. Viktor T. Toth, "Calculators and the Gamma Function".
4. Mathworld, The Lanczos Approximation.

The Remez Method

The Remez algorithm is a methodology for locating the minimax rational approximation to a function. This short article gives a brief overview of the method, but it should not be regarded as a thorough theoretical treatment, for that you should consult your favorite textbook.

Imagine that you want to approximate some function f(x) by way of a rational function R(x), where R(x) may be either a polynomial P(x) or a ratio of two polynomials P(x)/Q(x) (a rational function). Initially we'll concentrate on the polynomial case, as it's by far the easier to deal with, later we'll extend to the full rational function case.

We want to find the "best" rational approximation, where "best" is defined to be the approximation that has the least deviation from f(x). We can measure the deviation by way of an error function:

$E_{abs}(x) = f(x) - R(x)$

which is expressed in terms of absolute error, but we can equally use relative error:

$E_{rel}(x) = (f(x) - R(x)) / |f(x)|$

And indeed in general we can scale the error function in any way we want, it makes no difference to the maths, although the two forms above cover almost every practical case that you're likely to encounter.

The minimax rational function R(x) is then defined to be the function that yields the smallest maximal value of the error function. Chebyshev showed that there is a unique minimax solution for R(x) that has the following properties:

- If R(x) is a polynomial of degree N, then there are N+2 unknowns: the N+1 coefficients of the polynomial, and maximal value of the error function.
- The error function has N+1 roots, and N+2 extrema (minima and maxima).
- The extrema alternate in sign, and all have the same magnitude.

That means that if we know the location of the extrema of the error function then we can write N+2 simultaneous equations:

$R(x_i) + (-1)^i E = f(x_i)$

where E is the maximal error term, and x_i are the abscissa values of the N+2 extrema of the error function. It is then trivial to solve the simultaneous equations to obtain the polynomial coefficients and the error term.

Unfortunately we don't know where the extrema of the error function are located!

The Remez Method

The Remez method is an iterative technique which, given a broad range of assumptions, will converge on the extrema of the error function, and therefore the minimax solution.

In the following discussion we'll use a concrete example to illustrate the Remez method: an approximation to the function e^x over the range [-1, 1].

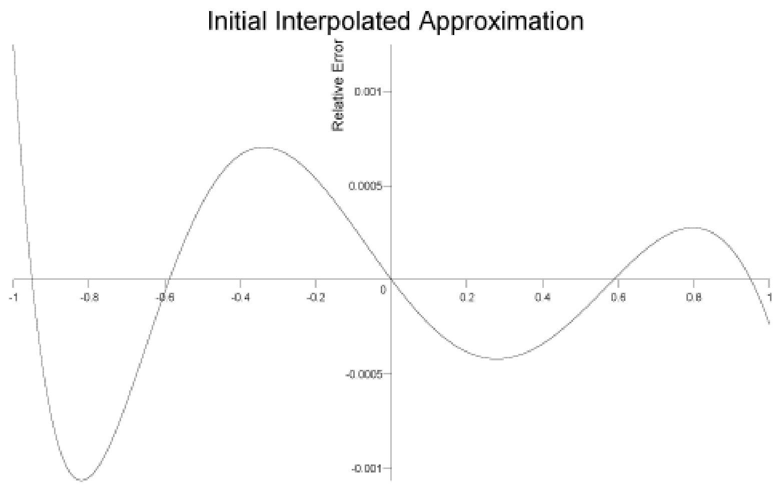
Before we can begin the Remez method, we must obtain an initial value for the location of the extrema of the error function. We could "guess" these, but a much closer first approximation can be obtained by first constructing an interpolated polynomial approximation to f(x).

In order to obtain the N+1 coefficients of the interpolated polynomial we need N+1 points ($x_0 \ldots x_N$): with our interpolated form passing through each of those points that yields N+1 simultaneous equations:

$f(x_i) = P(x_i) = c_0 + c_1 x_i \ldots + c_N x_i^N$

Which can be solved for the coefficients $c_0 \ldots c_N$ in P(x).

Obviously this is not a minimax solution, indeed our only guarantee is that f(x) and P(x) touch at N+1 locations, away from those points the error may be arbitrarily large. However, we would clearly like this initial approximation to be as close to f(x) as possible, and it turns out that using the zeros of an orthogonal polynomial as the initial interpolation points is a good choice. In our example we'll use the zeros of a Chebyshev polynomial as these are particularly easy to calculate, interpolating for a polynomial of degree 4, and measuring *relative error* we get the following error function:



Which has a peak relative error of 1.2×10^{-3}.

While this is a pretty good approximation already, judging by the shape of the error function we can clearly do better. Before starting on the Remez method propper, we have one more step to perform: locate all the extrema of the error function, and store these locations as our initial *Chebyshev control points*.

Note

In the simple case of a polynomial approximation, by interpolating through the roots of a Chebyshev polynomial we have in fact created a *Chebyshev approximation* to the function: in terms of *absolute error* this is the best a priori choice for the interpolated form we can achieve, and typically is very close to the minimax solution.

However, if we want to optimise for *relative error*, or if the approximation is a rational function, then the initial Chebyshev solution can be quite far from the ideal minimax solution.

A more technical discussion of the theory involved can be found in this online course.

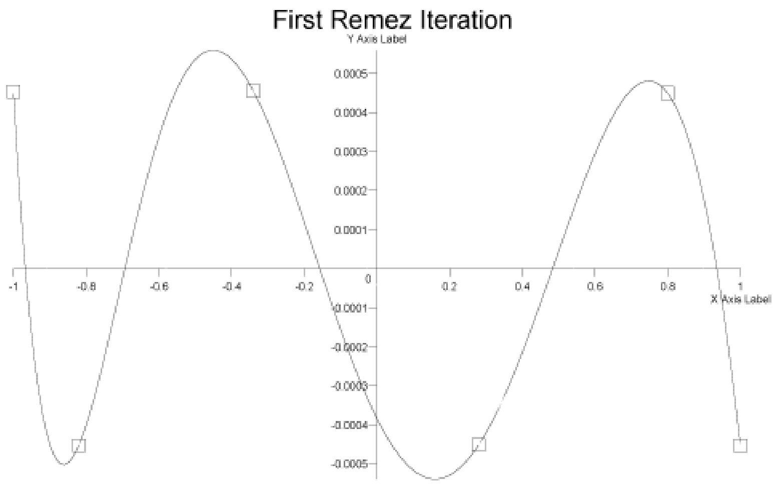
Remez Step 1

The first step in the Remez method, given our current set of N+2 Chebyshev control points x_i, is to solve the N+2 simultaneous equations:

$P(x_i) + (-1)^i E = f(x_i)$

To obtain the error term E, and the coefficients of the polynomial P(x).

This gives us a new approximation to f(x) that has the same error *E* at each of the control points, and whose error function *alternates in sign* at the control points. This is still not necessarily the minimax solution though: since the control points may not be at the extrema of the error function. After this first step here's what our approximation's error function looks like:



Clearly this is still not the minimax solution since the control points are not located at the extrema, but the maximum relative error has now dropped to 5.6×10^{-4}.

Remez Step 2

The second step is to locate the extrema of the new approximation, which we do in two stages: first, since the error function changes sign at each control point, we must have N+1 roots of the error function located between each pair of N+2 control points. Once these roots are found by standard root finding techniques, we know that N extrema are bracketed between each pair of roots, plus two more between the endpoints of the range and the first and last roots. The N+2 extrema can then be found using standard function minimisation techniques.

We now have a choice: multi-point exchange, or single point exchange.

In single point exchange, we move the control point nearest to the largest extrema to the absissa value of the extrema.

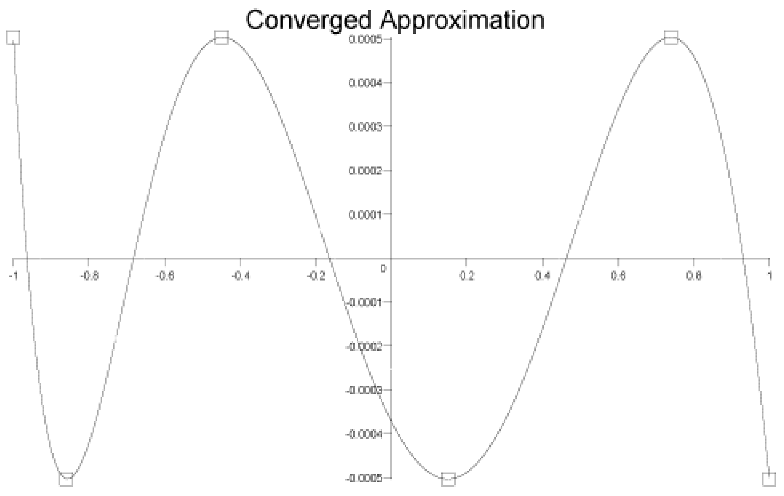
In multi-point exchange we swap all the current control points, for the locations of the extrema.

In our example we perform multi-point exchange.

Iteration

The Remez method then performs steps 1 and 2 above iteratively until the control points are located at the extrema of the error function: this is then the minimax solution.

For our current example, two more iterations converges on a minimax solution with a peak relative error of 5×10^{-4} and an error function that looks like:



Rational Approximations

If we wish to extend the Remez method to a rational approximation of the form

f(x) = R(x) = P(x) / Q(x)

where P(x) and Q(x) are polynomials, then we proceed as before, except that now we have N+M+2 unknowns if P(x) is of order N and Q(x) is of order M. This assumes that Q(x) is normalised so that its leading coefficient is 1, giving N+M+1 polynomial coefficients in total, plus the error term E.

The simultaneous equations to be solved are now:

$P(x_i) / Q(x_i) + (-1)^i E = f(x_i)$

Evaluated at the N+M+2 control points x_i.

Unfortunately these equations are non-linear in the error term E: we can only solve them if we know E, and yet E is one of the unknowns!

The method usually adopted to solve these equations is an iterative one: we guess the value of E, solve the equations to obtain a new value for E (as well as the polynomial coefficients), then use the new value of E as the next guess. The method is repeated until E converges on a stable value.

These complications extend the running time required for the development of rational approximations quite considerably. It is often desirable to obtain a rational rather than polynomial approximation none the less: rational approximations will often match more difficult to approximate functions, to greater accuracy, and with greater efficiency, than their polynomial alternatives. For example, if we takes our previous example of an approximation to e^x, we obtained $5x10^{-4}$ accuracy with an order 4 polynomial. If we move two of the unknowns into the denominator to give a pair of order 2 polynomials, and re-minimise, then the peak relative error drops to $8.7x10^{-5}$. That's a 5 fold increase in accuracy, for the same number of terms overall.

Practical Considerations

Most treatises on approximation theory stop at this point. However, from a practical point of view, most of the work involves finding the right approximating form, and then persuading the Remez method to converge on a solution.

So far we have used a direct approximation:

f(x) = R(x)

But this will converge to a useful approximation only if f(x) is smooth. In addition round-off errors when evaluating the rational form mean that this will never get closer than within a few epsilon of machine precision. Therefore this form of direct approximation is often reserved for situations where we want efficiency, rather than accuracy.

The first step in improving the situation is generally to split f(x) into a dominant part that we can compute accurately by another method, and a slowly changing remainder which can be approximated by a rational approximation. We might be tempted to write:

f(x) = g(x) + R(x)

where g(x) is the dominant part of f(x), but if f(x)/g(x) is approximately constant over the interval of interest then:

f(x) = g(x)(c + R(x))

Will yield a much better solution: here *c* is a constant that is the approximate value of f(x)/g(x) and R(x) is typically tiny compared to *c*. In this situation if R(x) is optimised for absolute error, then as long as its error is small compared to the constant *c*, that error will effectively get wiped out when R(x) is added to *c*.

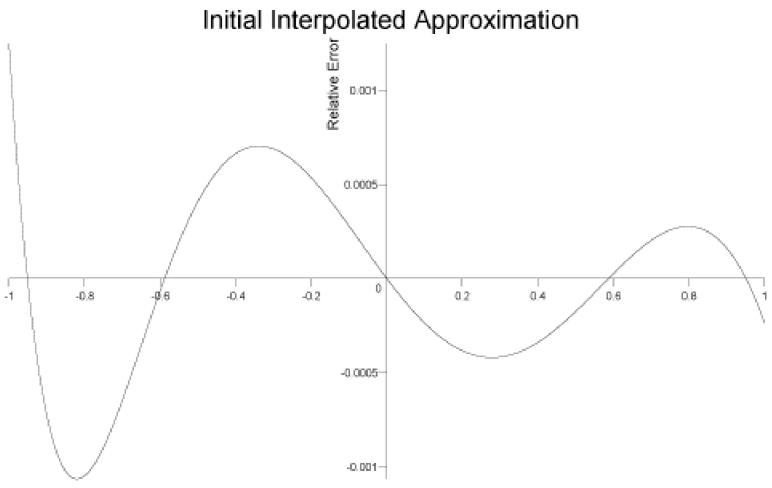
The difficult part is obviously finding the right g(x) to extract from your function: often the asymptotic behaviour of the function will give a clue, so for example the function erfc becomes proportional to e^{-x^2}/x as x becomes large. Therefore using:

erfc(z) = (C + R(x)) e^{-x^2}/x

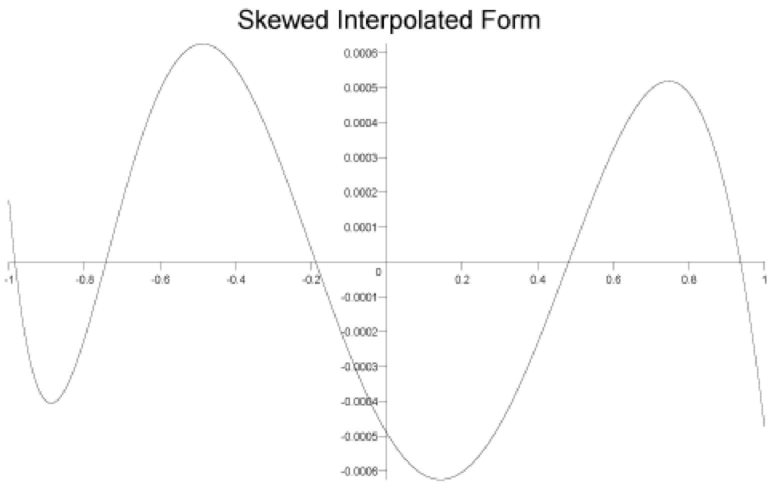
as the approximating form seems like an obvious thing to try, and does indeed yield a useful approximation.

However, the difficulty then becomes one of converging the minimax solution. Unfortunately, it is known that for some functions the Remez method can lead to divergent behaviour, even when the initial starting approximation is quite good. Furthermore, it is not uncommon for the solution obtained in the first Remez step above to be a bad one: the equations to be solved are generally "stiff", often very close to being singular, and assuming a solution is found at all, round-off errors and a rapidly changing error function, can lead to a situation where the error function does not in fact change sign at each control point as required. If this occurs, it is fatal to the Remez method. It is also possible to obtain solutions that are perfectly valid mathematically, but which are quite useless computationally: either because there is an unavoidable amount of roundoff error in the computation of the rational function, or because the denominator has one or more roots over the interval of the approximation. In the latter case while the approximation may have the correct limiting value at the roots, the approximation is nonetheless useless.

Assuming that the approximation does not have any fatal errors, and that the only issue is converging adequately on the minimax solution, the aim is to get as close as possible to the minimax solution before beginning the Remez method. Using the zeros of a Chebyshev polynomial for the initial interpolation is a good start, but may not be ideal when dealing with relative errors and/or rational (rather than polynomial) approximations. One approach is to skew the initial interpolation points to one end: for example if we raise the roots of the Chebyshev polynomial to a positive power greater than 1 then the roots will be skewed towards the middle of the [-1,1] interval, while a positive power less than one will skew them towards either end. More usefully, if we initially rescale the points over [0,1] and then raise to a positive power, we can skew them to the left or right. Returning to our example of e^x over [-1,1], the initial interpolated form was some way from the minimax solution:



However, if we first skew the interpolation points to the left (rescale them to [0, 1], raise to the power 1.3, and then rescale back to [-1,1]) we reduce the error from 1.3x10^{-3} to 6x10^{-4}:



It's clearly still not ideal, but it is only a few percent away from our desired minimax solution (5x10^{-4}).

Remez Method Checklist

The following lists some of the things to check if the Remez method goes wrong, it is by no means an exhaustive list, but is provided in the hopes that it will prove useful.

- Is the function smooth enough? Can it be better separated into a rapidly changing part, and an asymptotic part?
- Does the function being approximated have any "blips" in it? Check for problems as the function changes computation method, or if a root, or an infinity has been divided out. The telltale sign is if there is a narrow region where the Remez method will not converge.

- Check you have enough accuracy in your calculations: remember that the Remez method works on the difference between the approximation and the function being approximated: so you must have more digits of precision available than the precision of the approximation being constructed. So for example at double precision, you shouldn't expect to be able to get better than a float precision approximation.
- Try skewing the initial interpolated approximation to minimise the error before you begin the Remez steps.
- If the approximation won't converge or is ill-conditioned from one starting location, try starting from a different location.
- If a rational function won't converge, one can minimise a polynomial (which presents no problems), then rotate one term from the numerator to the denominator and minimise again. In theory one can continue moving terms one at a time from numerator to denominator, and then re-minimising, retaining the last set of control points at each stage.
- Try using a smaller interval. It may also be possible to optimise over one (small) interval, rescale the control points over a larger interval, and then re-minimise.
- Keep absissa values small: use a change of variable to keep the abscissa over, say [0, b], for some smallish value *b*.

References

The original references for the Remez Method and it's extension to rational functions are unfortunately in Russian:

Remez, E.Ya., *Fundamentals of numerical methods for Chebyshev approximations*, "Naukova Dumka", Kiev, 1969.

Remez, E.Ya., Gavrilyuk, V.T., *Computer development of certain approaches to the approximate construction of solutions of Chebyshev problems nonlinearly depending on parameters*, Ukr. Mat. Zh. 12 (1960), 324-338.

Gavrilyuk, V.T., *Generalization of the first polynomial algorithm of E.Ya.Remez for the problem of constructing rational-fractional Chebyshev approximations*, Ukr. Mat. Zh. 16 (1961), 575-585.

Some English language sources include:

Fraser, W., Hart, J.F., *On the computation of rational approximations to continuous functions*, Comm. of the ACM 5 (1962), 401-403, 414.

Ralston, A., *Rational Chebyshev approximation by Remes' algorithms*, Numer.Math. 7 (1965), no. 4, 322-330.

A. Ralston, *Rational Chebyshev approximation, Mathematical Methods for Digital Computers v. 2* (Ralston A., Wilf H., eds.), Wiley, New York, 1967, pp. 264-284.

Hart, J.F. e.a., *Computer approximations*, Wiley, New York a.o., 1968.

Cody, W.J., Fraser, W., Hart, J.F., *Rational Chebyshev approximation using linear equations*, Numer.Math. 12 (1968), 242-251.

Cody, W.J., *A survey of practical rational and polynomial approximation of functions*, SIAM Review 12 (1970), no. 3, 400-423.

Barrar, R.B., Loeb, H.J., *On the Remez algorithm for non-linear families*, Numer.Math. 15 (1970), 382-391.

Dunham, Ch.B., *Convergence of the Fraser-Hart algorithm for rational Chebyshev approximation*, Math. Comp. 29 (1975), no. 132, 1078-1082.

G. L. Litvinov, *Approximate construction of rational approximations and the effect of error autocorrection*, Russian Journal of Mathematical Physics, vol.1, No. 3, 1994.

References

General references

(Specific detailed sources for individual functions and distributions are given at the end of each individual section).

DLMF (NIST Digital Library of Mathematical Functions) is a replacement for the legendary Abramowitz and Stegun's Handbook of Mathematical Functions (often called simply A&S),

M. Abramowitz and I. A. Stegun (Eds.) (1964) Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, National Bureau of Standards Applied Mathematics Series, U.S. Government Printing Office, Washington, D.C.

NIST Handbook of Mathematical Functions Edited by: Frank W. J. Olver, University of Maryland and National Institute of Standards and Technology, Maryland, Daniel W. Lozier, National Institute of Standards and Technology, Maryland, Ronald F. Boisvert, National Institute of Standards and Technology, Maryland, Charles W. Clark, National Institute of Standards and Technology, Maryland and University of Maryland.

ISBN: 978-0521140638 (paperback), 9780521192255 (hardback), July 2010, Cambridge University Press.

NIST/SEMATECH e-Handbook of Statistical Methods

Mathematica Documentation: DiscreteDistributions The Wolfram Research Documentation Center is a collection of online reference materials about Mathematica, CalculationCenter, and other Wolfram Research products.

Mathematica Documentation: ContinuousDistributions The Wolfram Research Documentation Center is a collection of online reference materials about Mathematica, CalculationCenter, and other Wolfram Research products.

Statistical Distributions (Wiley Series in Probability & Statistics) (Paperback) by N.A.J. Hastings, Brian Peacock, Merran Evans, ISBN: 0471371246, Wiley 2000.

Extreme Value Distributions, Theory and Applications Samuel Kotz & Saralees Nadarajah, ISBN 978-1-86094-224-2 & 1-86094-224-5 Oct 2000, Chapter 1.2 discusses the various extreme value distributions.

pugh.pdf (application/pdf Object) Pugh Msc Thesis on the Lanczos approximation to the gamma function.

N1514, 03-0097, A Proposal to Add Mathematical Special Functions to the C++ Standard Library (version 2), Walter E. Brown

Calculators

We found (and used to create cross-check spot values - as far as their accuracy allowed).

The Wolfram Functions Site The Wolfram Functions Site - Providing the mathematical and scientific community with the world's largest (and most authorititive) collection of formulas and graphics about mathematical functions.

100-decimal digit calculator provided some spot values.

http://www.adsciengineering.com/bpdcalc/ Binomial Probability Distribution Calculator.

Other Libraries

Cephes library by Shephen Moshier and his book:

Methods and programs for mathematical functions, Stephen L B Moshier, Ellis Horwood (1989) ISBN 0745802893 0470216093 provided inspiration.

CDFLIB Library of Fortran Routines for Cumulative Distribution functions.

DCFLIB C++ version.

DCDFLIB C++ version DCDFLIB is a library of C++ routines, using double precision arithmetic, for evaluating cumulative probability density functions.

http://www.softintegration.com/docs/package/chnagstat/

NAG libraries.

MathCAD

JMSL Numerical Library (Java).

John F Hart, Computer Approximations, (1978) ISBN 0 088275 642-7.

William J Cody, Software Manual for the Elementary Functions, Prentice-Hall (1980) ISBN 0138220646.

Nico Temme, Special Functions, An Introduction to the Classical Functions of Mathematical Physics, Wiley, ISBN: 0471-11313-1 (1996) who also gave valueable advice.

Statistics Glossary, Valerie Easton and John H. McColl.

_R R Development Core Team (2010). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

For use of R, see:

Jim Albert, Bayesian Computation with R, ISBN 978-0-387-71384-7.

C++ Statistical Distributions in Boost - QuantNetwork forum discusses using Boost.Math in finance.

Quantnet Boost and computational finance. Robert Demming & Daniel J. Duffy, Introduction to the C++ Boost Libraries - Volume I - Foundations and Volume II ISBN 978-94-91028-01-4, Advanced Libraries and Applications, ISBN 978-94-91028-02-1 (to be published in 2011). discusses application of Boost.Math, especially in finance.

Error logs and tables

Tables of Error Rates for all Functions

Table 134. Error rates for beta

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = +INFε (Mean = +INFε) And other failures.) (*Rmath 3.2.3:* Max = 1.14ε (Mean = 0.574ε))	Max = 2.86ε (Mean = 1.22ε) (*<cmath>:* Max = 364ε (Mean = 76.2ε))	Max = 2.86ε (Mean = 1.22ε)	Max = 2.23ε (Mean = 1.14ε)
Beta Function: Medium Values	Max = 0.978ε (Mean = 0.0595ε) (*GSL 2.1:* Max = 1.18e+03ε (Mean = 238ε)) (*Rmath 3.2.3:* Max = 1.09e+03ε (Mean = 265ε))	Max = 61.4ε (Mean = 19.4ε) (*<cmath>:* Max = 1.07e+03ε (Mean = 264ε))	Max = 107ε (Mean = 24.5ε)	Max = 96.5ε (Mean = 22.4ε)
Beta Function: Divergent Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 12.1ε (Mean = 1.99ε)) (*Rmath 3.2.3:* Max = 176ε (Mean = 28ε))	Max = 8.99ε (Mean = 2.44ε) (*<cmath>:* Max = 128ε (Mean = 23.8ε))	Max = 18.8ε (Mean = 2.71ε)	Max = 11.4ε (Mean = 2.19ε)

Table 135. Error rates for beta (incomplete)

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε)	Max = 11.1ε (Mean = 2.32ε)	Max = 18.7ε (Mean = 3.19ε)	Max = 9.94ε (Mean = 2.17ε)
Incomplete Beta Function: Medium Values	Max = 0.568ε (Mean = 0.0254ε)	Max = 69.2ε (Mean = 13.4ε)	Max = 174ε (Mean = 25ε)	Max = 90ε (Mean = 12.7ε)
Incomplete Beta Function: Large and Diverse Values	Max = 0.999ε (Mean = 0.0325ε)	Max = 6.84e+04ε (Mean = 2.76e+03ε)	Max = 6.86e+04ε (Mean = 2.79e+03ε)	Max = 633ε (Mean = 29.7ε)
Incomplete Beta Function: Small Integer Values	Max = 0.786ε (Mean = 0.0323ε)	Max = 11.6ε (Mean = 3.6ε)	Max = 51.8ε (Mean = 11ε)	Max = 26ε (Mean = 6.28ε)

Table 136. Error rates for betac

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Incomplete Beta Function: Small Values	Max = 0.676ε (Mean = 0.0302ε)	Max = 9.92ε (Mean = 2.3ε)	Max = 11.2ε (Mean = 2.94ε)	Max = 8.94ε (Mean = 2.06ε)
Incomplete Beta Function: Medium Values	Max = 0.949ε (Mean = 0.098ε)	Max = 63.5ε (Mean = 13.5ε)	Max = 97.6ε (Mean = 24.3ε)	Max = 90.6ε (Mean = 14.8ε)
Incomplete Beta Function: Large and Diverse Values	Max = 1.12ε (Mean = 0.0458ε)	Max = 1.05e+05ε (Mean = 5.45e+03ε)	Max = 1.04e+05ε (Mean = 5.46e+03ε)	Max = 3.72e+03ε (Mean = 113ε)
Incomplete Beta Function: Small Integer Values	Max = 0.586ε (Mean = 0.0314ε)	Max = 11.1ε (Mean = 3.65ε)	Max = 103ε (Mean = 17.4ε)	Max = 26.2ε (Mean = 6.36ε)

Table 137. Error rates for binomial_coefficient

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Binomials: small arguments	Max = 1ε (Mean = 0.369ε)	Max = 1.5ε (Mean = 0.339ε)	Max = 1.5ε (Mean = 0.339ε)	Max = 1ε (Mean = 0.369ε)
Binomials: large arguments	Max = 0.939ε (Mean = 0.314ε)	Max = 26.6ε (Mean = 6.13ε)	Max = 53.2ε (Mean = 10.8ε)	Max = 37.2ε (Mean = 7.4ε)

Table 138. Error rates for boost::math::powm1

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
powm1	(*<math.h>:* Max = 2.04ε (Mean = 0.493ε)) Max = 2.04ε (Mean = 0.493ε) (*<cmath>:* Max = 2.04ε (Mean = 0.493ε))	(*Rmath 3.2.3:* Max = 1.06ε (Mean = 0.425ε)) (*GSL 2.1:* Max = 1.06ε (Mean = 0.425ε)) Max = 1.06ε (Mean = 0.425ε)	Max = 1.88ε (Mean = 0.49ε) (*<math.h>:* Max = 1.88ε (Mean = 0.49ε))	(*<math.h>:* Max = 1.84ε (Mean = 0.486ε)) Max = 1.84ε (Mean = 0.486ε)

Table 139. Error rates for cbrt

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
cbrt Function	Max = 0ε (Mean = 0ε)	Max = 1.34ε (Mean = 0.471ε) (*<cmath>:* Max = 1.34ε (Mean = 0.471ε)) (*<math.h>:* Max = 1.34ε (Mean = 0.471ε))	Max = 1.34ε (Mean = 0.471ε) (*<math.h>:* Max = 1.34ε (Mean = 0.471ε))	Max = 1.7ε (Mean = 0.565ε) (*<math.h>:* Max = 1.7ε (Mean = 0.565ε))

Table 140. Error rates for cos_pi

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
sin_pi and cos_pi	Max = 0ε (Mean = 0ε)	Max = 0.991ε (Mean = 0.302ε)	Max = 0.991ε (Mean = 0.302ε)	Max = 0.996ε (Mean = 0.284ε)
sin_pi and cos_pi near integers and half integers	Max = 0ε (Mean = 0ε)	Max = 0.976ε (Mean = 0.28ε)	Max = 0.976ε (Mean = 0.28ε)	Max = 0.996ε (Mean = 0.298ε)

Table 141. Error rates for cyl_bessel_i

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel I0: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 270ε (Mean = 91.6ε) And other failures.) (*Rmath 3.2.3:* Max = 1.52ε (Mean = 0.622ε) And other failures.)	Max = 1.95ε (Mean = 0.738ε) (*<cmath>:* Max = 8.49ε (Mean = 3.46ε) And other failures.)	Max = 1.95ε (Mean = 0.661ε)	Max = 0.762ε (Mean = 0.329ε)
Bessel I1: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 128ε (Mean = 41ε) And other failures.) (*Rmath 3.2.3:* Max = 1.53ε (Mean = 0.483ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε) (*<cmath>:* Max = 5ε (Mean = 2.15ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε)	Max = 0.767ε (Mean = 0.398ε)
Bessel In: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.31ε (Mean = 0.838ε) And other failures.) (*Rmath 3.2.3:* Max = 1.73ε (Mean = 0.601ε) And other failures.)	Max = 1.8ε (Mean = 1.33ε) (*<cmath>:* Max = 430ε (Mean = 163ε) And other failures.)	Max = 463ε (Mean = 140ε)	Max = 3.46ε (Mean = 1.32ε)
Bessel Iv: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.95ε (Mean = 2.08ε) And other failures.) (*Rmath 3.2.3:* Max = 3.53ε (Mean = 1.39ε))	Max = 4.12ε (Mean = 1.85ε) (*<cmath>:* Max = 616ε (Mean = 221ε) And other failures.)	Max = 4.12ε (Mean = 1.95ε)	Max = 2.97ε (Mean = 1.24ε)
Bessel In: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 261ε (Mean = 53.2ε) And other failures.) (*Rmath 3.2.3:* Max = 7.37ε (Mean = 2.4ε))	Max = 4.62ε (Mean = 1.06ε) (*<cmath>:* Max = 645ε (Mean = 132ε))	Max = 176ε (Mean = 39.1ε)	Max = 9.67ε (Mean = 1.88ε)
Bessel Iv: Random Data	Max = 0.661ε (Mean = 0.0441ε) (*GSL 2.1:* Max = 6.18e+03ε (Mean = 1.55e+03ε) And other failures.) (*Rmath 3.2.3:* Max = 4.28e+08ε (Mean = 2.85e+07ε))	Max = 8.35ε (Mean = 1.62ε) (*<cmath>:* Max = 1.05e+03ε (Mean = 224ε) And other failures.)	Max = 283ε (Mean = 88.4ε)	Max = 7.46ε (Mean = 1.71ε)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel Iv: Mathworld Data (large values)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 37ε (Mean = 18ε) And other failures.) (*Rmath 3.2.3:* Max = 3.77e+168ε (Mean = 2.39e+168ε) And other failures.)	Max = 14.7ε (Mean = 6.66ε) (*<cmath>:* Max = 118ε (Mean = 57.2ε) And other failures.)	Max = 14.7ε (Mean = 6.59ε)	Max = 3.67ε (Mean = 1.64ε)

Table 142. Error rates for cyl_bessel_i (integer orders)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel I0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.79ε (Mean = 0.482ε)) (*Rmath 3.2.3:* Max = 1.52ε (Mean = 0.622ε) And other failures.)	Max = 1.95ε (Mean = 0.738ε) (*<cmath>:* Max = 8.49ε (Mean = 3.46ε) And other failures.)	Max = 1.95ε (Mean = 0.661ε)	Max = 0.762ε (Mean = 0.329ε)
Bessel I1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.82ε (Mean = 0.456ε)) (*Rmath 3.2.3:* Max = 1.53ε (Mean = 0.483ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε) (*<cmath>:* Max = 5ε (Mean = 2.15ε) And other failures.)	Max = 0.64ε (Mean = 0.202ε)	Max = 0.767ε (Mean = 0.398ε)
Bessel In: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.15ε (Mean = 2.13ε) And other failures.) (*Rmath 3.2.3:* Max = 1.73ε (Mean = 0.601ε) And other failures.)	Max = 1.8ε (Mean = 1.33ε) (*<cmath>:* Max = 430ε (Mean = 163ε) And other failures.)	Max = 463ε (Mean = 140ε)	Max = 3.46ε (Mean = 1.32ε)

Table 143. Error rates for cyl_bessel_i_prime

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel I'0: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.354ε)
Bessel I'1: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.36ε (Mean = 0.782ε)
Bessel I'n: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 2.31ε (Mean = 1.41ε)	Max = 701ε (Mean = 212ε)	Max = 3.61ε (Mean = 1.22ε)
Bessel I'v: Mathworld Data	Max = 1.62ε (Mean = 0.512ε)	Max = 2.89e+03ε (Mean = 914ε)	Max = 2.89e+03ε (Mean = 914ε)	Max = 3.76e+03ε (Mean = 1.19e+03ε)
Bessel I'n: Random Data	Max = 0ε (Mean = 0ε)	Max = 3.95ε (Mean = 1.06ε)	Max = 195ε (Mean = 37.1ε)	Max = 9.85ε (Mean = 1.82ε)
Bessel I'v: Random Data	Max = 0ε (Mean = 0ε)	Max = 14.1ε (Mean = 2.93ε)	Max = 336ε (Mean = 68.5ε)	Max = 14ε (Mean = 2.5ε)
Bessel I'v: Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 42.6ε (Mean = 20.2ε)	Max = 42.6ε (Mean = 20.2ε)	Max = 59.5ε (Mean = 26.6ε)

Table 144. Error rates for cyl_bessel_i_prime (integer orders)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel I'0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.259ε)	Max = 0.82ε (Mean = 0.354ε)
Bessel I'1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.97ε (Mean = 0.757ε)	Max = 1.36ε (Mean = 0.782ε)
Bessel I'n: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 2.31ε (Mean = 1.41ε)	Max = 701ε (Mean = 212ε)	Max = 3.61ε (Mean = 1.22ε)

Table 145. Error rates for cyl_bessel_j

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel J0: Mathworld Data	Max = 6.55ε (Mean = 2.86ε) (*<cmath>:* Max = 5.04ε (Mean = 1.78ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.629ε (Mean = 0.223ε) And other failures.) (*Rmath 3.2.3:* Max = 0.629ε (Mean = 0.223ε) And other failures.)	Max = 6.55ε (Mean = 2.86ε)	Max = 2.52ε (Mean = 1.2ε)
Bessel J0: Mathworld Data (Tricky cases)	Max = 1.64e+08ε (Mean = 6.69e+07ε) (*<cmath>:* Max = 4.79e+08ε (Mean = 1.96e+08ε))	Max = 8e+04ε (Mean = 3.27e+04ε) (*GSL 2.1:* Max = 6.5e+07ε (Mean = 2.66e+07ε)) (*Rmath 3.2.3:* Max = 1.04e+07ε (Mean = 4.29e+06ε))	Max = 1.64e+08ε (Mean = 6.69e+07ε)	Max = 1e+07ε (Mean = 4.09e+06ε)
Bessel J1: Mathworld Data	Max = 3.59ε (Mean = 1.33ε) (*<cmath>:* Max = 6.1ε (Mean = 2.95ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.62ε (Mean = 2.35ε) And other failures.) (*Rmath 3.2.3:* Max = 0.946ε (Mean = 0.39ε) And other failures.)	Max = 1.44ε (Mean = 0.637ε)	Max = 1.73ε (Mean = 0.976ε)
Bessel J1: Mathworld Data (tricky cases)	Max = 2.18e+05ε (Mean = 9.76e+04ε) (*<cmath>:* Max = 2.15e+06ε (Mean = 1.58e+06ε))	Max = 106ε (Mean = 47.5ε) (*GSL 2.1:* Max = 8.75e+05ε (Mean = 5.32e+05ε)) (*Rmath 3.2.3:* Max = 2.93e+06ε (Mean = 1.7e+06ε))	Max = 2.18e+05ε (Mean = 9.76e+04ε)	Max = 3.23e+04ε (Mean = 1.45e+04ε)
Bessel JN: Mathworld Data	Max = 6.85ε (Mean = 3.35ε) (*<cmath>:* Max = 2.13e+19ε (Mean = 5.16e+18ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.9e+05ε (Mean = 2.15e+05ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 463ε (Mean = 112ε)	Max = 14.7ε (Mean = 5.4ε)

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Bessel J: Mathworld Data	Max = 14.7ε (Mean = 4.11ε) (*<cmath>:* Max = 3.49e+05ε (Mean = 8.09e+04ε) And other failures.)	Max = 10ε (Mean = 2.24ε) (*GSL 2.1:* Max = 2.39e+05ε (Mean = 5.37e+04ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 14.7ε (Mean = 4.22ε)	Max = 14.9ε (Mean = 3.89ε)
Bessel J: Mathworld Data (large values)	Max = 607ε (Mean = 305ε) (*<cmath>:* Max = 34.9ε (Mean = 17.4ε) And other failures.)	Max = 0.536ε (Mean = 0.268ε) (*GSL 2.1:* Max = 4.91e+03ε (Mean = 2.46e+03ε) And other failures.) (*Rmath 3.2.3:* Max = 5.9ε (Mean = 3.76ε))	Max = 607ε (Mean = 305ε)	Max = 9.31ε (Mean = 5.52ε)
Bessel JN: Random Data	Max = 50.8ε (Mean = 3.69ε) (*<cmath>:* Max = 1.12e+03ε (Mean = 88.7ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 75.7ε (Mean = 5.36ε)) (*Rmath 3.2.3:* Max = 3.93ε (Mean = 1.22ε))	Max = 99.6ε (Mean = 22ε)	Max = 17.5ε (Mean = 1.46ε)
Bessel J: Random Data	Max = 11.4ε (Mean = 1.68ε) (*<cmath>:* Max = 501ε (Mean = 52.3ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 15.5ε (Mean = 3.33ε) And other failures.) (*Rmath 3.2.3:* Max = 6.74ε (Mean = 1.3ε))	Max = 260ε (Mean = 34ε)	Max = 9.24ε (Mean = 1.17ε)
Bessel J: Random Data (Tricky large values)	Max = 785ε (Mean = 94.2ε) (*<cmath>:* Max = 5.01e+17ε (Mean = 6.23e+16ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.48e+05ε (Mean = 5.11e+04ε)) (*Rmath 3.2.3:* Max = 71.6ε (Mean = 11.7ε))	Max = 785ε (Mean = 97.4ε)	Max = 59.2ε (Mean = 8.67ε)

Table 146. Error rates for cyl_bessel_j (integer orders)

	GNU C++ version 7.1.0 **linux** **long double**	**GNU C++ version 7.1.0** **linux** **double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Bessel J0: Mathworld Data (Integer Version)	Max = 6.55ε (Mean = 2.86ε) (*<cmath>:* Max = 5.04ε (Mean = 1.78ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.12ε (Mean = 0.488ε)) (*Rmath 3.2.3:* Max = 0.629ε (Mean = 0.223ε) And other failures.)	Max = 6.55ε (Mean = 2.86ε)	Max = 2.52ε (Mean = 1.2ε) (*<math.h>:* Max = 1.89ε (Mean = 0.988ε))
Bessel J0: Mathworld Data (Tricky cases) (Integer Version)	Max = 1.64e+08ε (Mean = 6.69e+07ε) (*<cmath>:* Max = 4.79e+08ε (Mean = 1.96e+08ε))	Max = 8e+04ε (Mean = 3.27e+04ε) (*GSL 2.1:* Max = 1e+07ε (Mean = 4.11e+06ε)) (*Rmath 3.2.3:* Max = 1.04e+07ε (Mean = 4.29e+06ε))	Max = 1.64e+08ε (Mean = 6.69e+07ε)	Max = 1e+07ε (Mean = 4.09e+06ε) (*<math.h>:* Max = 2.54e+08ε (Mean = 1.04e+08ε))
Bessel J1: Mathworld Data (Integer Version)	Max = 3.59ε (Mean = 1.33ε) (*<cmath>:* Max = 6.1ε (Mean = 2.95ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.89ε (Mean = 0.721ε)) (*Rmath 3.2.3:* Max = 0.946ε (Mean = 0.39ε) And other failures.)	Max = 1.44ε (Mean = 0.637ε)	Max = 1.73ε (Mean = 0.976ε) (*<math.h>:* Max = 11.4ε (Mean = 4.15ε))
Bessel J1: Mathworld Data (tricky cases) (Integer Version)	Max = 2.18e+05ε (Mean = 9.76e+04ε) (*<cmath>:* Max = 2.15e+06ε (Mean = 1.58e+06ε))	Max = 106ε (Mean = 47.5ε) (*GSL 2.1:* Max = 1.26e+06ε (Mean = 6.28e+05ε)) (*Rmath 3.2.3:* Max = 2.93e+06ε (Mean = 1.7e+06ε))	Max = 2.18e+05ε (Mean = 9.76e+04ε)	Max = 3.23e+04ε (Mean = 1.45e+04ε) (*<math.h>:* Max = 1.44e+07ε (Mean = 6.5e+06ε))
Bessel JN: Mathworld Data (Integer Version)	Max = 6.85ε (Mean = 3.35ε) (*<cmath>:* Max = 2.13e+19ε (Mean = 5.16e+18ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.9e+05ε (Mean = 2.53e+05ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 463ε (Mean = 112ε)	Max = 14.7ε (Mean = 5.4ε) (*<math.h>:* Max = +INFε (Mean = +INFε) And other failures.)

Table 147. Error rates for cyl_bessel_j_prime

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Bessel J0': Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 18.9ε (Mean = 6.82ε)	Max = 18.9ε (Mean = 6.72ε)	Max = 6.62ε (Mean = 2.55ε)
Bessel J0': Mathworld Data (Tricky cases)	Max = 0ε (Mean = 0ε)	Max = 7.44ε (Mean = 3.34ε)	Max = 7.44ε (Mean = 3.31ε)	Max = 3.67ε (Mean = 1.74ε)
Bessel J1': Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 0.999ε (Mean = 0.627ε)
Bessel J1': Mathworld Data (tricky cases)	Max = 287ε (Mean = 129ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 288ε (Mean = 129ε)
Bessel JN': Mathworld Data	Max = 0.527ε (Mean = 0.128ε)	Max = 1.29e+03ε (Mean = 312ε)	Max = 1.29e+03ε (Mean = 355ε)	Max = 14ε (Mean = 6.13ε)
Bessel J': Mathworld Data	Max = 21.5ε (Mean = 4.7ε)	Max = 42.5ε (Mean = 9.31ε)	Max = 42.5ε (Mean = 9.32ε)	Max = 23.7ε (Mean = 8ε)
Bessel J': Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 989ε (Mean = 495ε)	Max = 989ε (Mean = 495ε)	Max = 2.9ε (Mean = 1.61ε)
Bessel JN': Random Data	Max = 0.593ε (Mean = 0.0396ε)	Max = 11.3ε (Mean = 1.85ε)	Max = 79.4ε (Mean = 16.2ε)	Max = 6.34ε (Mean = 0.999ε)
Bessel J': Random Data	Max = 0.885ε (Mean = 0.033ε)	Max = 139ε (Mean = 6.47ε)	Max = 279ε (Mean = 27.2ε)	Max = 176ε (Mean = 9.75ε)
Bessel J': Random Data (Tricky large values)	Max = 0ε (Mean = 0ε)	Max = 474ε (Mean = 62.2ε)	Max = 474ε (Mean = 64.5ε)	Max = 379ε (Mean = 45.4ε)

Table 148. Error rates for cyl_bessel_j_prime (integer orders)

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Bessel J0': Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 18.9ε (Mean = 6.82ε)	Max = 18.9ε (Mean = 6.72ε)	Max = 6.62ε (Mean = 2.55ε)
Bessel J0': Mathworld Data (Tricky cases) (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 7.44ε (Mean = 3.34ε)	Max = 7.44ε (Mean = 3.31ε)	Max = 3.67ε (Mean = 1.74ε)
Bessel J1': Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 7.9ε (Mean = 3.37ε)	Max = 0.999ε (Mean = 0.627ε)
Bessel J1': Mathworld Data (tricky cases) (Integer Version)	Max = 287ε (Mean = 129ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 5.88e+05ε (Mean = 2.63e+05ε)	Max = 288ε (Mean = 129ε)
Bessel JN': Mathworld Data (Integer Version)	Max = 0.527ε (Mean = 0.128ε)	Max = 1.29e+03ε (Mean = 312ε)	Max = 1.29e+03ε (Mean = 355ε)	Max = 14ε (Mean = 6.13ε)

Table 149. Error rates for cyl_bessel_k

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K0: Mathworld Data	Max = 0.833ε (Mean = 0.436ε) (*<cmath>:* Max = 9.33ε (Mean = 3.25ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.04ε (Mean = 2.16ε)) (*Rmath 3.2.3:* Max = 0.833ε (Mean = 0.601ε))	Max = 0.833ε (Mean = 0.436ε)	Max = 0.833ε (Mean = 0.552ε)
Bessel K1: Mathworld Data	Max = 0.786ε (Mean = 0.329ε) (*<cmath>:* Max = 8.94ε (Mean = 3.19ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.26ε (Mean = 2.21ε)) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.516ε))	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel Kn: Mathworld Data	Max = 2.6ε (Mean = 1.21ε) (*<cmath>:* Max = 12.9ε (Mean = 4.91ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.36ε (Mean = 1.43ε) And other failures.) (*Rmath 3.2.3:* Max = 8.48ε (Mean = 2.98ε))	Max = 2.6ε (Mean = 1.21ε)	Max = 3.63ε (Mean = 1.46ε)
Bessel Kv: Mathworld Data	Max = 3.58ε (Mean = 2.39ε) (*<cmath>:* Max = 13ε (Mean = 4.81ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.47ε (Mean = 2.04ε) And other failures.) (*Rmath 3.2.3:* Max = 3.15ε (Mean = 1.35ε))	Max = 5.21ε (Mean = 2.53ε)	Max = 4.78ε (Mean = 2.19ε)
Bessel Kv: Mathworld Data (large values)	Max = 42.3ε (Mean = 21ε) (*<cmath>:* Max = 42.3ε (Mean = 19.8ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 308ε (Mean = 142ε) And other failures.) (*Rmath 3.2.3:* Max = 84.6ε (Mean = 37.8ε))	Max = 42.3ε (Mean = 21ε)	Max = 59.8ε (Mean = 26.9ε)
Bessel Kn: Random Data	Max = 4.55ε (Mean = 1.12ε) (*<cmath>:* Max = 13.9ε (Mean = 2.91ε))	Max = 0.764ε (Mean = 0.0348ε) (*GSL 2.1:* Max = 8.71ε (Mean = 1.76ε) And other failures.) (*Rmath 3.2.3:* Max = 7.47ε (Mean = 1.34ε))	Max = 4.55ε (Mean = 1.12ε)	Max = 9.34ε (Mean = 1.7ε)
Bessel Kv: Random Data	Max = 7.88ε (Mean = 1.48ε) (*<cmath>:* Max = 13.6ε (Mean = 2.68ε) And other failures.)	Max = 0.507ε (Mean = 0.0313ε) (*GSL 2.1:* Max = 9.71ε (Mean = 1.47ε) And other failures.) (*Rmath 3.2.3:* Max = 7.37ε (Mean = 1.49ε))	Max = 7.88ε (Mean = 1.47ε)	Max = 8.33ε (Mean = 1.62ε)

Table 150. Error rates for cyl_bessel_k (integer orders)

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K0: Mathworld Data (Integer Version)	Max = 0.833ε (Mean = 0.436ε) (*<cmath>:* Max = 9.33ε (Mean = 3.25ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.2ε (Mean = 0.733ε)) (*Rmath 3.2.3:* Max = 0.833ε (Mean = 0.601ε))	Max = 0.833ε (Mean = 0.436ε)	Max = 0.833ε (Mean = 0.552ε)
Bessel K1: Mathworld Data (Integer Version)	Max = 0.786ε (Mean = 0.329ε) (*<cmath>:* Max = 8.94ε (Mean = 3.19ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.626ε (Mean = 0.333ε)) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.516ε))	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel Kn: Mathworld Data (Integer Version)	Max = 2.6ε (Mean = 1.21ε) (*<cmath>:* Max = 12.9ε (Mean = 4.91ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 168ε (Mean = 59.5ε)) (*Rmath 3.2.3:* Max = 8.48ε (Mean = 2.98ε))	Max = 2.6ε (Mean = 1.21ε)	Max = 3.63ε (Mean = 1.46ε)

Table 151. Error rates for cyl_bessel_k_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K'0: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel K'1: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.761ε (Mean = 0.444ε)
Bessel K'n: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 4.17ε (Mean = 1.75ε)
Bessel K'v: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 3.94ε (Mean = 2.44ε)	Max = 3.94ε (Mean = 2.34ε)	Max = 3.94ε (Mean = 1.47ε)
Bessel K'v: Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 59.2ε (Mean = 42.9ε)	Max = 58.7ε (Mean = 42.6ε)	Max = 18.6ε (Mean = 11.8ε)
Bessel K'n: Random Data	Max = 0ε (Mean = 0ε)	Max = 4.45ε (Mean = 1.19ε)	Max = 4.45ε (Mean = 1.19ε)	Max = 9.67ε (Mean = 1.73ε)
Bessel K'v: Random Data	Max = 0ε (Mean = 0ε)	Max = 7.95ε (Mean = 1.53ε)	Max = 7.95ε (Mean = 1.52ε)	Max = 8.32ε (Mean = 1.65ε)

Table 152. Error rates for cyl_bessel_k_prime (integer orders)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel K'0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.329ε)	Max = 0.786ε (Mean = 0.39ε)
Bessel K'1: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.736ε (Mean = 0.389ε)	Max = 0.761ε (Mean = 0.444ε)
Bessel K'n: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 2.16ε (Mean = 1.08ε)	Max = 4.17ε (Mean = 1.75ε)

Table 153. Error rates for cyl_neumann

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Y0: Mathworld Data	Max = 5.53ε (Mean = 2.4ε) (*<cmath>:* Max = 2.05e+05ε (Mean = 6.87e+04ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 60.9ε (Mean = 20.4ε)) (*Rmath 3.2.3:* Max = 167ε (Mean = 56.5ε))	Max = 5.53ε (Mean = 2.4ε)	Max = 4.61ε (Mean = 2.29ε)
Y1: Mathworld Data	Max = 6.33ε (Mean = 2.25ε) (*<cmath>:* Max = 9.71e+03ε (Mean = 4.08e+03ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 23.4ε (Mean = 8.1ε)) (*Rmath 3.2.3:* Max = 193ε (Mean = 64.4ε))	Max = 6.33ε (Mean = 2.29ε)	Max = 4.75ε (Mean = 1.72ε)
Yn: Mathworld Data	Max = 55.2ε (Mean = 17.8ε) (*<cmath>:* Max = 2.2e+20ε (Mean = 6.97e+19ε) And other failures.)	Max = 0.993ε (Mean = 0.314ε) (*GSL 2.1:* Max = 2.41e+05ε (Mean = 7.62e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 1.24e+04ε (Mean = 4e+03ε))	Max = 55.2ε (Mean = 17.8ε)	Max = 35ε (Mean = 11.9ε)
Yv: Mathworld Data	Max = 10.7ε (Mean = 4.93ε) (*<cmath>:* Max = 3.49e+15ε (Mean = 1.05e+15ε) And other failures.)	Max = 10ε (Mean = 3.02ε) (*GSL 2.1:* Max = 1.07e+05ε (Mean = 3.22e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 243ε (Mean = 73.9ε))	Max = 10.7ε (Mean = 5.1ε)	Max = 7.89ε (Mean = 3.27ε)
Yv: Mathworld Data (large values)	Max = 1.7ε (Mean = 1.33ε) (*<cmath>:* Max = 43.2ε (Mean = 16.3ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 60.8ε (Mean = 23ε) And other failures.) (*Rmath 3.2.3:* Max = 0.682ε (Mean = 0.335ε))	Max = 1.7ε (Mean = 1.33ε)	Max = 0.682ε (Mean = 0.423ε)
Y0 and Y1: Random Data	Max = 10.8ε (Mean = 3.04ε) (*<cmath>:* Max = 2.59e+03ε (Mean = 500ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 34.4ε (Mean = 8.9ε)) (*Rmath 3.2.3:* Max = 83ε (Mean = 14.2ε))	Max = 10.8ε (Mean = 3.04ε)	Max = 4.17ε (Mean = 1.24ε)
Yn: Random Data	Max = 338ε (Mean = 27.5ε) (*<cmath>:* Max = 4.01e+03ε (Mean = 348ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 500ε (Mean = 47.8ε)) (*Rmath 3.2.3:* Max = 691ε (Mean = 67.9ε))	Max = 338ε (Mean = 27.5ε)	Max = 117ε (Mean = 10.2ε)

	GNU C++ version 7.1.0 **linux** **long double**	**GNU C++ version 7.1.0** **linux** **double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Yv: Random Data	Max = 2.08e+03ε (Mean = 149ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 1.53ε (Mean = 0.102ε) (*GSL 2.1:* Max = 1.41e+06ε (Mean = 7.67e+04ε)) (*Rmath 3.2.3:* Max = 1.79e+05ε (Mean = 9.64e+03ε))	Max = 2.08e+03ε (Mean = 149ε)	Max = 1.23e+03ε (Mean = 69.9ε)

Table 154. Error rates for cyl_neumann (integer orders)

	GNU C++ version 7.1.0 **linux** **long double**	**GNU C++ version 7.1.0** **linux** **double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Y0: Mathworld Data (Integer Version)	Max = 5.53ε (Mean = 2.4ε) (*<cmath>:* Max = 2.05e+05ε (Mean = 6.87e+04ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.46ε (Mean = 2.38ε)) (*Rmath 3.2.3:* Max = 167ε (Mean = 56.5ε))	Max = 5.53ε (Mean = 2.4ε)	Max = 4.61ε (Mean = 2.29ε) (*<math.h>:* Max = 5.37e+03ε (Mean = 1.81e+03ε))
Y1: Mathworld Data (Integer Version)	Max = 6.33ε (Mean = 2.25ε) (*<cmath>:* Max = 9.71e+03ε (Mean = 4.08e+03ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.51ε (Mean = 0.839ε)) (*Rmath 3.2.3:* Max = 193ε (Mean = 64.4ε))	Max = 6.33ε (Mean = 2.29ε)	Max = 4.75ε (Mean = 1.72ε) (*<math.h>:* Max = 1.86e+04ε (Mean = 6.2e+03ε))
Yn: Mathworld Data (Integer Version)	Max = 55.2ε (Mean = 17.8ε) (*<cmath>:* Max = 2.2e+20ε (Mean = 6.97e+19ε) And other failures.)	Max = 0.993ε (Mean = 0.314ε) (*GSL 2.1:* Max = 2.41e+05ε (Mean = 7.62e+04ε)) (*Rmath 3.2.3:* Max = 1.24e+04ε (Mean = 4e+03ε))	Max = 55.2ε (Mean = 17.8ε)	Max = 35ε (Mean = 11.9ε) (*<math.h>:* Max = 2.49e+05ε (Mean = 8.14e+04ε))

Table 155. Error rates for cyl_neumann_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Y'0: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 6.33ε (Mean = 3.12ε)	Max = 6.33ε (Mean = 3.14ε)	Max = 4.75ε (Mean = 1.75ε)
Y'1: Mathworld Data	Max = 0.58ε (Mean = 0.193ε)	Max = 37.1ε (Mean = 12.8ε)	Max = 34ε (Mean = 11.8ε)	Max = 3.08ε (Mean = 1.2ε)
Y'n: Mathworld Data	Max = 2.05ε (Mean = 0.677ε)	Max = 56ε (Mean = 18.2ε)	Max = 56ε (Mean = 21.3ε)	Max = 563ε (Mean = 178ε)
Y'v: Mathworld Data	Max = 21.5ε (Mean = 6.49ε)	Max = 42.5ε (Mean = 13.4ε)	Max = 42.5ε (Mean = 13.6ε)	Max = 23.7ε (Mean = 10.1ε)
Y'v: Mathworld Data (large values)	Max = 0ε (Mean = 0ε)	Max = 1.57ε (Mean = 1.24ε)	Max = 1.57ε (Mean = 1.24ε)	Max = 0.627ε (Mean = 0.237ε)
Y'0 and Y'1: Random Data	Max = 0ε (Mean = 0ε)	Max = 23.8ε (Mean = 3.69ε)	Max = 23.8ε (Mean = 3.69ε)	Max = 5.95ε (Mean = 1.36ε)
Y'n: Random Data	Max = 1.53ε (Mean = 0.0885ε)	Max = 2.35e+03ε (Mean = 136ε)	Max = 2.35e+03ε (Mean = 136ε)	Max = 621ε (Mean = 36ε)
Y'v: Random Data	Max = 56.8ε (Mean = 2.59ε)	Max = 1.16e+05ε (Mean = 5.28e+03ε)	Max = 1.16e+05ε (Mean = 5.28e+03ε)	Max = 3.23e+04ε (Mean = 1.13e+03ε)

Table 156. Error rates for cyl_neumann_prime (integer orders)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Y'0: Mathworld Data (Integer Version)	Max = 0ε (Mean = 0ε)	Max = 6.33ε (Mean = 3.12ε)	Max = 6.33ε (Mean = 3.14ε)	Max = 4.75ε (Mean = 1.75ε)
Y'1: Mathworld Data (Integer Version)	Max = 0.58ε (Mean = 0.193ε)	Max = 37.1ε (Mean = 12.8ε)	Max = 34ε (Mean = 11.8ε)	Max = 3.08ε (Mean = 1.2ε)
Y'n: Mathworld Data (Integer Version)	Max = 2.05ε (Mean = 0.677ε)	Max = 56ε (Mean = 18.2ε)	Max = 56ε (Mean = 21.3ε)	Max = 563ε (Mean = 178ε)

Table 157. Error rates for digamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Digamma Function: Large Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.84ε (Mean = 0.71ε)) (*Rmath 3.2.3:* Max = 1.18ε (Mean = 0.331ε))	Max = 1.39ε (Mean = 0.413ε)	Max = 1.39ε (Mean = 0.413ε)	Max = 0.98ε (Mean = 0.369ε)
Digamma Function: Near the Positive Root	Max = 0.891ε (Mean = 0.0995ε) (*GSL 2.1:* Max = 135ε (Mean = 11.9ε)) (*Rmath 3.2.3:* Max = 2.02e+03ε (Mean = 256ε))	Max = 1.37ε (Mean = 0.477ε)	Max = 1.31ε (Mean = 0.471ε)	Max = 0.997ε (Mean = 0.527ε)
Digamma Function: Near Zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.953ε (Mean = 0.348ε)) (*Rmath 3.2.3:* Max = 1.17ε (Mean = 0.564ε))	Max = 0.984ε (Mean = 0.361ε)	Max = 0.984ε (Mean = 0.361ε)	Max = 0.953ε (Mean = 0.337ε)
Digamma Function: Negative Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.56e+04ε (Mean = 3.91e+03ε)) (*Rmath 3.2.3:* Max = 4.6e+04ε (Mean = 3.94e+03ε))	Max = 180ε (Mean = 13ε)	Max = 180ε (Mean = 13ε)	Max = 214ε (Mean = 16.1ε)
Digamma Function: Values near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.866ε (Mean = 0.387ε)) (*Rmath 3.2.3:* Max = 3.58e+05ε (Mean = 1.6e+05ε))	Max = 1ε (Mean = 0.592ε)	Max = 1ε (Mean = 0.592ε)	Max = 0ε (Mean = 0ε)
Digamma Function: Integer arguments	Max = 0.992ε (Mean = 0.215ε) (*GSL 2.1:* Max = 1.18ε (Mean = 0.607ε)) (*Rmath 3.2.3:* Max = 4.33ε (Mean = 0.982ε))	Max = 0.888ε (Mean = 0.403ε)	Max = 0.888ε (Mean = 0.403ε)	Max = 0.992ε (Mean = 0.452ε)
Digamma Function: Half integer arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.09ε (Mean = 0.531ε)) (*Rmath 3.2.3:* Max = 46.2ε (Mean = 7.24ε))	Max = 0.906ε (Mean = 0.409ε)	Max = 0.906ε (Mean = 0.409ε)	Max = 0.78ε (Mean = 0.314ε)

Table 158. Error rates for ellint_1

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral F: Mathworld Data	Max = 0.94ε (Mean = 0.509ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.919ε (Mean = 0.544ε))	Max = 0.94ε (Mean = 0.509ε)	Max = 0.919ε (Mean = 0.542ε)
Elliptic Integral F: Random Data	Max = 1.57ε (Mean = 0.56ε) (*<cmath>:* Max = 2.56ε (Mean = 0.816ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.99ε (Mean = 0.797ε))	Max = 1.57ε (Mean = 0.561ε)	Max = 2.26ε (Mean = 0.631ε)

Table 159. Error rates for ellint_1 (complete)

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral K: Mathworld Data	Max = 0.887ε (Mean = 0.296ε) (*<cmath>:* Max = 1.19ε (Mean = 0.765ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.623ε (Mean = 0.393ε))	Max = 0.887ε (Mean = 0.296ε)	Max = 0.915ε (Mean = 0.547ε)
Elliptic Integral K: Random Data	Max = 1.27ε (Mean = 0.473ε) (*<cmath>:* Max = 2.19ε (Mean = 0.694ε))	Max = 0.851ε (Mean = 0.0851ε) (*GSL 2.1:* Max = 2.32ε (Mean = 0.688ε))	Max = 1.27ε (Mean = 0.473ε)	Max = 0.958ε (Mean = 0.408ε)

Table 160. Error rates for ellint_2

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral E: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.63ε (Mean = 0.325ε))	Max = 0.656ε (Mean = 0.317ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0.656ε (Mean = 0.317ε)	Max = 1.31ε (Mean = 0.727ε)
Elliptic Integral E: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.4ε (Mean = 1.16ε))	Max = 2.05ε (Mean = 0.632ε) (*<cmath>:* Max = 3.08e+04ε (Mean = 3.84e+03ε))	Max = 2.05ε (Mean = 0.632ε)	Max = 2.23ε (Mean = 0.639ε)
Elliptic Integral E: Small Angles	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.5ε (Mean = 0.118ε))	Max = 1ε (Mean = 0.283ε) (*<cmath>:* Max = 2ε (Mean = 0.333ε))	Max = 1ε (Mean = 0.283ε)	Max = 1ε (Mean = 0.421ε)

Table 161. Error rates for ellint_2 (complete)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral E: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.09ε (Mean = 1.04ε) And other failures.)	Max = 0.836ε (Mean = 0.469ε) (*<cmath>:* Max = 170ε (Mean = 55.1ε))	Max = 0.836ε (Mean = 0.469ε)	Max = 1.3ε (Mean = 0.615ε)
Elliptic Integral E: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.34ε (Mean = 1.18ε))	Max = 1.97ε (Mean = 0.629ε) (*<cmath>:* Max = 2.49e+04ε (Mean = 3.39e+03ε))	Max = 1.97ε (Mean = 0.629ε)	Max = 1.71ε (Mean = 0.553ε)

Table 162. Error rates for ellint_3

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral PI: Mathworld Data	Max = 475ε (Mean = 86.3ε) (*<cmath>:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.48e+05ε (Mean = 2.54e+04ε) And other failures.)	Max = 475ε (Mean = 86.3ε)	Max = 565ε (Mean = 102ε)
Elliptic Integral PI: Random Data	Max = 4.54ε (Mean = 0.895ε) (*<cmath>:* Max = 3.37e+20ε (Mean = 3.47e+19ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 633ε (Mean = 50.1ε))	Max = 4.49ε (Mean = 0.885ε)	Max = 8.33ε (Mean = 0.971ε)
Elliptic Integral PI: Large Random Data	Max = 3.7ε (Mean = 0.893ε) (*<cmath>:* Max = 2.52e+18ε (Mean = 4.83e+17ε) And other failures.)	Max = 0.557ε (Mean = 0.0389ε) (*GSL 2.1:* Max = 40.1ε (Mean = 7.77ε))	Max = 3.7ε (Mean = 0.892ε)	Max = 2.86ε (Mean = 0.944ε)

Table 163. Error rates for ellint_3 (complete)

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Complete Elliptic Integral PI: Mathworld Data	Max = 1.4ε (Mean = 0.575ε) (*<cmath>:* Max = 6.31e+20ε (Mean = 1.53e+20ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 6.33e+04ε (Mean = 1.54e+04ε) And other failures.)	Max = 1.4ε (Mean = 0.575ε)	Max = 0.971ε (Mean = 0.464ε)
Complete Elliptic Integral PI: Random Data	Max = 2.45ε (Mean = 0.696ε) (*<cmath>:* Max = 8.78e+20ε (Mean = 1.02e+20ε) And other failures.)	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 24ε (Mean = 2.99ε))	Max = 2.4ε (Mean = 0.677ε)	Max = 2.46ε (Mean = 0.657ε)

Table 164. Error rates for ellint_d

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Elliptic Integral E: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.862ε (Mean = 0.568ε))	Max = 1.3ε (Mean = 0.813ε)	Max = 1.3ε (Mean = 0.813ε)	Max = 0.862ε (Mean = 0.457ε)
Elliptic Integral D: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.01ε (Mean = 0.928ε))	Max = 2.51ε (Mean = 0.883ε)	Max = 2.51ε (Mean = 0.883ε)	Max = 2.87ε (Mean = 0.805ε)

Table 165. Error rates for ellint_d (complete)

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Elliptic Integral E: Mathworld Data	Max = 0.637ε (Mean = 0.368ε)	Max = 1.27ε (Mean = 0.735ε)	Max = 1.27ε (Mean = 0.735ε)	Max = 0.637ε (Mean = 0.368ε)
Elliptic Integral D: Random Data	Max = 0ε (Mean = 0ε)	Max = 1.27ε (Mean = 0.334ε)	Max = 1.27ε (Mean = 0.334ε)	Max = 1.27ε (Mean = 0.355ε)

Table 166. Error rates for ellint_rc

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
RC: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.4ε (Mean = 0.624ε))	Max = 0.995ε (Mean = 0.433ε)	Max = 0.962ε (Mean = 0.407ε)

Table 167. Error rates for ellint_rd

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
RD: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.59ε (Mean = 0.878ε))	Max = 2.73ε (Mean = 0.831ε)	Max = 2.16ε (Mean = 0.803ε)
RD: y = z	Max = 0.896ε (Mean = 0.022ε) (*GSL 2.1:* Max = 2.88ε (Mean = 0.839ε))	Max = 2.65ε (Mean = 0.82ε)	Max = 16.5ε (Mean = 0.843ε)
RD: x = y	Max = 0.824ε (Mean = 0.0272ε) (*GSL 2.1:* Max = 3.74ε (Mean = 0.84ε))	Max = 2.85ε (Mean = 0.865ε)	Max = 3.51ε (Mean = 0.816ε)
RD: x = 0, y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2ε (Mean = 0.656ε))	Max = 1.19ε (Mean = 0.522ε)	Max = 1.16ε (Mean = 0.497ε)
RD: x = y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.03ε (Mean = 0.418ε))	Max = 0.998ε (Mean = 0.387ε)	Max = 1.03ε (Mean = 0.418ε)
RD: x = 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.85ε (Mean = 0.781ε))	Max = 2.79ε (Mean = 0.883ε)	Max = 2.64ε (Mean = 0.894ε)

Table 168. Error rates for ellint_rf

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RF: Random data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.73ε (Mean = 0.804ε))	Max = 2.54ε (Mean = 0.674ε)	Max = 2.02ε (Mean = 0.677ε)
RF: x = y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.999ε (Mean = 0.34ε))	Max = 0.991ε (Mean = 0.345ε)	Max = 0.999ε (Mean = 0.34ε)
RF: x = y or y = z or x = z	Max = 0.536ε (Mean = 0.00658ε) (*GSL 2.1:* Max = 2.89ε (Mean = 0.749ε))	Max = 1.95ε (Mean = 0.418ε)	Max = 1.21ε (Mean = 0.394ε)
RF: x = 0, y = z	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.29ε (Mean = 0.527ε))	Max = 0.894ε (Mean = 0.338ε)	Max = 0.999ε (Mean = 0.407ε)
RF: z = 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.54ε (Mean = 0.781ε))	Max = 1.7ε (Mean = 0.539ε)	Max = 1.89ε (Mean = 0.587ε)

Table 169. Error rates for ellint_rg

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RG: Random Data	Max = 0.983ε (Mean = 0.0172ε) (*GSL 2.1:* Max = 0.983ε (Mean = 0.0172ε))	Max = 3.95ε (Mean = 0.951ε)	Max = 3.65ε (Mean = 0.929ε)
RG: two values 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε)	Max = 0ε (Mean = 0ε)
RG: All values the same or zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0.992ε (Mean = 0.288ε)	Max = 1.06ε (Mean = 0.348ε)
RG: two values the same	Max = 0.594ε (Mean = 0.0103ε) (*GSL 2.1:* Max = 0.594ε (Mean = 0.0103ε))	Max = 1.51ε (Mean = 0.404ε)	Max = 1.96ε (Mean = 0.374ε)
RG: one value zero	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 2.14ε (Mean = 0.722ε)	Max = 1.96ε (Mean = 0.674ε)

Table 170. Error rates for ellint_rj

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Microsoft Visual C++ version 14.1 Win32 double**
RJ: Random data	Max = 0.52ε (Mean = 0.0184ε) (*GSL 2.1:* Max = 3.57ε (Mean = 0.704ε) And other failures.)	Max = 186ε (Mean = 6.67ε)	Max = 215ε (Mean = 7.66ε)
RJ: 4 Equal Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.03ε (Mean = 0.418ε))	Max = 0.998ε (Mean = 0.387ε)	Max = 1.03ε (Mean = 0.418ε)
RJ: 3 Equal Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.96ε (Mean = 1.06ε))	Max = 20.8ε (Mean = 0.986ε)	Max = 39.9ε (Mean = 1.17ε)
RJ: 2 Equal Values	Max = 0.6ε (Mean = 0.0228ε) (*GSL 2.1:* Max = 2.57ε (Mean = 0.754ε))	Max = 220ε (Mean = 6.64ε)	Max = 214ε (Mean = 5.28ε)
RJ: Equal z and p	Max = 0.742ε (Mean = 0.0166ε) (*GSL 2.1:* Max = 2.62ε (Mean = 0.699ε))	Max = 17.2ε (Mean = 1.16ε)	Max = 16.1ε (Mean = 1.14ε)

Table 171. Error rates for erf

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Erf Function: Small Values	Max = 0.925ε (Mean = 0.193ε) (*<cmath>:* Max = 0.944ε (Mean = 0.191ε)) (*<math.h>:* Max = 0.944ε (Mean = 0.191ε))	Max = 0.841ε (Mean = 0.0687ε) (*GSL 2.1:* Max = 2.06ε (Mean = 0.319ε))	Max = 0.925ε (Mean = 0.193ε) (*<math.h>:* Max = 0.944ε (Mean = 0.194ε))	Max = 0.996ε (Mean = 0.182ε) (*<math.h>:* Max = 1.57ε (Mean = 0.317ε))
Erf Function: Medium Values	Max = 1.5ε (Mean = 0.193ε) (*<cmath>:* Max = 0.921ε (Mean = 0.0723ε)) (*<math.h>:* Max = 0.921ε (Mean = 0.0723ε))	Max = 1ε (Mean = 0.119ε) (*GSL 2.1:* Max = 2.31ε (Mean = 0.368ε))	Max = 1.5ε (Mean = 0.197ε) (*<math.h>:* Max = 0.921ε (Mean = 0.071ε))	Max = 1ε (Mean = 0.171ε) (*<math.h>:* Max = 1.19ε (Mean = 0.244ε))
Erf Function: Large Values	Max = 0ε (Mean = 0ε) (*<cmath>:* Max = 0ε (Mean = 0ε)) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))

Table 172. Error rates for erf_inv

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse Erf Function	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.389ε)	Max = 1.08ε (Mean = 0.395ε)	Max = 1.09ε (Mean = 0.502ε)

Table 173. Error rates for erfc

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Erf Function: Small Values	Max = 0ε (Mean = 0ε) (*<cmath>:* Max = 0ε (Mean = 0ε)) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0.658ε (Mean = 0.0537ε) (*GSL 2.1:* Max = 1.01ε (Mean = 0.485ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 0ε (Mean = 0ε) (*<math.h>:* Max = 0ε (Mean = 0ε))
Erf Function: Medium Values	Max = 1.76ε (Mean = 0.365ε) (*<cmath>:* Max = 1.35ε (Mean = 0.307ε)) (*<math.h>:* Max = 1.35ε (Mean = 0.307ε))	Max = 0.983ε (Mean = 0.213ε) (*GSL 2.1:* Max = 2.64ε (Mean = 0.662ε))	Max = 1.76ε (Mean = 0.38ε) (*<math.h>:* Max = 2.81ε (Mean = 0.739ε))	Max = 1.65ε (Mean = 0.373ε) (*<math.h>:* Max = 2.36ε (Mean = 0.539ε))
Erf Function: Large Values	Max = 1.57ε (Mean = 0.542ε) (*<cmath>:* Max = 1.26ε (Mean = 0.441ε)) (*<math.h>:* Max = 1.26ε (Mean = 0.441ε))	Max = 0.868ε (Mean = 0.147ε) (*GSL 2.1:* Max = 3.9ε (Mean = 0.472ε))	Max = 1.57ε (Mean = 0.564ε) (*<math.h>:* Max = 4.91ε (Mean = 1.54ε))	Max = 1.14ε (Mean = 0.248ε) (*<math.h>:* Max = 1.84ε (Mean = 0.331ε))

Table 174. Error rates for erfc_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse Erfc Function	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.397ε)	Max = 1.08ε (Mean = 0.403ε)	Max = 1ε (Mean = 0.491ε)
Inverse Erfc Function: extreme values		Max = 1.62ε (Mean = 0.383ε)	Max = 1.62ε (Mean = 0.383ε)	

Table 175. Error rates for expint (Ei)

	GNU C++ version 7.1.0 **linux** **long double**	**GNU C++ version 7.1.0** **linux** **double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Exponential Integral Ei	Max = 5.05ε (Mean = 0.821ε) (*<cmath>:* Max = 14.1ε (Mean = 2.43ε) And other failures.)	Max = 0.994ε (Mean = 0.142ε) (*GSL 2.1:* Max = 8.96ε (Mean = 0.703ε))	Max = 5.05ε (Mean = 0.835ε)	Max = 1.43ε (Mean = 0.54ε)
Exponential Integral Ei: double exponent range	Max = 1.72ε (Mean = 0.593ε) (*<cmath>:* Max = 3.11ε (Mean = 1.13ε))	Max = 0.998ε (Mean = 0.156ε) (*GSL 2.1:* Max = 1.5ε (Mean = 0.612ε))	Max = 1.72ε (Mean = 0.607ε)	Max = 1.7ε (Mean = 0.66ε)
Exponential Integral Ei: long exponent range	Max = 1.98ε (Mean = 0.595ε) (*<cmath>:* Max = 1.93ε (Mean = 0.855ε))		Max = 1.98ε (Mean = 0.575ε)	

Table 176. Error rates for expint (En)

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Exponential Integral En	Max = 0.589ε (Mean = 0.0331ε) (*GSL 2.1:* Max = 58.5ε (Mean = 17.1ε))	Max = 9.97ε (Mean = 2.13ε)	Max = 9.97ε (Mean = 2.13ε)	Max = 7.16ε (Mean = 1.85ε)
Exponential Integral En: small z values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 115ε (Mean = 23.6ε))	Max = 1.99ε (Mean = 0.559ε)	Max = 1.99ε (Mean = 0.559ε)	Max = 2.62ε (Mean = 0.531ε)
Exponential Integral E1	Max = 0.556ε (Mean = 0.0625ε) (*GSL 2.1:* Max = 0.988ε (Mean = 0.469ε))	Max = 0.965ε (Mean = 0.414ε)	Max = 0.965ε (Mean = 0.408ε)	Max = 0.988ε (Mean = 0.486ε)

Table 177. Error rates for expm1

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Random test data	Max = 0.992ε (Mean = 0.402ε) (*<cmath>:* Max = 0.992ε (Mean = 0.402ε)) (*<math.h>:* Max = 0.992ε (Mean = 0.402ε))	Max = 0.793ε (Mean = 0.126ε) (*Rmath 3.2.3:* Max = 0.793ε (Mean = 0.126ε))	Max = 1.31ε (Mean = 0.428ε) (*<math.h>:* Max = 0.996ε (Mean = 0.426ε))	Max = 1.31ε (Mean = 0.496ε) (*<math.h>:* Max = 1.31ε (Mean = 0.496ε))

Table 178. Error rates for gamma_p

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma(a, z) medium values	Max = 0.955ε (Mean = 0.05ε) (*GSL 2.1:* Max = 342ε (Mean = 45.8ε)) (*Rmath 3.2.3:* Max = 389ε (Mean = 44ε))	Max = 41.6ε (Mean = 8.09ε)	Max = 239ε (Mean = 30.2ε)	Max = 35.1ε (Mean = 6.98ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.82ε (Mean = 0.758ε)) (*Rmath 3.2.3:* Max = 1.01ε (Mean = 0.306ε))	Max = 2ε (Mean = 0.464ε)	Max = 2ε (Mean = 0.461ε)	Max = 1.54ε (Mean = 0.439ε)
tgamma(a, z) large values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.02e+03ε (Mean = 105ε)) (*Rmath 3.2.3:* Max = 1.11e+03ε (Mean = 67.5ε))	Max = 3.08e+04ε (Mean = 1.86e+03ε)	Max = 3.02e+04ε (Mean = 1.91e+03ε)	Max = 243ε (Mean = 20.2ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 128ε (Mean = 22.6ε)) (*Rmath 3.2.3:* Max = 66.2ε (Mean = 12.2ε))	Max = 11.8ε (Mean = 2.66ε)	Max = 71.6ε (Mean = 9.47ε)	Max = 13ε (Mean = 2.97ε)

Table 179. Error rates for gamma_p_inv

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
incomplete gamma inverse(a, z) medium values	Max = 0.993ε (Mean = 0.15ε) (*Rmath 3.2.3:* Max = 4.88ε (Mean = 0.868ε))	Max = 1.8ε (Mean = 0.406ε)	Max = 1.89ε (Mean = 0.466ε)	Max = 1.71ε (Mean = 0.34ε)
incomplete gamma inverse(a, z) large values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 0.816ε (Mean = 0.0874ε))	Max = 0.509ε (Mean = 0.0447ε)	Max = 0.509ε (Mean = 0.0447ε)	Max = 0.924ε (Mean = 0.108ε)
incomplete gamma inverse(a, z) small values	Max = 441ε (Mean = 53.9ε) (*Rmath 3.2.3:* Max = 547ε (Mean = 61.6ε))	Max = 9.17e+03ε (Mean = 1.45e+03ε)	Max = 1.09e+04ε (Mean = 1.3e+03ε)	Max = 1.1e+03ε (Mean = 131ε)

Table 180. Error rates for gamma_p_inva

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Incomplete gamma inverses.	Max = 0ε (Mean = 0ε)	Max = 7.87ε (Mean = 1.15ε)	Max = 4.08ε (Mean = 1.12ε)	Max = 4.92ε (Mean = 1.03ε)

Table 181. Error rates for gamma_q

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma(a, z) medium values	Max = 0.927ε (Mean = 0.035ε) (*GSL 2.1:* Max = 201ε (Mean = 13.5ε)) (*Rmath 3.2.3:* Max = 131ε (Mean = 12.7ε))	Max = 32.3ε (Mean = 6.61ε)	Max = 199ε (Mean = 26.6ε)	Max = 23.7ε (Mean = 4ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.38e+10ε (Mean = 1.05e+09ε)) (*Rmath 3.2.3:* Max = 65.6ε (Mean = 11ε))	Max = 2.45ε (Mean = 0.885ε)	Max = 2.45ε (Mean = 0.819ε)	Max = 2.26ε (Mean = 0.74ε)
tgamma(a, z) large values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.71e+04ε (Mean = 2.16e+03ε)) (*Rmath 3.2.3:* Max = 1.02e+03ε (Mean = 62.7ε))	Max = 6.82e+03ε (Mean = 414ε)	Max = 1.15e+04ε (Mean = 733ε)	Max = 469ε (Mean = 31.5ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 118ε (Mean = 12.5ε)) (*Rmath 3.2.3:* Max = 138ε (Mean = 16.9ε))	Max = 11.1ε (Mean = 2.07ε)	Max = 54.7ε (Mean = 6.16ε)	Max = 8.72ε (Mean = 1.48ε)

Table 182. Error rates for gamma_q_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
incomplete gamma inverse(a, z) medium values	Max = 0.912ε (Mean = 0.154ε) (*Rmath 3.2.3:* Max = 4.66ε (Mean = 0.792ε))	Max = 6.2ε (Mean = 0.627ε)	Max = 6.2ε (Mean = 0.683ε)	Max = 2.88ε (Mean = 0.469ε)
incomplete gamma inverse(a, z) large values	Max = 0.894ε (Mean = 0.0915ε) (*Rmath 3.2.3:* Max = 0.894ε (Mean = 0.106ε))	Max = 0ε (Mean = 0ε)	Max = 0ε (Mean = 0ε)	Max = 0.814ε (Mean = 0.0856ε)
incomplete gamma inverse(a, z) small values	Max = 292ε (Mean = 36.4ε) (*Rmath 3.2.3:* Max = 415ε (Mean = 48.7ε))	Max = 8.28e+03ε (Mean = 1.09e+03ε)	Max = 8.98e+03ε (Mean = 877ε)	Max = 451ε (Mean = 64.7ε)

Table 183. Error rates for gamma_q_inva

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Incomplete gamma inverses.	Max = 0ε (Mean = 0ε)	Max = 8.42ε (Mean = 1.3ε)	Max = 7.86ε (Mean = 1.24ε)	Max = 5.05ε (Mean = 1.08ε)

Table 184. Error rates for hermite

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Hermite Polynomials	Max = 0ε (Mean = 0ε)	Max = 6.24ε (Mean = 2.07ε)	Max = 6.24ε (Mean = 2.07ε)	Max = 4.46ε (Mean = 1.41ε)

Table 185. Error rates for heuman_lambda

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Elliptic Integral Jacobi Zeta: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.89ε (Mean = 0.887ε)	Max = 1.89ε (Mean = 0.887ε)	Max = 1.08ε (Mean = 0.734ε)
Elliptic Integral Heuman Lambda: Random Data	Max = 0ε (Mean = 0ε)	Max = 3.82ε (Mean = 0.609ε)	Max = 3.82ε (Mean = 0.608ε)	Max = 2.12ε (Mean = 0.588ε)

Table 186. Error rates for ibeta

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 682ε (Mean = 32.6ε)) (*Rmath 3.2.3:* Max = 22.9ε (Mean = 3.35ε))	Max = 8.97ε (Mean = 2.09ε)	Max = 21.3ε (Mean = 2.75ε)	Max = 8.4ε (Mean = 1.93ε)
Incomplete Beta Function: Medium Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 690ε (Mean = 151ε)) (*Rmath 3.2.3:* Max = 232ε (Mean = 27.9ε))	Max = 50ε (Mean = 12.1ε)	Max = 124ε (Mean = 18.4ε)	Max = 106ε (Mean = 16.3ε)
Incomplete Beta Function: Large and Diverse Values	Max = 1.26ε (Mean = 0.063ε) (*GSL 2.1:* Max = 3.9e+05ε (Mean = 1.82e+04ε) And other failures.) (*Rmath 3.2.3:* Max = 574ε (Mean = 49.4ε))	Max = 1.96e+04ε (Mean = 997ε)	Max = 4.98e+04ε (Mean = 2.07e+03ε)	Max = 1.32e+03ε (Mean = 68.5ε)
Incomplete Beta Function: Small Integer Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 254ε (Mean = 50.9ε)) (*Rmath 3.2.3:* Max = 62.2ε (Mean = 8.95ε))	Max = 4.45ε (Mean = 0.814ε)	Max = 44.5ε (Mean = 10.1ε)	Max = 3.85ε (Mean = 0.791ε)

Table 187. Error rates for ibeta_inv

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse incomplete beta	Max = 11ε (Mean = 0.345ε) (*Rmath 3.2.3:* Max = 1.14e+121ε (Mean = 3.28e+119ε) And other failures.)	Max = 3.8e+04ε (Mean = 2.66e+03ε)	Max = 4.07e+04ε (Mean = 2.86e+03ε)	Max = 8.59e+03ε (Mean = 277ε)

Table 188. Error rates for ibeta_inva

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse incomplete beta	Max = 0.602ε (Mean = 0.0239ε)	Max = 377ε (Mean = 24.4ε)	Max = 438ε (Mean = 31.3ε)	Max = 242ε (Mean = 22.9ε)

Table 189. Error rates for ibeta_invb

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Inverse incomplete beta	Max = 0.765ε (Mean = 0.0422ε)	Max = 407ε (Mean = 27.2ε)	Max = 407ε (Mean = 24.4ε)	Max = 409ε (Mean = 19.3ε)

Table 190. Error rates for ibetac

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Incomplete Beta Function: Small Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 22.4ε (Mean = 3.67ε))	Max = 10.6ε (Mean = 2.22ε)	Max = 13.8ε (Mean = 2.68ε)	Max = 6.94ε (Mean = 1.71ε)
Incomplete Beta Function: Medium Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 204ε (Mean = 25.8ε))	Max = 73.9ε (Mean = 11.2ε)	Max = 132ε (Mean = 19.8ε)	Max = 56.7ε (Mean = 14.3ε)
Incomplete Beta Function: Large and Diverse Values	Max = 0.981ε (Mean = 0.0573ε) (*Rmath 3.2.3:* Max = 889ε (Mean = 68.4ε))	Max = 3.45e+04ε (Mean = 1.32e+03ε)	Max = 6.31e+04ε (Mean = 2.04e+03ε)	Max = 1.88e+03ε (Mean = 82.7ε)
Incomplete Beta Function: Small Integer Values	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 84.6ε (Mean = 18ε))	Max = 5.34ε (Mean = 1.11ε)	Max = 107ε (Mean = 17.1ε)	Max = 6.37ε (Mean = 1.03ε)

Table 191. Error rates for ibetac_inv

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Inverse incomplete beta	Max = 0.977ε (Mean = 0.0976ε) (*Rmath 3.2.3:* Max = 3.01e+132ε (Mean = 8.65e+130ε) And other failures.)	Max = 4.88e+04ε (Mean = 3.16e+03ε)	Max = 5.05e+04ε (Mean = 3.33e+03ε)	Max = 2.93e+03ε (Mean = 198ε)

Table 192. Error rates for ibetac_inva

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Inverse incomplete beta	Max = 0.683ε (Mean = 0.0314ε)	Max = 382ε (Mean = 22.2ε)	Max = 315ε (Mean = 23.7ε)	Max = 408ε (Mean = 26.7ε)

Table 193. Error rates for ibetac_invb

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Inverse incomplete beta	Max = 0.724ε (Mean = 0.0303ε)	Max = 317ε (Mean = 19.8ε)	Max = 369ε (Mean = 22.6ε)	Max = 271ε (Mean = 16.4ε)

Table 194. Error rates for jacobi_cn

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 17.3ε (Mean = 4.29ε) And other failures.)	Max = 71.6ε (Mean = 19.3ε)	Max = 71.6ε (Mean = 19.4ε)	Max = 45.8ε (Mean = 11.4ε)
Jacobi Elliptic: Random Data	Max = 0.816ε (Mean = 0.0563ε) (*GSL 2.1:* Max = 2.43ε (Mean = 0.803ε))	Max = 1.68ε (Mean = 0.443ε)	Max = 1.68ε (Mean = 0.454ε)	Max = 1.83ε (Mean = 0.455ε)
Jacobi Elliptic: Random Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 55.2ε (Mean = 1.64ε) And other failures.)	Max = 10.4ε (Mean = 0.594ε)	Max = 10.4ε (Mean = 0.602ε)	Max = 26.2ε (Mean = 1.17ε)
Jacobi Elliptic: Modulus near 1	Max = 0.919ε (Mean = 0.127ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 675ε (Mean = 87.1ε)	Max = 675ε (Mean = 86.8ε)	Max = 513ε (Mean = 126ε)
Jacobi Elliptic: Large Phi	Max = 14.2ε (Mean = 0.927ε) (*GSL 2.1:* Max = 5.92e+03ε (Mean = 477ε))	Max = 2.97e+04ε (Mean = 1.9e+03ε)	Max = 2.97e+04ε (Mean = 1.9e+03ε)	Max = 3.27e+04ε (Mean = 1.93e+03ε)

Table 195. Error rates for jacobi_dn

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.82ε (Mean = 1.18ε) And other failures.)	Max = 49ε (Mean = 14ε)	Max = 49ε (Mean = 14ε)	Max = 34.3ε (Mean = 8.71ε)
Jacobi Elliptic: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3ε (Mean = 0.61ε))	Max = 1.53ε (Mean = 0.473ε)	Max = 1.53ε (Mean = 0.481ε)	Max = 1.52ε (Mean = 0.466ε)
Jacobi Elliptic: Random Small Values	Max = 0.5ε (Mean = 0.0122ε) (*GSL 2.1:* Max = 1.5ε (Mean = 0.391ε) And other failures.)	Max = 22.4ε (Mean = 0.777ε)	Max = 22.4ε (Mean = 0.763ε)	Max = 16.1ε (Mean = 0.685ε)
Jacobi Elliptic: Modulus near 1	Max = 2.28ε (Mean = 0.194ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 3.75e+03ε (Mean = 293ε)	Max = 3.75e+03ε (Mean = 293ε)	Max = 6.24e+03ε (Mean = 482ε)
Jacobi Elliptic: Large Phi	Max = 14.1ε (Mean = 0.897ε) (*GSL 2.1:* Max = 121ε (Mean = 22ε))	Max = 2.82e+04ε (Mean = 1.79e+03ε)	Max = 2.82e+04ε (Mean = 1.79e+03ε)	Max = 1.67e+04ε (Mean = 1e+03ε)

Table 196. Error rates for jacobi_sn

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Jacobi Elliptic: Mathworld Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 588ε (Mean = 146ε) And other failures.)	Max = 341ε (Mean = 80.7ε)	Max = 341ε (Mean = 80.7ε)	Max = 481ε (Mean = 113ε)
Jacobi Elliptic: Random Data	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.02ε (Mean = 1.07ε))	Max = 2.01ε (Mean = 0.584ε)	Max = 2.01ε (Mean = 0.593ε)	Max = 1.92ε (Mean = 0.567ε)
Jacobi Elliptic: Random Small Values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 11.7ε (Mean = 1.65ε) And other failures.)	Max = 1.99ε (Mean = 0.347ε)	Max = 1.99ε (Mean = 0.347ε)	Max = 2.11ε (Mean = 0.385ε)
Jacobi Elliptic: Modulus near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 109ε (Mean = 7.35ε)	Max = 109ε (Mean = 7.38ε)	Max = 23.2ε (Mean = 1.85ε)
Jacobi Elliptic: Large Phi	Max = 12ε (Mean = 0.771ε) (*GSL 2.1:* Max = 4.54e+04ε (Mean = 2.63e+03ε))	Max = 2.45e+04ε (Mean = 1.51e+03ε)	Max = 2.45e+04ε (Mean = 1.51e+03ε)	Max = 4.36e+04ε (Mean = 2.54e+03ε)

Table 197. Error rates for jacobi_zeta

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Elliptic Integral Jacobi Zeta: Mathworld Data	Max = 0ε (Mean = 0ε)	Max = 1.66ε (Mean = 0.48ε)	Max = 1.66ε (Mean = 0.48ε)	Max = 1.52ε (Mean = 0.357ε)
Elliptic Integral Jacobi Zeta: Random Data	Max = 0ε (Mean = 0ε)	Max = 2.99ε (Mean = 0.824ε)	Max = 3.96ε (Mean = 1.06ε)	Max = 3.89ε (Mean = 0.824ε)
Elliptic Integral Jacobi Zeta: Large Phi Values	Max = 0ε (Mean = 0ε)	Max = 2.92ε (Mean = 0.951ε)	Max = 3.05ε (Mean = 1.13ε)	Max = 2.52ε (Mean = 0.977ε)

Table 198. Error rates for laguerre(n, m, x)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Associated Laguerre Polynomials	Max = 0.84ε (Mean = 0.0358ε) (*GSL 2.1:* Max = 434ε (Mean = 10.7ε))	Max = 167ε (Mean = 6.38ε) (*<cmath>:* Max = 206ε (Mean = 6.86ε))	Max = 167ε (Mean = 6.38ε)	Max = 434ε (Mean = 11.1ε)

Table 199. Error rates for laguerre(n, x)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Laguerre Polynomials	Max = 6.82ε (Mean = 0.408ε) (*GSL 2.1:* Max = 3.1e+03ε (Mean = 185ε))	Max = 1.39e+04ε (Mean = 828ε) (*<cmath>:* Max = 4.2e+03ε (Mean = 251ε))	Max = 1.39e+04ε (Mean = 828ε)	Max = 3.1e+03ε (Mean = 185ε)

Table 200. Error rates for legendre_p

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Legendre Polynomials: Small Values	Max = 0.732ε (Mean = 0.0619ε) (*GSL 2.1:* Max = 211ε (Mean = 20.4ε))	Max = 69.2ε (Mean = 9.58ε) (*<cmath>:* Max = 124ε (Mean = 13.2ε))	Max = 69.2ε (Mean = 9.58ε)	Max = 211ε (Mean = 20.4ε)
Legendre Polynomials: Large Values	Max = 0.632ε (Mean = 0.0693ε) (*GSL 2.1:* Max = 300ε (Mean = 33.2ε))	Max = 699ε (Mean = 59.6ε) (*<cmath>:* Max = 343ε (Mean = 32.1ε))	Max = 699ε (Mean = 59.6ε)	Max = 300ε (Mean = 33.2ε)

Table 201. Error rates for legendre_p (associated)

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Associated Legendre Polynomials: Small Values	Max = 0.999ε (Mean = 0.05ε) (*GSL 2.1:* Max = 121ε (Mean = 6.75ε) And other failures.)	Max = 175ε (Mean = 9.88ε) (*<cmath>:* Max = 175ε (Mean = 9.36ε) And other failures.)	Max = 77.7ε (Mean = 5.59ε)	Max = 121ε (Mean = 7.14ε)

Table 202. Error rates for legendre_q

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Legendre Polynomials: Small Values	Max = 0.612ε (Mean = 0.0517ε) (*GSL 2.1:* Max = 46.4ε (Mean = 7.46ε))	Max = 50.9ε (Mean = 9ε)	Max = 50.9ε (Mean = 8.98ε)	Max = 46.4ε (Mean = 7.32ε)
Legendre Polynomials: Large Values	Max = 2.49ε (Mean = 0.202ε) (*GSL 2.1:* Max = 4.6e+03ε (Mean = 366ε))	Max = 5.98e+03ε (Mean = 478ε)	Max = 5.98e+03ε (Mean = 478ε)	Max = 4.6e+03ε (Mean = 366ε)

Table 203. Error rates for lgamma

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
factorials	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 33.6ε (Mean = 2.78ε)) (*Rmath 3.2.3:* Max = 1.55ε (Mean = 0.592ε))	Max = 0.991ε (Mean = 0.308ε) (*<cmath>:* Max = 1.67ε (Mean = 0.487ε)) (*<math.h>:* Max = 1.67ε (Mean = 0.487ε))	Max = 0.991ε (Mean = 0.383ε) (*<math.h>:* Max = 1.36ε (Mean = 0.476ε))	Max = 0.914ε (Mean = 0.175ε) (*<math.h>:* Max = 0.958ε (Mean = 0.38ε))
near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 5.21ε (Mean = 1.57ε)) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε))	Max = 1.42ε (Mean = 0.566ε) (*<cmath>:* Max = 0.964ε (Mean = 0.543ε)) (*<math.h>:* Max = 0.964ε (Mean = 0.543ε))	Max = 1.42ε (Mean = 0.566ε) (*<math.h>:* Max = 0.964ε (Mean = 0.543ε))	Max = 0.964ε (Mean = 0.462ε) (*<math.h>:* Max = 0.962ε (Mean = 0.372ε))
near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 442ε (Mean = 88.8ε)) (*Rmath 3.2.3:* Max = 7.99e+04ε (Mean = 1.68e+04ε))	Max = 0.948ε (Mean = 0.36ε) (*<cmath>:* Max = 0.615ε (Mean = 0.096ε)) (*<math.h>:* Max = 0.615ε (Mean = 0.096ε))	Max = 0.948ε (Mean = 0.36ε) (*<math.h>:* Max = 1.71ε (Mean = 0.581ε))	Max = 0.867ε (Mean = 0.468ε) (*<math.h>:* Max = 0.906ε (Mean = 0.565ε))
near 2	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.17e+03ε (Mean = 274ε)) (*Rmath 3.2.3:* Max = 2.63e+05ε (Mean = 5.84e+04ε))	Max = 0.878ε (Mean = 0.242ε) (*<cmath>:* Max = 0.741ε (Mean = 0.263ε)) (*<math.h>:* Max = 0.741ε (Mean = 0.263ε))	Max = 0.878ε (Mean = 0.242ε) (*<math.h>:* Max = 0.598ε (Mean = 0.235ε))	Max = 0.591ε (Mean = 0.159ε) (*<math.h>:* Max = 0.741ε (Mean = 0.473ε))
near -10	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 24.9ε (Mean = 4.6ε)) (*Rmath 3.2.3:* Max = 4.22ε (Mean = 1.26ε))	Max = 3.81ε (Mean = 1.01ε) (*<cmath>:* Max = 0.997ε (Mean = 0.412ε)) (*<math.h>:* Max = 0.997ε (Mean = 0.412ε))	Max = 3.81ε (Mean = 1.01ε) (*<math.h>:* Max = 3.04ε (Mean = 1.01ε))	Max = 4.22ε (Mean = 1.33ε) (*<math.h>:* Max = 0.997ε (Mean = 0.444ε))
near -55	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.02ε (Mean = 1.47ε)) (*Rmath 3.2.3:* Max = 250ε (Mean = 60.9ε))	Max = 0.821ε (Mean = 0.513ε) (*<cmath>:* Max = 1.58ε (Mean = 0.672ε)) (*<math.h>:* Max = 1.58ε (Mean = 0.672ε))	Max = 1.59ε (Mean = 0.587ε) (*<math.h>:* Max = 0.821ε (Mean = 0.674ε))	Max = 0.821ε (Mean = 0.419ε) (*<math.h>:* Max = 249ε (Mean = 43.1ε))

Table 204. Error rates for log1p

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Random test data	Max = 0.818ε (Mean = 0.227ε) (*<cmath>:* Max = 0.818ε (Mean = 0.227ε)) (*<math.h>:* Max = 0.818ε (Mean = 0.227ε))	Max = 0.846ε (Mean = 0.153ε) (*Rmath 3.2.3:* Max = 0.846ε (Mean = 0.153ε))	Max = 2.3ε (Mean = 0.66ε) (*<math.h>:* Max = 0.818ε (Mean = 0.249ε))	Max = 0.509ε (Mean = 0.057ε) (*<math.h>:* Max = 0.509ε (Mean = 0.057ε))

Table 205. Error rates for non central beta CDF

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Beta, medium parameters	Max = 0.998ε (Mean = 0.0649ε) (*Rmath 3.2.3:* Max = 1.46e+26ε (Mean = 3.5e+24ε) And other failures.)	Max = 824ε (Mean = 27.4ε)	Max = 832ε (Mean = 38.1ε)	Max = 242ε (Mean = 31ε)
Non Central Beta, large parameters	Max = 1.18ε (Mean = 0.175ε) (*Rmath 3.2.3:* Max = 1.01e+36ε (Mean = 1.19e+35ε) And other failures.)	Max = 2.5e+04ε (Mean = 3.78e+03ε)	Max = 2.57e+04ε (Mean = 4.45e+03ε)	Max = 3.66e+03ε (Mean = 500ε)

Table 206. Error rates for non central beta CDF complement

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Beta, medium parameters	Max = 0.998ε (Mean = 0.0936ε) (*Rmath 3.2.3:* Max = 7.5e+97ε (Mean = 1.37e+96ε) And other failures.)	Max = 396ε (Mean = 50.7ε)	Max = 554ε (Mean = 57.2ε)	Max = 624ε (Mean = 62.7ε)
Non Central Beta, large parameters	Max = 0.986ε (Mean = 0.188ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 6.83e+03ε (Mean = 993ε)	Max = 3.56e+03ε (Mean = 707ε)	Max = 1.25e+04ε (Mean = 1.49e+03ε)

Table 207. Error rates for non central chi squared CDF

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Chi Squared, medium parameters	Max = 0.99ε (Mean = 0.0544ε) (*Rmath 3.2.3:* Max = 727ε (Mean = 121ε))	Max = 46.5ε (Mean = 10.3ε)	Max = 115ε (Mean = 13.9ε)	Max = 48.9ε (Mean = 10ε)
Non Central Chi Squared, large parameters	Max = 1.07ε (Mean = 0.102ε) (*Rmath 3.2.3:* Max = 3.27e+08ε (Mean = 2.23e+07ε))	Max = 3.07e+03ε (Mean = 336ε)	Max = 6.17e+03ε (Mean = 677ε)	Max = 9.79e+03ε (Mean = 723ε)

Table 208. Error rates for non central chi squared CDF complement

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Non Central Chi Squared, medium parameters	Max = 0.96ε (Mean = 0.0635ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 107ε (Mean = 17.2ε)	Max = 171ε (Mean = 22.8ε)	Max = 98.6ε (Mean = 15.8ε)
Non Central Chi Squared, large parameters	Max = 2.11ε (Mean = 0.278ε) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 5.02e+03ε (Mean = 630ε)	Max = 5.1e+03ε (Mean = 577ε)	Max = 5.43e+03ε (Mean = 705ε)

Table 209. Error rates for non central t CDF

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Non Central T	Max = 0.796ε (Mean = 0.0691ε) (*Rmath 3.2.3:* Max = 5.28e+15ε (Mean = 8.49e+14ε) And other failures.)	Max = 139ε (Mean = 31ε)	Max = 145ε (Mean = 30.9ε)	Max = 135ε (Mean = 32.1ε)
Non Central T (small non-centrality)	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 2.09e+03ε (Mean = 244ε))	Max = 3.86ε (Mean = 1.4ε)	Max = 9.15ε (Mean = 2.13ε)	Max = 6.17ε (Mean = 1.45ε)
Non Central T (large parameters)	Max = 257ε (Mean = 72.1ε) (*Rmath 3.2.3:* Max = 2.46ε (Mean = 0.657ε))	Max = 5.26e+05ε (Mean = 1.48e+05ε)	Max = 5.24e+05ε (Mean = 1.47e+05ε)	Max = 286ε (Mean = 62.8ε)

Table 210. Error rates for non central t CDF complement

	GNU C++ version 7.1.0 **linux** **double**	**GNU C++ version 7.1.0** **linux** **long double**	**Sun compiler version 0x5150** **Sun Solaris** **long double**	**Microsoft Visual C++ version 14.1** **Win32** **double**
Non Central T	Max = 0.707ε (Mean = 0.0497ε) (*Rmath 3.2.3:* Max = 6.19e+15ε (Mean = 6.72e+14ε) And other failures.)	Max = 201ε (Mean = 31.7ε)	Max = 340ε (Mean = 43.2ε)	Max = 154ε (Mean = 32.1ε)
Non Central T (small non-centrality)	Max = 0ε (Mean = 0ε) (*Rmath 3.2.3:* Max = 1.87e+03ε (Mean = 263ε))	Max = 10.5ε (Mean = 2.13ε)	Max = 10.5ε (Mean = 2.39ε)	Max = 4.6ε (Mean = 1.63ε)
Non Central T (large parameters)	Max = 478ε (Mean = 96.3ε) (*Rmath 3.2.3:* Max = 2.24ε (Mean = 0.945ε))	Max = 9.79e+05ε (Mean = 1.97e+05ε)	Max = 9.79e+05ε (Mean = 1.97e+05ε)	Max = 227ε (Mean = 50.4ε)

Table 211. Error rates for owens_t

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Owens T (medium small values)	Max = 0ε (Mean = 0ε)	Max = 3.34ε (Mean = 0.944ε)	Max = 3.34ε (Mean = 0.911ε)	Max = 4.37ε (Mean = 0.98ε)
Owens T (large and diverse values)	Max = 0ε (Mean = 0ε)	Max = 49ε (Mean = 2.16ε)	Max = 24.5ε (Mean = 1.39ε)	Max = 3.78ε (Mean = 0.621ε)

Table 212. Error rates for polygamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Mathematica Data	Max = 0.824ε (Mean = 0.0574ε) (*GSL 2.1:* Max = 62.9ε (Mean = 12.8ε)) (*Rmath 3.2.3:* Max = 108ε (Mean = 15.2ε))	Max = 7.38ε (Mean = 1.84ε)	Max = 34.3ε (Mean = 7.65ε)	Max = 9.32ε (Mean = 1.95ε)
Mathematica Data - large arguments	Max = 0.998ε (Mean = 0.0592ε) (*GSL 2.1:* Max = 244ε (Mean = 32.8ε) And other failures.) (*Rmath 3.2.3:* Max = 1.71e+56ε (Mean = 1.01e+55ε) And other failures.)	Max = 2.23ε (Mean = 0.323ε)	Max = 11.1ε (Mean = 0.848ε)	Max = 150ε (Mean = 13.9ε)
Mathematica Data - negative arguments	Max = 0.516ε (Mean = 0.022ε) (*GSL 2.1:* Max = 36.6ε (Mean = 3.04ε) And other failures.) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 269ε (Mean = 87.7ε)	Max = 269ε (Mean = 88.4ε)	Max = 497ε (Mean = 129ε)
Mathematica Data - large negative arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.79ε (Mean = 0.197ε) And other failures.) (*Rmath 3.2.3:* Max = 0ε (Mean = 0ε) And other failures.)	Max = 155ε (Mean = 96.4ε)	Max = 155ε (Mean = 96.4ε)	Max = 162ε (Mean = 101ε)
Mathematica Data - small arguments	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 15.2ε (Mean = 5.03ε)) (*Rmath 3.2.3:* Max = 106ε (Mean = 20ε))	Max = 3.33ε (Mean = 0.75ε)	Max = 3.33ε (Mean = 0.75ε)	Max = 3ε (Mean = 0.496ε)
Mathematica Data - Large orders and other bug cases	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 151ε (Mean = 39.3ε) And other failures.) (*Rmath 3.2.3:* Max = +INFε (Mean = +INFε) And other failures.)	Max = 54.5ε (Mean = 13.3ε)	Max = 145ε (Mean = 55.9ε)	Max = 200ε (Mean = 57.2ε)

Table 213. Error rates for powm1

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
powm1	Max = 1.06ε (Mean = 0.425ε)	Max = 2.04ε (Mean = 0.493ε)	Max = 1.88ε (Mean = 0.49ε)	Max = 1.84ε (Mean = 0.486ε)

Table 214. Error rates for sin_pi

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
sin_pi and cos_pi	Max = 0ε (Mean = 0ε)	Max = 0.996ε (Mean = 0.335ε)	Max = 0.996ε (Mean = 0.336ε)	Max = 0.99ε (Mean = 0.328ε)
sin_pi and cos_pi near integers and half integers	Max = 0ε (Mean = 0ε)	Max = 0.976ε (Mean = 0.293ε)	Max = 0.976ε (Mean = 0.293ε)	Max = 0.996ε (Mean = 0.343ε)

Table 215. Error rates for sph_bessel

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel j: Random Data	Max = 243ε (Mean = 13.3ε) (*<cmath>:* Max = 1.91e+06ε (Mean = 1.09e+05ε))	Max = 0.978ε (Mean = 0.0445ε) (*GSL 2.1:* Max = 1.79e+03ε (Mean = 107ε))	Max = 243ε (Mean = 33.7ε)	Max = 245ε (Mean = 16.3ε)

Table 216. Error rates for sph_bessel_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Bessel j': Random Data	Max = 0.753ε (Mean = 0.0343ε)	Max = 167ε (Mean = 12ε)	Max = 167ε (Mean = 33.2ε)	Max = 307ε (Mean = 25.2ε)

Table 217. Error rates for sph_neumann

	GNU C++ version 7.1.0 linux long double	GNU C++ version 7.1.0 linux double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
y: Random Data	Max = 234ε (Mean = 19.5ε) (*<cmath>:* Max = 1.6e+06ε (Mean = 1.4e+05ε))	Max = 0.995ε (Mean = 0.0665ε) (*GSL 2.1:* Max = 8.5e+04ε (Mean = 5.33e+03ε))	Max = 234ε (Mean = 19.8ε)	Max = 281ε (Mean = 31.1ε)

Table 218. Error rates for sph_neumann_prime

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
y': Random Data	Max = 0.988ε (Mean = 0.0869ε)	Max = 158ε (Mean = 18.8ε)	Max = 158ε (Mean = 20.2ε)	Max = 296ε (Mean = 25.6ε)

Table 219. Error rates for spherical_harmonic_i

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Spherical Harmonics	Max = 1.36ε (Mean = 0.0765ε)	Max = 2.89e+03ε (Mean = 108ε)	Max = 1.03e+04ε (Mean = 327ε)	Max = 2.27e+04ε (Mean = 725ε)

Table 220. Error rates for spherical_harmonic_r

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
Spherical Harmonics	Max = 1.58ε (Mean = 0.0707ε)	Max = 2.89e+03ε (Mean = 108ε)	Max = 1.03e+04ε (Mean = 327ε)	Max = 2.27e+04ε (Mean = 725ε)

Table 221. Error rates for sqrt1pm1

	GNU C++ version 7.1.0 linux double	GNU C++ version 7.1.0 linux long double	Sun compiler version 0x5150 Sun Solaris long double	Microsoft Visual C++ version 14.1 Win32 double
sqrt1pm1	Max = 1.3ε (Mean = 0.404ε)	Max = 1.33ε (Mean = 0.404ε)	Max = 1.54ε (Mean = 0.563ε)	Max = 1.35ε (Mean = 0.497ε)

Table 222. Error rates for tgamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
factorials	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.95ε (Mean = 0.783ε)) (*Rmath 3.2.3:* Max = 314ε (Mean = 93.4ε))	Max = 2.67ε (Mean = 0.617ε) (*<cmath>:* Max = 1.66ε (Mean = 0.584ε)) (*<math.h>:* Max = 1.66ε (Mean = 0.584ε))	Max = 172ε (Mean = 41ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 1.85ε (Mean = 0.566ε) (*<math.h>:* Max = 3.17ε (Mean = 0.928ε))
near 0	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.51ε (Mean = 1.92ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.335ε))	Max = 2ε (Mean = 0.608ε) (*<cmath>:* Max = 1ε (Mean = 0.376ε)) (*<math.h>:* Max = 1ε (Mean = 0.376ε))	Max = 2ε (Mean = 0.647ε) (*<math.h>:* Max = 0.5ε (Mean = 0.0791ε))	Max = 1.5ε (Mean = 0.635ε) (*<math.h>:* Max = 1ε (Mean = 0.405ε))
near 1	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 4.41ε (Mean = 1.81ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.32ε))	Max = 2.51ε (Mean = 1.02ε) (*<cmath>:* Max = 0.918ε (Mean = 0.203ε)) (*<math.h>:* Max = 0.918ε (Mean = 0.203ε))	Max = 3.01ε (Mean = 1.06ε) (*<math.h>:* Max = 1ε (Mean = 0.175ε))	Max = 1.1ε (Mean = 0.59ε) (*<math.h>:* Max = 1ε (Mean = 0.4ε))
near 2	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.95ε (Mean = 3.12ε)) (*Rmath 3.2.3:* Max = 1ε (Mean = 0.191ε))	Max = 4.1ε (Mean = 1.55ε) (*<cmath>:* Max = 0.558ε (Mean = 0.298ε)) (*<math.h>:* Max = 0.558ε (Mean = 0.298ε))	Max = 5.01ε (Mean = 1.89ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 2ε (Mean = 0.733ε) (*<math.h>:* Max = 0ε (Mean = 0ε))
near -10	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 2.6ε (Mean = 1.05ε)) (*Rmath 3.2.3:* Max = 34.9ε (Mean = 9.2ε))	Max = 1.75ε (Mean = 0.895ε) (*<cmath>:* Max = 2.26ε (Mean = 1.08ε)) (*<math.h>:* Max = 2.26ε (Mean = 1.08ε))	Max = 1.75ε (Mean = 0.819ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 1.86ε (Mean = 0.881ε) (*<math.h>:* Max = 0.866ε (Mean = 0.445ε))
near -55	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 1.8ε (Mean = 0.782ε)) (*Rmath 3.2.3:* Max = 3.89e+04ε (Mean = 9.52e+03ε))	Max = 2.69ε (Mean = 1.09ε) (*<cmath>:* Max = 1.79ε (Mean = 0.75ε)) (*<math.h>:* Max = 1.79ε (Mean = 0.75ε))	Max = 98.5ε (Mean = 53.4ε) (*<math.h>:* Max = 0ε (Mean = 0ε))	Max = 2.7ε (Mean = 1.35ε) (*<math.h>:* Max = 3.87e+04ε (Mean = 6.71e+03ε))

Table 223. Error rates for tgamma1pm1

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma1pm1(dz)	Max = 0ε (Mean = 0ε)	Max = 1.12ε (Mean = 0.49ε)	Max = 6.61ε (Mean = 0.84ε)	Max = 3.31ε (Mean = 0.517ε)

Table 224. Error rates for tgamma_delta_ratio

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma + small delta ratios	Max = 0ε (Mean = 0ε)	Max = 5.83ε (Mean = 1.3ε)	Max = 15.4ε (Mean = 2.09ε)	Max = 7.56ε (Mean = 1.31ε)
tgamma + small delta ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 7.94ε (Mean = 1.4ε)	Max = 18.3ε (Mean = 2.03ε)	Max = 7.43ε (Mean = 1.42ε)
tgamma + small integer ratios	Max = 0ε (Mean = 0ε)	Max = 1.96ε (Mean = 0.677ε)	Max = 1.96ε (Mean = 0.677ε)	Max = 2.74ε (Mean = 0.736ε)
tgamma + small integer ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 1.62ε (Mean = 0.451ε)	Max = 1.62ε (Mean = 0.451ε)	Max = 2.15ε (Mean = 0.685ε)
integer tgamma ratios	Max = 0ε (Mean = 0ε)	Max = 0.997ε (Mean = 0.4ε)	Max = 0.997ε (Mean = 0.4ε)	Max = 0.968ε (Mean = 0.386ε)
integer tgamma ratios (negative delta)	Max = 0ε (Mean = 0ε)	Max = 0.853ε (Mean = 0.176ε)	Max = 0.853ε (Mean = 0.176ε)	Max = 0.974ε (Mean = 0.175ε)

Table 225. Error rates for tgamma (incomplete)

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma(a, z) medium values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 200ε (Mean = 13.3ε))	Max = 8.47ε (Mean = 1.9ε)	Max = 412ε (Mean = 95.5ε)	Max = 8.14ε (Mean = 1.76ε)
tgamma(a, z) small values	Max = 0.753ε (Mean = 0.0474ε) (*GSL 2.1:* Max = 1.38e+10ε (Mean = 1.05e+09ε))	Max = 2.31ε (Mean = 0.775ε)	Max = 2.13ε (Mean = 0.717ε)	Max = 2.53ε (Mean = 0.66ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 117ε (Mean = 12.5ε))	Max = 5.52ε (Mean = 1.48ε)	Max = 79.6ε (Mean = 20.9ε)	Max = 5.16ε (Mean = 1.33ε)

Table 226. Error rates for tgamma_lower

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma(a, z) medium values	Max = 0.833ε (Mean = 0.0315ε) (*GSL 2.1:* Max = 0.833ε (Mean = 0.0315ε))	Max = 6.79ε (Mean = 1.46ε)	Max = 363ε (Mean = 63.8ε)	Max = 5.62ε (Mean = 1.49ε)
tgamma(a, z) small values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 1.97ε (Mean = 0.555ε)	Max = 1.97ε (Mean = 0.558ε)	Max = 1.57ε (Mean = 0.525ε)
tgamma(a, z) integer and half integer values	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0ε (Mean = 0ε))	Max = 4.83ε (Mean = 1.15ε)	Max = 84.7ε (Mean = 17.5ε)	Max = 2.69ε (Mean = 0.849ε)

Table 227. Error rates for tgamma_ratio

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
tgamma ratios	Max = 0.694ε (Mean = 0.0347ε)	Max = 2.99ε (Mean = 1.15ε)	Max = 174ε (Mean = 61.2ε)	Max = 3.28ε (Mean = 1.12ε)

Table 228. Error rates for trigamma

	GNU C++ version 7.1.0 linux double	**GNU C++ version 7.1.0 linux long double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Mathematica Data	Max = 0.998ε (Mean = 0.105ε) (*GSL 2.1:* Max = 1.34e+04ε (Mean = 1.49e+03ε)) (*Rmath 3.2.3:* Max = 1.34e+04ε (Mean = 1.51e+03ε))	Max = 1.28ε (Mean = 0.449ε)	Max = 1.28ε (Mean = 0.449ε)	Max = 1ε (Mean = 0.382ε)

Table 229. Error rates for zeta

	GNU C++ version 7.1.0 linux long double	**GNU C++ version 7.1.0 linux double**	**Sun compiler version 0x5150 Sun Solaris long double**	**Microsoft Visual C++ version 14.1 Win32 double**
Zeta: Random values greater than 1	Max = 0.846ε (Mean = 0.0833ε) (*<cmath>:* Max = 5.45ε (Mean = 1ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 8.69ε (Mean = 1.03ε))	Max = 0.846ε (Mean = 0.0833ε)	Max = 0.836ε (Mean = 0.093ε)
Zeta: Random values less than 1	Max = 7.03ε (Mean = 2.93ε) (*<cmath>:* Max = 538ε (Mean = 59.3ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 137ε (Mean = 13.8ε))	Max = 70.1ε (Mean = 17.1ε)	Max = 6.84ε (Mean = 3.12ε)
Zeta: Values close to and greater than 1	Max = 0.995ε (Mean = 0.5ε) (*<cmath>:* Max = 1.9e+06ε (Mean = 5.11e+05ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 7.73ε (Mean = 4.07ε))	Max = 0.995ε (Mean = 0.5ε)	Max = 0.994ε (Mean = 0.421ε)
Zeta: Values close to and less than 1	Max = 0.998ε (Mean = 0.508ε) (*<cmath>:* Max = 8.53e+06ε (Mean = 1.87e+06ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 0.991ε (Mean = 0.28ε))	Max = 0.998ε (Mean = 0.508ε)	Max = 0.991ε (Mean = 0.375ε)
Zeta: Integer arguments	Max = 9ε (Mean = 3.06ε) (*<cmath>:* Max = 70.3ε (Mean = 17.4ε))	Max = 0ε (Mean = 0ε) (*GSL 2.1:* Max = 3.75ε (Mean = 1.1ε))	Max = 28ε (Mean = 5.62ε)	Max = 9ε (Mean = 3ε)

Error Logs For Error Rate Tables

Error Output For cyl_bessel_j (integer orders) with compiler Microsoft Visual C++ version 14.1 and library <math.h> and test data Bessel JN: Mathworld Data (Integer Version)

```
CAUTION: Found non-finite result, when a finite value was expected at entry 16
Found: -nan(ind) Expected 0 Error: 1.79769e+308
10, 1e-100, 0
CAUTION: Gross error found at entry 16.
Found: -nan(ind) Expected 0 Error: 1.79769e+308
10, 1e-100, 0
```

Error Output For legendre_p (associated) with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Associated Legendre Polynomials: Small Values

```
domain error
3.75573, -3, 0.264719, 0.0186823
domain error
3.75573, -3, 0.670017, 0.0085227
domain error
3.75573, -3, 0.915014, 0.00136786
domain error
```

```
3.75573, -3, 0.93539, 0.000921218
domain error
3.75573, -2, -0.804919, -0.035427
domain error
3.75573, -2, -0.623236, -0.0476446
domain error
3.75573, -2, 0.629447, 0.0475072
domain error
3.75573, -2, 0.929777, 0.0157498
domain error
3.75573, -2, 0.985763, 0.0034837
domain error
3.75573, -1, 0.093763, -0.118979
domain error
4.28576, -4, 0.0944412, 0.00255792
domain error
4.28576, -4, 0.670017, 0.000790849
domain error
4.28576, -3, -0.746026, -0.00458957
domain error
4.28576, -2, -0.623236, 0.0219016
domain error
4.28576, -2, 0.629447, 0.0223081
domain error
4.28576, -2, 0.93539, 0.0133504
domain error
4.28576, -1, 0.915014, 0.132001
domain error
4.28576, -1, 0.985763, 0.0787743
domain error
4.43859, -4, 0.093763, 0.00255858
domain error
4.43859, -4, 0.811584, 0.000303404
domain error
4.43859, -4, 0.826752, 0.000260835
domain error
4.43859, -4, 0.929777, 4.78235e-05
domain error
4.43859, -3, -0.804919, -0.00350364
domain error
4.43859, -3, -0.729046, -0.00487043
domain error
4.43859, -3, -0.623236, -0.00620995
domain error
4.43859, -3, 0.93539, 0.000861698
domain error
4.43859, -2, -0.557932, 0.0169167
domain error
4.43859, -2, -0.443004, 0.0062586
domain error
4.43859, -2, 0.915014, 0.016481
domain error
4.43859, -1, 0.629447, -0.0138523
domain error
5.39088, -5, 0.0944412, 0.000254649
domain error
5.39088, -5, 0.264719, 0.000217164
domain error
```

```
5.39088, -5, 0.670017, 5.87083e-05
domain error
5.39088, -5, 0.915014, 2.78273e-06
domain error
5.39088, -3, 0.929777, 0.000880849
domain error
5.39088, -2, 0.629447, 0.00448021
domain error
5.39088, -2, 0.826752, 0.01718
domain error
5.39088, -2, 0.937736, 0.011583
domain error
5.39088, -1, -0.804919, 0.0276144
domain error
5.39088, -1, -0.746026, -0.0119425
domain error
5.39088, -1, -0.443004, -0.0525987
domain error
5.39088, -1, 0.811584, 0.032475
domain error
5.39088, -1, 0.985763, 0.0759289
domain error
5.97861, -5, -0.729046, 3.91223e-05
domain error
5.97861, -5, -0.383666, 0.000174899
domain error
5.97861, -5, 0.93539, 1.43993e-06
domain error
5.97861, -4, -0.623236, -0.000607048
domain error
5.97861, -4, 0.264719, 0.00059614
domain error
5.97861, -3, 0.629447, 0.00313497
domain error
5.97861, -3, 0.670017, 0.00323895
domain error
5.97861, -2, 0.915014, 0.0140705
domain error
5.97861, -2, 0.992923, 0.00171356
domain error
5.97861, -1, -0.746026, -0.0119425
domain error
5.97861, -1, 0.937736, 0.106972
domain error
7.01297, -6, -0.443004, -4.99177e-06
domain error
7.01297, -6, 0.629447, 3.00689e-06
domain error
7.01297, -6, 0.811584, 7.00407e-07
domain error
7.01297, -6, 0.985763, 4.83431e-10
domain error
7.01297, -3, 0.670017, 0.000233323
domain error
7.01297, -2, -0.804919, -0.0027739
domain error
7.01297, -1, -0.383666, 0.0397866
domain error
```

```
7.01297, -1, 0.929777, 0.0544549
domain error
7.54701, -7, 0.929777, 1.42008e-09
domain error
7.54701, -6, 0.992923, 6.04622e-11
domain error
7.54701, -5, -0.804919, 1.18502e-05
domain error
7.54701, -5, -0.623236, 2.57049e-05
domain error
7.54701, -5, -0.557932, 2.60266e-05
domain error
7.54701, -5, 0.826752, 9.64276e-06
domain error
7.54701, -4, -0.746026, -0.0001618
domain error
7.54701, -3, 0.0944412, 0.000622493
domain error
7.54701, -3, 0.985763, 9.14782e-05
domain error
7.54701, -1, 0.811584, -0.0376184
domain error
11.8439, -10, -0.557932, -2.32652e-11
domain error
11.8439, -10, 0.811584, 1.01194e-12
domain error
11.8439, -8, -0.746026, -1.34891e-09
domain error
11.8439, -8, -0.729046, -1.5428e-09
domain error
11.8439, -8, 0.985763, 5.90035e-14
domain error
11.8439, -4, 0.629447, -1.44723e-05
domain error
11.8439, -4, 0.929777, 1.98812e-05
domain error
11.8439, -3, 0.670017, -4.58296e-05
domain error
11.8439, -2, 0.826752, -0.00244759
domain error
11.8439, -2, 0.992923, 0.00151458
domain error
11.8439, -1, -0.383666, 0.00419108
domain error
11.85, -11, 0.093763, 1.16526e-11
domain error
11.85, -11, 0.929777, 2.05797e-16
domain error
11.85, -11, 0.93539, 1.32249e-16
domain error
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For legendre_p (associated) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Associated Legendre Polynomials: Small Values

```
order parameters less than 0 not supported in TR1
3.75573, -3, 0.264719, 0.0186823
order parameters less than 0 not supported in TR1
```

```
3.75573, -3, 0.670017, 0.0085227
order parameters less than 0 not supported in TR1
3.75573, -3, 0.915014, 0.00136786
order parameters less than 0 not supported in TR1
3.75573, -3, 0.93539, 0.000921218
order parameters less than 0 not supported in TR1
3.75573, -2, -0.804919, -0.035427
order parameters less than 0 not supported in TR1
3.75573, -2, -0.623236, -0.0476446
order parameters less than 0 not supported in TR1
3.75573, -2, 0.629447, 0.0475072
order parameters less than 0 not supported in TR1
3.75573, -2, 0.929777, 0.0157498
order parameters less than 0 not supported in TR1
3.75573, -2, 0.985763, 0.0034837
order parameters less than 0 not supported in TR1
3.75573, -1, 0.093763, -0.118979
order parameters less than 0 not supported in TR1
4.28576, -4, 0.0944412, 0.00255792
order parameters less than 0 not supported in TR1
4.28576, -4, 0.670017, 0.000790849
order parameters less than 0 not supported in TR1
4.28576, -3, -0.746026, -0.00458957
order parameters less than 0 not supported in TR1
4.28576, -2, -0.623236, 0.0219016
order parameters less than 0 not supported in TR1
4.28576, -2, 0.629447, 0.0223081
order parameters less than 0 not supported in TR1
4.28576, -2, 0.93539, 0.0133504
order parameters less than 0 not supported in TR1
4.28576, -1, 0.915014, 0.132001
order parameters less than 0 not supported in TR1
4.28576, -1, 0.985763, 0.0787743
order parameters less than 0 not supported in TR1
4.43859, -4, 0.093763, 0.00255858
order parameters less than 0 not supported in TR1
4.43859, -4, 0.811584, 0.000303404
order parameters less than 0 not supported in TR1
4.43859, -4, 0.826752, 0.000260835
order parameters less than 0 not supported in TR1
4.43859, -4, 0.929777, 4.78235e-05
order parameters less than 0 not supported in TR1
4.43859, -3, -0.804919, -0.00350364
order parameters less than 0 not supported in TR1
4.43859, -3, -0.729046, -0.00487043
order parameters less than 0 not supported in TR1
4.43859, -3, -0.623236, -0.00620995
order parameters less than 0 not supported in TR1
4.43859, -3, 0.93539, 0.000861698
order parameters less than 0 not supported in TR1
4.43859, -2, -0.557932, 0.0169167
order parameters less than 0 not supported in TR1
4.43859, -2, -0.443004, 0.0062586
order parameters less than 0 not supported in TR1
4.43859, -2, 0.915014, 0.016481
order parameters less than 0 not supported in TR1
4.43859, -1, 0.629447, -0.0138523
order parameters less than 0 not supported in TR1
```

```
5.39088, -5, 0.0944412, 0.000254649
order parameters less than 0 not supported in TR1
5.39088, -5, 0.264719, 0.000217164
order parameters less than 0 not supported in TR1
5.39088, -5, 0.670017, 5.87083e-05
order parameters less than 0 not supported in TR1
5.39088, -5, 0.915014, 2.78273e-06
order parameters less than 0 not supported in TR1
5.39088, -3, 0.929777, 0.000880849
order parameters less than 0 not supported in TR1
5.39088, -2, 0.629447, 0.00448021
order parameters less than 0 not supported in TR1
5.39088, -2, 0.826752, 0.01718
order parameters less than 0 not supported in TR1
5.39088, -2, 0.937736, 0.011583
order parameters less than 0 not supported in TR1
5.39088, -1, -0.804919, 0.0276144
order parameters less than 0 not supported in TR1
5.39088, -1, -0.746026, -0.0119425
order parameters less than 0 not supported in TR1
5.39088, -1, -0.443004, -0.0525987
order parameters less than 0 not supported in TR1
5.39088, -1, 0.811584, 0.032475
order parameters less than 0 not supported in TR1
5.39088, -1, 0.985763, 0.0759289
order parameters less than 0 not supported in TR1
5.97861, -5, -0.729046, 3.91223e-05
order parameters less than 0 not supported in TR1
5.97861, -5, -0.383666, 0.000174899
order parameters less than 0 not supported in TR1
5.97861, -5, 0.93539, 1.43993e-06
order parameters less than 0 not supported in TR1
5.97861, -4, -0.623236, -0.000607048
order parameters less than 0 not supported in TR1
5.97861, -4, 0.264719, 0.00059614
order parameters less than 0 not supported in TR1
5.97861, -3, 0.629447, 0.00313497
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel Iv: Mathworld Data (large values)

```
CAUTION: Gross error found at entry 0.
Found: 0 Expected 1.86459e-155 Error: 8.37988e+152
-1, 3.72917e-155, 1.86459e-155
CAUTION: Gross error found at entry 1.
Found: 0 Expected 1.86459e-155 Error: 8.37988e+152
1, 3.72917e-155, 1.86459e-155
CAUTION: Gross error found at entry 3.
Found: 0 Expected 8.02269e-175 Error: 3.60559e+133
1.125, 3.72917e-155, 8.02269e-175
```

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel In: Mathworld Data (Integer Version)

```
Unsupported domain
-5, -1, -0.000271463
Unsupported domain
```

10, -5, 0.00458004
Unsupported domain
-100, -200, 4.35275e+74

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel I1: Mathworld Data (Integer Version)

Unsupported domain
1, -2, -1.59064
Unsupported domain
1, -8, -399.873
Unsupported domain
1, -10, -2670.99

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel I0: Mathworld Data (Integer Version)

Unsupported domain
0, -2, 2.27959
Unsupported domain
0, -7, 168.594
Unsupported domain
0, -1, 1.26607

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel In: Mathworld Data

Unsupported domain
-5, -1, -0.000271463
Unsupported domain
10, -5, 0.00458004
Unsupported domain
-100, -200, 4.35275e+74

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel I1: Mathworld Data

Unsupported domain
1, -2, -1.59064
Unsupported domain
1, -8, -399.873
Unsupported domain
1, -10, -2670.99

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel I0: Mathworld Data

Unsupported domain
0, -2, 2.27959
Unsupported domain
0, -7, 168.594
Unsupported domain
0, -1, 1.26607

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel J: Mathworld Data

```
CAUTION: Gross error found at entry 6.
Found: 0 Expected -0.000747424 Error: 3.3591e+304
5.5, 1e+06, -0.000747424
CAUTION: Gross error found at entry 7.
Found: 0 Expected -0.0007766 Error: 3.49022e+304
5.125, 1e+06, -0.0007766
CAUTION: Gross error found at entry 8.
Found: 0 Expected -0.000466323 Error: 2.09576e+304
5.875, 1e+06, -0.000466323
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel JN: Mathworld Data (Integer Version)

```
Unsupported domain
5, -10, 0.234062
CAUTION: Gross error found at entry 6.
Found: 0 Expected 0.000725964 Error: 3.26265e+304
-5, 1e+06, 0.000725964
CAUTION: Gross error found at entry 7.
Found: 0 Expected -0.000725964 Error: 3.26265e+304
5, 1e+06, -0.000725964
Unsupported domain
-5, -1, 0.000249758
Unsupported domain
10, -10, 0.207486
Unsupported domain
10, -5, 0.0014678
CAUTION: Gross error found at entry 12.
Found: 0 Expected -0.000331079 Error: 1.48795e+304
-10, 1e+06, -0.000331079
CAUTION: Gross error found at entry 13.
Found: 0 Expected -0.000331079 Error: 1.48795e+304
10, 1e+06, -0.000331079
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel J1: Mathworld Data (Integer Version)

```
Unsupported domain
1, -2, -0.576725
Unsupported domain
1, -8, -0.234636
Unsupported domain
1, -10, -0.0434727
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel J0: Mathworld Data (Integer Version)

```
Unsupported domain
0, -2, 0.223891
Unsupported domain
0, -8, 0.171651
Unsupported domain
0, -10, -0.245936
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel JN: Mathworld Data

```
Unsupported domain
5, -10, 0.234062
CAUTION: Gross error found at entry 6.
Found: 0 Expected 0.000725964 Error: 3.26265e+304
-5, 1e+06, 0.000725964
CAUTION: Gross error found at entry 7.
Found: 0 Expected -0.000725964 Error: 3.26265e+304
5, 1e+06, -0.000725964
Unsupported domain
-5, -1, 0.000249758
Unsupported domain
10, -10, 0.207486
Unsupported domain
10, -5, 0.0014678
CAUTION: Gross error found at entry 12.
Found: 0 Expected -0.000331079 Error: 1.48795e+304
-10, 1e+06, -0.000331079
CAUTION: Gross error found at entry 13.
Found: 0 Expected -0.000331079 Error: 1.48795e+304
10, 1e+06, -0.000331079
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel J1: Mathworld Data

```
Unsupported domain
1, -2, -0.576725
Unsupported domain
1, -8, -0.234636
Unsupported domain
1, -10, -0.0434727
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Bessel J0: Mathworld Data

```
Unsupported domain
0, -2, 0.223891
Unsupported domain
0, -8, 0.171651
Unsupported domain
0, -10, -0.245936
```

Error Output For ibetac_inv with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Inverse incomplete beta

```
CAUTION: Gross error found at entry 7.
Found: 3.8247e-302 Expected 0 Error: 1.71891e+06
1.38853e-05, 0.0497627, 0.632396, 0, 0
CAUTION: Gross error found at entry 71.
Found: 1.38362e-204 Expected 0 Error: 6.21832e+103
3.77931e-05, 0.0150073, 0.835025, 0, 0
CAUTION: Gross error found at entry 90.
Found: 1.09275e-303 Expected 0 Error: 49109.6
4.29383e-05, 0.0428761, 0.814742, 0, 0
CAUTION: Gross error found at entry 102.
```

```
Found: 3.8625e-304 Expected 0 Error: 17358
4.80089e-05, 0.0296236, 0.913384, 0, 0
CAUTION: Gross error found at entry 115.
Found: 1.51774e-303 Expected 0 Error: 68209.8
0.000130387, 0.0404969, 0.814742, 0, 0
CAUTION: Gross error found at entry 123.
Found: 1.28036e-303 Expected 0 Error: 57541.4
0.000149328, 0.0201182, 0.905801, 5.70765e-267, 0
CAUTION: Gross error found at entry 133.
Found: 1.96732e-302 Expected 0 Error: 884160
0.000173563, 0.0301908, 0.913384, 4.21662e-213, 0
CAUTION: Gross error found at entry 159.
Found: 1.48697e-191 Expected 0 Error: 6.68279e+116
0.000260723, 0.0252933, 0.632396, 0, 0
CAUTION: Gross error found at entry 256.
Found: 9.24166e-245 Expected 0 Error: 4.15342e+63
0.00246975, 0.016063, 0.913384, 1, 0
```

Error Output For ibeta_inv with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Inverse incomplete beta

```
CAUTION: Gross error found at entry 1.
Found: 1.90197e-247 Expected 0 Error: 8.54789e+60
1.12733e-05, 0.022662, 0.135563, 0, 0
CAUTION: Gross error found at entry 30.
Found: 1.36217e-301 Expected 0 Error: 6.12191e+06
2.10769e-05, 0.0448972, 0.221112, 0, 0
CAUTION: Gross error found at entry 152.
Found: 2.92621e-285 Expected 0 Error: 1.31511e+23
0.000240381, 0.017982, 0.221112, 0, 0
CAUTION: Gross error found at entry 184.
Found: 5.63355e-203 Expected 0 Error: 2.53185e+105
0.000348822, 0.0275467, 0.135563, 0, 1.88165e-166
CAUTION: Gross error found at entry 205.
Found: 5.52731e-303 Expected 0 Error: 248409
0.000441212, 0.0313573, 0.127074, 0, 9.07221e-121
```

Error Output For non central beta CDF complement with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Beta, large parameters

```
CAUTION: Gross error found at entry 10.
Found: 9.76918e-10 Expected 1.61248e-15 Error: 605846
290.682, 72.6705, 20005.4, 0.997663, 1, 1.61248e-15
CAUTION: Gross error found at entry 11.
Found: 9.94184e-10 Expected 3.0108e-42 Error: 3.30205e+32
290.682, 145.341, 53489.1, 0.998663, 1, 3.0108e-42
CAUTION: Gross error found at entry 16.
Found: 8.45406e-10 Expected 4.46652e-22 Error: 1.89276e+12
290.682, 1162.73, 2308.07, 0.656921, 1, 4.46652e-22
CAUTION: Gross error found at entry 17.
Found: 9.41971e-10 Expected 1.7241e-50 Error: 5.46356e+40
290.682, 1453.41, 8064.48, 0.832237, 1, 1.7241e-50
CAUTION: Gross error found at entry 18.
Found: 9.30663e-10 Expected 2.09803e-305 Error: 4.43589e+295
975.766, 731.824, 232.285, 0.919742, 1, 2.09803e-305
CAUTION: Gross error found at entry 27.
Found: 9.76918e-10 Expected 9.3474e-18 Error: 1.04512e+08
1879.05, 187.905, 20005.4, 0.992215, 1, 9.3474e-18
```

```
CAUTION: Gross error found at entry 28.
Found: 9.94184e-10 Expected 1.8122e-90 Error: 5.48607e+80
1879.05, 469.762, 53489.1, 0.994618, 1, 1.8122e-90
CAUTION: Gross error found at entry 32.
Found: 9.27224e-10 Expected 3.18255e-15 Error: 291345
1879.05, 3758.1, 1879.05, 0.480508, 1, 3.18255e-15
CAUTION: Gross error found at entry 33.
Found: 8.45406e-10 Expected 1.10218e-77 Error: 7.67029e+67
1879.05, 5637.15, 2308.07, 0.458181, 1, 1.10218e-77
CAUTION: Gross error found at entry 34.
Found: 9.30663e-10 Expected 0 Error: 4.18262e+298
2308.07, 1154.03, 232.285, 0.919371, 1, 0
CAUTION: Gross error found at entry 35.
Found: 8.93617e-10 Expected 0 Error: 4.01612e+298
2308.07, 1731.05, 290.682, 0.917262, 1, 0
CAUTION: Gross error found at entry 43.
Found: 9.94184e-10 Expected 3.57283e-70 Error: 2.78262e+60
8064.48, 806.448, 53489.1, 0.988678, 1, 3.57283e-70
CAUTION: Gross error found at entry 48.
Found: 8.45406e-10 Expected 8.78057e-74 Error: 9.62814e+63
8064.48, 16129, 2308.07, 0.421531, 1, 8.78057e-74
CAUTION: Gross error found at entry 49.
Found: 9.30663e-10 Expected 0 Error: 4.18262e+298
15674.4, 3918.59, 232.285, 0.933726, 1, 0
CAUTION: Gross error found at entry 50.
Found: 8.93617e-10 Expected 0 Error: 4.01612e+298
15674.4, 7837.19, 290.682, 0.917179, 1, 0
CAUTION: Gross error found at entry 51.
Found: 8.9318e-10 Expected 0 Error: 4.01416e+298
15674.4, 11755.8, 975.766, 0.915784, 1, 0
CAUTION: Gross error found at entry 63.
Found: 9.41971e-10 Expected 2.31296e-171 Error: 4.07258e+161
20005.4, 40010.8, 8064.48, 0.432094, 1, 2.31296e-171
CAUTION: Gross error found at entry 64.
Found: 9.30663e-10 Expected 0 Error: 4.18262e+298
53489.1, 5348.92, 232.285, 0.954635, 1, 0
CAUTION: Gross error found at entry 65.
Found: 8.93617e-10 Expected 0 Error: 4.01612e+298
53489.1, 13372.3, 290.682, 0.933478, 1, 0
CAUTION: Gross error found at entry 66.
Found: 8.9318e-10 Expected 0 Error: 4.01416e+298
53489.1, 26744.6, 975.766, 0.91717, 1, 0
```

Error Output For non central beta CDF with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Beta, large parameters

```
CAUTION: Gross error found at entry 0.
Found: 9.1136e-209 Expected 5.82279e-200 Error: 6.38913e+08
232.285, 209.056, 232.285, 0.062486, 5.82279e-200, 1
CAUTION: Gross error found at entry 1.
Found: 4.08108e-115 Expected 2.37643e-112 Error: 581.304
232.285, 229.962, 290.682, 0.155342, 2.37643e-112, 1
CAUTION: Gross error found at entry 2.
Found: 1.07549e-93 Expected 9.53431e-89 Error: 88650
232.285, 232.052, 975.766, 0.378086, 9.53431e-89, 1
CAUTION: Gross error found at entry 3.
Found: 2.58402e-54 Expected 8.27353e-53 Error: 31.0181
232.285, 232.285, 1879.05, 0.625865, 8.27353e-53, 1
```

```
CAUTION: Gross error found at entry 4.
Found: 1.93718e-19 Expected 6.64275e-16 Error: 3428.08
232.285, 232.308, 2308.07, 0.770774, 6.64275e-16, 1
CAUTION: Gross error found at entry 21.
Found: 8.12962e-240 Expected 1.82294e-219 Error: 2.24234e+20
975.766, 974.79, 1879.05, 0.331337, 1.82294e-219, 1
CAUTION: Gross error found at entry 22.
Found: 3.47274e-69 Expected 1.42183e-67 Error: 39.9426
975.766, 975.766, 2308.07, 0.514323, 1.42183e-67, 1
CAUTION: Gross error found at entry 23.
Found: 5.86885e-50 Expected 1.27896e-47 Error: 216.923
975.766, 975.863, 8064.48, 0.753209, 1.27896e-47, 1
CAUTION: Gross error found at entry 39.
Found: 4.82785e-230 Expected 1.25446e-213 Error: 2.59838e+16
2308.07, 2308.07, 8064.48, 0.54983, 1.25446e-213, 1
CAUTION: Gross error found at entry 40.
Found: 1.22971e-87 Expected 1.82618e-85 Error: 147.505
2308.07, 2308.3, 15674.4, 0.733174, 1.82618e-85, 1
CAUTION: Gross error found at entry 56.
Found: 2.97337e-127 Expected 2.56068e-124 Error: 860.205
15674.4, 15675.9, 20005.4, 0.55883, 2.56068e-124, 1
```

Error Output For non central beta CDF complement with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Beta, medium parameters

```
CAUTION: Gross error found at entry 296.
Found: 9.44166e-10 Expected 6.22975e-10 Error: 0.515577
22.9367, 114.683, 19.5081, 0.480981, 1, 6.22975e-10
CAUTION: Gross error found at entry 369.
Found: 2.52234e-10 Expected 1.40246e-10 Error: 0.79851
27.5277, 20.6457, 0.956697, 0.915111, 1, 1.40246e-10
CAUTION: Gross error found at entry 429.
Found: 1.18105e-09 Expected 7.45745e-10 Error: 0.58372
28.8063, 21.6047, 60.3826, 0.946143, 1, 7.45745e-10
CAUTION: Gross error found at entry 430.
Found: 2.44435e-09 Expected 1.60695e-09 Error: 0.521115
28.8063, 21.6047, 148.129, 0.965121, 1, 1.60695e-09
CAUTION: Gross error found at entry 477.
Found: 7.57435e-10 Expected 7.14133e-11 Error: 9.60635
28.8063, 144.032, 42.3849, 0.504845, 1, 7.14133e-11
CAUTION: Gross error found at entry 489.
Found: 4.8561e-10 Expected 5.62991e-11 Error: 7.62553
31.9438, 23.9579, 44.2068, 0.93835, 1, 5.62991e-11
CAUTION: Gross error found at entry 490.
Found: 8.35187e-10 Expected 1.87483e-10 Error: 3.45473
31.9438, 23.9579, 135.747, 0.961117, 1, 1.87483e-10
CAUTION: Gross error found at entry 491.
Found: 1.00174e-09 Expected 2.38491e-10 Error: 3.20032
31.9438, 23.9579, 191.501, 0.968273, 1, 2.38491e-10
CAUTION: Gross error found at entry 537.
Found: 7.29746e-10 Expected 1.31223e-12 Error: 555.111
31.9438, 159.719, 34.2373, 0.489796, 1, 1.31223e-12
CAUTION: Gross error found at entry 538.
Found: 2.49663e-09 Expected 1.54239e-09 Error: 0.618681
31.9438, 159.719, 126.472, 0.581861, 1, 1.54239e-09
CAUTION: Gross error found at entry 549.
Found: 4.16125e-10 Expected 4.8536e-13 Error: 856.353
38.0822, 28.5617, 34.773, 0.931853, 1, 4.8536e-13
```

```
CAUTION: Gross error found at entry 550.
Found: 9.69907e-10 Expected 2.87054e-12 Error: 336.883
38.0822, 28.5617, 127.953, 0.956104, 1, 2.87054e-12
CAUTION: Gross error found at entry 551.
Found: 5.90132e-10 Expected 4.08361e-12 Error: 143.512
38.0822, 28.5617, 183.147, 0.963764, 1, 4.08361e-12
CAUTION: Gross error found at entry 597.
Found: 4.67033e-10 Expected 9.82939e-16 Error: 475139
38.0822, 190.411, 27.0954, 0.475419, 1, 9.82939e-16
CAUTION: Gross error found at entry 598.
Found: 9.33207e-10 Expected 4.03465e-12 Error: 230.298
38.0822, 190.411, 100.733, 0.544491, 1, 4.03465e-12
CAUTION: Gross error found at entry 599.
Found: 7.4092e-10 Expected 9.53942e-11 Error: 6.76693
38.0822, 190.411, 169.826, 0.594614, 1, 9.53942e-11
CAUTION: Gross error found at entry 609.
Found: 5.71813e-10 Expected 1.17207e-14 Error: 48785.7
42.7789, 32.0842, 28.3773, 0.927814, 1, 1.17207e-14
CAUTION: Gross error found at entry 610.
Found: 5.16834e-10 Expected 9.62679e-14 Error: 5367.71
42.7789, 32.0842, 109.376, 0.950307, 1, 9.62679e-14
CAUTION: Gross error found at entry 611.
Found: 6.08012e-10 Expected 1.7454e-13 Error: 3482.51
42.7789, 32.0842, 175.686, 0.960431, 1, 1.7454e-13
CAUTION: Gross error found at entry 657.
Found: 5.59489e-10 Expected 2.86344e-18 Error: 1.95391e+08
42.7789, 213.895, 21.9724, 0.467166, 1, 2.86344e-18
CAUTION: Gross error found at entry 658.
Found: 5.14798e-10 Expected 2.50972e-14 Error: 20511.2
42.7789, 213.895, 84.4175, 0.522676, 1, 2.50972e-14
CAUTION: Gross error found at entry 659.
Found: 8.49991e-10 Expected 2.38005e-12 Error: 356.131
42.7789, 213.895, 160.056, 0.576191, 1, 2.38005e-12
CAUTION: Gross error found at entry 671.
Found: 3.03281e-10 Expected 2.22036e-15 Error: 136590
44.5963, 33.4472, 22.4929, 0.924976, 1, 2.22036e-15
CAUTION: Gross error found at entry 672.
Found: 8.40636e-10 Expected 2.22384e-14 Error: 37800.1
44.5963, 33.4472, 94.9517, 0.946545, 1, 2.22384e-14
CAUTION: Gross error found at entry 673.
Found: 8.15021e-10 Expected 4.75974e-14 Error: 17122.2
44.5963, 33.4472, 162.945, 0.95793, 1, 4.75974e-14
CAUTION: Gross error found at entry 716.
Found: 1.11988e-10 Expected 2.84965e-22 Error: 3.92989e+11
44.5963, 222.981, 0.956697, 0.445432, 1, 2.84965e-22
CAUTION: Gross error found at entry 717.
Found: 7.99524e-10 Expected 3.04552e-15 Error: 262523
44.5963, 222.981, 78.4454, 0.515267, 1, 3.04552e-15
CAUTION: Gross error found at entry 718.
Found: 8.0958e-10 Expected 5.89458e-13 Error: 1372.43
44.5963, 222.981, 158.441, 0.57107, 1, 5.89458e-13
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For non central beta CDF with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Beta, medium parameters

```
CAUTION: Gross error found at entry 14.
Found: 4.64669e-35 Expected 7.14875e-33 Error: 152.846
```

```
1.45431, 1.30887, 158.441, 0.0983847, 7.14875e-33, 1
CAUTION: Gross error found at entry 15.
Found: 4.66674e-46 Expected 3.13332e-40 Error: 671416
1.45431, 1.30887, 196.222, 0.09869, 3.13332e-40, 1
CAUTION: Gross error found at entry 18.
Found: 5.84342e-28 Expected 3.61559e-27 Error: 5.18745
1.45431, 1.43976, 159.586, 0.245596, 3.61559e-27, 1
CAUTION: Gross error found at entry 19.
Found: 1.72833e-34 Expected 1.76943e-33 Error: 9.2378
1.45431, 1.43976, 198.576, 0.246444, 1.76943e-33, 1
CAUTION: Gross error found at entry 22.
Found: 1.76915e-19 Expected 3.69506e-18 Error: 19.8861
1.45431, 1.45285, 159.621, 0.491116, 3.69506e-18, 1
CAUTION: Gross error found at entry 23.
Found: 2.52007e-25 Expected 2.00482e-22 Error: 794.544
1.45431, 1.45285, 199.292, 0.492849, 2.00482e-22, 1
CAUTION: Gross error found at entry 73.
Found: 2.04477e-34 Expected 2.45287e-33 Error: 10.9958
7.62448, 6.86203, 148.129, 0.0921776, 2.45287e-33, 1
CAUTION: Gross error found at entry 74.
Found: 2.36587e-46 Expected 7.32638e-42 Error: 30966
7.62448, 6.86203, 193.539, 0.093784, 7.32638e-42, 1
CAUTION: Gross error found at entry 76.
Found: 3.29122e-26 Expected 7.418e-25 Error: 21.5387
7.62448, 7.54824, 148.626, 0.228717, 7.418e-25, 1
CAUTION: Gross error found at entry 77.
Found: 1.70126e-32 Expected 1.07666e-31 Error: 5.32864
7.62448, 7.54824, 193.774, 0.23303, 1.07666e-31, 1
CAUTION: Gross error found at entry 79.
Found: 1.3478e-15 Expected 4.21836e-15 Error: 2.12982
7.62448, 7.61686, 151.548, 0.457773, 4.21836e-15, 1
CAUTION: Gross error found at entry 80.
Found: 8.78487e-21 Expected 3.41238e-19 Error: 37.8438
7.62448, 7.61686, 194.119, 0.465826, 3.41238e-19, 1
CAUTION: Gross error found at entry 132.
Found: 3.85783e-23 Expected 1.54142e-22 Error: 2.99555
19.9593, 17.9634, 44.2068, 0.0698905, 1.54142e-22, 1
CAUTION: Gross error found at entry 133.
Found: 8.6122e-39 Expected 3.94361e-38 Error: 3.5791
19.9593, 17.9634, 135.747, 0.0829178, 3.94361e-38, 1
CAUTION: Gross error found at entry 134.
Found: 3.61781e-52 Expected 3.98669e-48 Error: 11018.6
19.9593, 17.9634, 191.501, 0.0864897, 3.98669e-48, 1
CAUTION: Gross error found at entry 135.
Found: 2.07122e-15 Expected 7.08614e-15 Error: 2.42124
19.9593, 19.7597, 55.6996, 0.176444, 7.08614e-15, 1
CAUTION: Gross error found at entry 136.
Found: 2.28223e-27 Expected 2.16759e-25 Error: 93.977
19.9593, 19.7597, 136.272, 0.20393, 2.16759e-25, 1
CAUTION: Gross error found at entry 137.
Found: 6.4251e-34 Expected 4.0064e-33 Error: 5.23554
19.9593, 19.7597, 191.898, 0.213398, 4.0064e-33, 1
CAUTION: Gross error found at entry 139.
Found: 2.1734e-14 Expected 4.65637e-14 Error: 1.14243
19.9593, 19.9394, 145.168, 0.410858, 4.65637e-14, 1
CAUTION: Gross error found at entry 140.
Found: 2.18388e-19 Expected 5.1677e-18 Error: 22.663
19.9593, 19.9394, 192.978, 0.426523, 5.1677e-18, 1
```

```
CAUTION: Gross error found at entry 192.
Found: 3.29537e-23 Expected 8.29996e-23 Error: 1.51867
22.4174, 20.1757, 34.773, 0.0661999, 8.29996e-23, 1
CAUTION: Gross error found at entry 193.
Found: 7.86091e-39 Expected 2.77686e-38 Error: 2.5325
22.4174, 20.1757, 127.953, 0.0809614, 2.77686e-38, 1
CAUTION: Gross error found at entry 194.
Found: 3.0161e-51 Expected 4.5396e-48 Error: 1504.12
22.4174, 20.1757, 183.147, 0.0848857, 4.5396e-48, 1
CAUTION: Gross error found at entry 195.
Found: 3.08022e-14 Expected 1.42713e-13 Error: 3.6332
22.4174, 22.1932, 37.6764, 0.162145, 1.42713e-13, 1
CAUTION: Gross error found at entry 196.
Found: 8.89935e-28 Expected 2.56187e-25 Error: 286.871
22.4174, 22.1932, 131.096, 0.199361, 2.56187e-25, 1
CAUTION: Gross error found at entry 197.
Found: 9.34392e-34 Expected 6.14831e-33 Error: 5.58001
22.4174, 22.1932, 186.799, 0.209601, 6.14831e-33, 1
CAUTION: Gross error found at entry 199.
Found: 2.79341e-13 Expected 4.79277e-13 Error: 0.71574
22.4174, 22.395, 131.148, 0.398015, 4.79277e-13, 1
CAUTION: Gross error found at entry 200.
Found: 3.13989e-19 Expected 7.01608e-18 Error: 21.345
22.4174, 22.395, 191.433, 0.419933, 7.01608e-18, 1
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For non central chi squared CDF complement with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Chi Squared, large parameters

```
CAUTION: Gross error found at entry 12.
Found: 0 Expected 1.17655e-12 Error: 5.28771e+295
101.815, 5236.73, 6406.25, 1, 1.17655e-12
CAUTION: Gross error found at entry 13.
Found: 0 Expected 1.79374e-44 Error: 8.06149e+263
101.815, 9735.22, 12788.2, 1, 1.79374e-44
CAUTION: Gross error found at entry 35.
Found: 2.58682e-14 Expected 1.84404e-61 Error: 1.4028e+47
107.623, 122.456, 920.317, 1, 1.84404e-61
CAUTION: Gross error found at entry 36.
Found: 0 Expected 2.30757e-102 Error: 1.03707e+206
107.623, 156.292, 1319.58, 1, 2.30757e-102
CAUTION: Gross error found at entry 52.
Found: 0 Expected 6.40952e-24 Error: 2.88059e+284
114.68, 417.884, 1065.13, 1, 6.40952e-24
CAUTION: Gross error found at entry 53.
Found: 0 Expected 1.02366e-98 Error: 4.60058e+209
114.68, 669.781, 2353.38, 1, 1.02366e-98
CAUTION: Gross error found at entry 69.
Found: 0 Expected 6.55726e-39 Error: 2.94699e+269
118.032, 3168.71, 4930.11, 1, 6.55726e-39
CAUTION: Gross error found at entry 85.
Found: 0 Expected 7.30688e-22 Error: 3.28388e+286
163.004, 9735.22, 11877.9, 1, 7.30688e-22
CAUTION: Gross error found at entry 86.
Found: 0 Expected 1.17171e-111 Error: 5.26596e+196
163.004, 25344.1, 33159.2, 1, 1.17171e-111
CAUTION: Gross error found at entry 108.
Found: 1.12355e-13 Expected 2.67349e-61 Error: 4.20255e+47
```

```
256.292, 122.456, 1136.25, 1, 2.67349e-61
CAUTION: Gross error found at entry 109.
Found: 1.16462e-13 Expected 8.30595e-116 Error: 1.40216e+102
256.292, 156.292, 1650.34, 1, 8.30595e-116
CAUTION: Gross error found at entry 124.
Found: 1.05804e-13 Expected 1.01672e-15 Error: 103.064
517.884, 417.884, 1403.65, 1, 1.01672e-15
CAUTION: Gross error found at entry 125.
Found: 2.00728e-13 Expected 3.50192e-56 Error: 5.73194e+42
517.884, 669.781, 2375.33, 1, 3.50192e-56
CAUTION: Gross error found at entry 141.
Found: 0 Expected 1.36924e-20 Error: 6.15368e+287
769.781, 3168.71, 5120.04, 1, 1.36924e-20
CAUTION: Gross error found at entry 142.
Found: 0 Expected 3.19215e-72 Error: 1.43463e+236
769.781, 5236.73, 9009.76, 1, 3.19215e-72
CAUTION: Gross error found at entry 157.
Found: 0 Expected 7.26231e-08 Error: 3.26385e+300
1223.88, 9735.22, 12055, 1, 7.26231e-08
CAUTION: Gross error found at entry 158.
Found: 0 Expected 4.5906e-56 Error: 2.06312e+252
1223.88, 25344.1, 31881.6, 1, 4.5906e-56
CAUTION: Gross error found at entry 194.
Found: 0 Expected 5.34714e-12 Error: 2.40313e+296
9835.22, 122.456, 10953.4, 1, 5.34714e-12
CAUTION: Gross error found at entry 195.
Found: 0 Expected 4.84412e-40 Error: 2.17706e+268
9835.22, 156.292, 11989.8, 1, 4.84412e-40
CAUTION: Gross error found at entry 196.
Found: 0 Expected 5.50199e-83 Error: 2.47272e+225
9835.22, 417.884, 13329, 1, 5.50199e-83
CAUTION: Gross error found at entry 197.
Found: 0 Expected 1.28192e-205 Error: 5.76124e+102
9835.22, 669.781, 15757.5, 1, 1.28192e-205
CAUTION: Gross error found at entry 211.
Found: 0 Expected 3.83272e-28 Error: 1.72251e+280
25444.1, 1123.88, 29224.8, 1, 3.83272e-28
CAUTION: Gross error found at entry 212.
Found: 0 Expected 1.69815e-101 Error: 7.63188e+206
25444.1, 3168.71, 34335.4, 1, 1.69815e-101
CAUTION: Gross error found at entry 213.
Found: 0 Expected 1.09245e-217 Error: 4.90974e+90
25444.1, 5236.73, 39885.1, 1, 1.09245e-217
```

Error Output For non central chi squared CDF complement with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central Chi Squared, medium parameters

```
CAUTION: Gross error found at entry 36.
Found: 1.11022e-14 Expected 1.30043e-26 Error: 8.53738e+11
1.95191, 109.376, 445.313, 1, 1.30043e-26
CAUTION: Gross error found at entry 37.
Found: 0 Expected 1.45478e-39 Error: 6.53812e+268
1.95191, 109.444, 556.98, 1, 1.45478e-39
CAUTION: Gross error found at entry 54.
Found: 2.91989e-14 Expected 4.25949e-21 Error: 6.85501e+06
1.95191, 159.586, 484.613, 1, 4.25949e-21
CAUTION: Gross error found at entry 55.
Found: 0 Expected 1.33424e-37 Error: 5.99639e+270
```

```
1.95191, 159.621, 646.292, 1, 1.33424e-37
CAUTION: Gross error found at entry 56.
Found: 1.25455e-14 Expected 1.95903e-56 Error: 6.40393e+41
1.95191, 160.056, 810.04, 1, 1.95903e-56
CAUTION: Gross error found at entry 73.
Found: 0 Expected 4.34735e-25 Error: 1.9538e+283
1.95191, 193.539, 586.473, 1, 4.34735e-25
CAUTION: Gross error found at entry 74.
Found: 0 Expected 4.66119e-45 Error: 2.09485e+263
1.95191, 193.774, 782.902, 1, 4.66119e-45
CAUTION: Gross error found at entry 75.
Found: 4.77396e-15 Expected 8.92248e-68 Error: 5.35048e+52
1.95191, 194.119, 980.352, 1, 8.92248e-68
CAUTION: Gross error found at entry 111.
Found: 0 Expected 3.1064e-15 Error: 1.39609e+293
20.4105, 84.4175, 314.484, 1, 3.1064e-15
CAUTION: Gross error found at entry 112.
Found: 0 Expected 7.50903e-29 Error: 3.37473e+279
20.4105, 94.9517, 461.449, 1, 7.50903e-29
CAUTION: Gross error found at entry 113.
Found: 3.77476e-15 Expected 1.74225e-43 Error: 2.1666e+28
20.4105, 97.0751, 587.428, 1, 1.74225e-43
CAUTION: Gross error found at entry 130.
Found: 8.88178e-16 Expected 4.13277e-23 Error: 2.14911e+07
20.4105, 151.548, 515.876, 1, 4.13277e-23
CAUTION: Gross error found at entry 131.
Found: 1.75415e-14 Expected 1.92146e-41 Error: 9.12928e+26
20.4105, 152.75, 692.642, 1, 1.92146e-41
CAUTION: Gross error found at entry 132.
Found: 1.38778e-14 Expected 7.09864e-64 Error: 1.95499e+49
20.4105, 158.441, 894.26, 1, 7.09864e-64
CAUTION: Gross error found at entry 149.
Found: 2.22045e-16 Expected 8.74501e-28 Error: 2.5391e+11
20.4105, 191.433, 635.532, 1, 8.74501e-28
CAUTION: Gross error found at entry 150.
Found: 0 Expected 6.94227e-50 Error: 3.12002e+258
20.4105, 191.501, 847.648, 1, 6.94227e-50
CAUTION: Gross error found at entry 151.
Found: 3.40838e-14 Expected 5.3889e-75 Error: 6.32482e+60
20.4105, 191.898, 1061.55, 1, 5.3889e-75
CAUTION: Gross error found at entry 206.
Found: 5.88418e-15 Expected 2.69136e-22 Error: 2.18632e+07
22.8625, 141.209, 492.215, 1, 2.69136e-22
CAUTION: Gross error found at entry 207.
Found: 3.60822e-14 Expected 1.64941e-40 Error: 2.18759e+26
22.8625, 145.168, 672.121, 1, 1.64941e-40
CAUTION: Gross error found at entry 208.
Found: 3.73035e-14 Expected 1.6094e-61 Error: 2.31784e+47
22.8625, 148.129, 854.96, 1, 1.6094e-61
CAUTION: Gross error found at entry 225.
Found: 0 Expected 3.73672e-27 Error: 1.67937e+281
22.8625, 182.675, 616.613, 1, 3.73672e-27
CAUTION: Gross error found at entry 226.
Found: 0 Expected 8.85688e-49 Error: 3.98049e+259
22.8625, 183.147, 824.038, 1, 8.85688e-49
CAUTION: Gross error found at entry 227.
Found: 0 Expected 2.29176e-74 Error: 1.02997e+234
22.8625, 186.799, 1048.31, 1, 2.29176e-74
```

```
CAUTION: Gross error found at entry 282.
Found: 0 Expected 2.18831e-21 Error: 9.8348e+286
23.3804, 132.721, 468.305, 1, 2.18831e-21
CAUTION: Gross error found at entry 283.
Found: 0 Expected 1.3071e-38 Error: 5.87439e+269
23.3804, 135.747, 636.51, 1, 1.3071e-38
CAUTION: Gross error found at entry 284.
Found: 1.84297e-14 Expected 8.27843e-58 Error: 2.22623e+43
23.3804, 136.272, 798.262, 1, 8.27843e-58
CAUTION: Gross error found at entry 301.
Found: 0 Expected 9.85282e-26 Error: 4.42808e+282
23.3804, 169.826, 579.619, 1, 9.85282e-26
CAUTION: Gross error found at entry 302.
Found: 0 Expected 4.8094e-47 Error: 2.16145e+261
23.3804, 174.486, 791.465, 1, 4.8094e-47
CAUTION: Gross error found at entry 303.
Found: 1.11022e-16 Expected 6.70476e-71 Error: 1.65587e+54
23.3804, 175.686, 995.333, 1, 6.70476e-71
CAUTION: Gross error found at entry 358.
Found: 0 Expected 3.9702e-21 Error: 1.7843e+287
26.2704, 126.472, 458.227, 1, 3.9702e-21
CAUTION: Gross error found at entry 359.
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For non central t CDF complement with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central T

```
CAUTION: Gross error found at entry 56.
Found: 0.000186411 Expected 7.85192e-05 Error: 1.37408
61.6335, 46.2251, 68.8608, 0.999921, 7.85192e-05
CAUTION: Gross error found at entry 75.
Found: 0.00011439 Expected 5.05344e-05 Error: 1.26361
80.8418, 60.6313, 86.1278, 0.999949, 5.05344e-05
CAUTION: Gross error found at entry 93.
Found: 0.000655162 Expected 0.000423927 Error: 0.545458
100.733, 50.3663, 65.7619, 0.999576, 0.000423927
CAUTION: Gross error found at entry 112.
Found: 0.000518249 Expected 0.00034473 Error: 0.503348
127.953, 63.9764, 81.0824, 0.999655, 0.00034473
```

Error Output For non central t CDF with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Non Central T

```
CAUTION: Gross error found at entry 74.
Found: 0.000830062 Expected 0.000522858 Error: 0.587549
79.7478, -39.8739, -53.8066, 0.000522858, 0.999477
CAUTION: Gross error found at entry 94.
Found: 7.69292e-05 Expected 3.54024e-05 Error: 1.17299
101.191, -75.8936, -104.104, 3.54024e-05, 0.999965
CAUTION: Gross error found at entry 113.
Found: 5.07713e-05 Expected 2.4439e-05 Error: 1.07747
128.792, -96.5942, -128.112, 2.4439e-05, 0.999976
CAUTION: Gross error found at entry 132.
Found: 4.08612e-05 Expected 2.01542e-05 Error: 1.02743
146.56, -109.92, -143.392, 2.01542e-05, 0.99998
CAUTION: Gross error found at entry 151.
Found: 3.55146e-05 Expected 1.7803e-05 Error: 0.994869
159.586, -119.689, -154.522, 1.7803e-05, 0.999982
```

```
CAUTION: Gross error found at entry 170.
Found: 3.03671e-05 Expected 1.55023e-05 Error: 0.958873
175.686, -131.765, -168.211, 1.55023e-05, 0.999984
CAUTION: Gross error found at entry 189.
Found: 2.61339e-05 Expected 1.3581e-05 Error: 0.924298
192.978, -144.733, -182.834, 1.3581e-05, 0.999986
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Mathematica Data - Large orders and other bug cases

```
CAUTION: Found non-finite result, when a finite value was expected at entry 0
Found: -nan Expected 2.07309e+257 Error: 1.79769e+308
171, 2, 2.07309e+257
CAUTION: Gross error found at entry 0.
Found: -nan Expected 2.07309e+257 Error: 1.79769e+308
171, 2, 2.07309e+257
CAUTION: Found non-finite result, when a finite value was expected at entry 1
Found: -nan Expected 7.42912e+188 Error: 1.79769e+308
171, 5, 7.42912e+188
CAUTION: Gross error found at entry 1.
Found: -nan Expected 7.42912e+188 Error: 1.79769e+308
171, 5, 7.42912e+188
CAUTION: Found non-finite result, when a finite value was expected at entry 2
Found: -nan Expected -4.81295e+247 Error: 1.79769e+308
166, 2, -4.81295e+247
CAUTION: Gross error found at entry 2.
Found: -nan Expected -4.81295e+247 Error: 1.79769e+308
166, 2, -4.81295e+247
CAUTION: Found non-finite result, when a finite value was expected at entry 3
Found: -nan Expected -1.88439e+218 Error: 1.79769e+308
166, 3, -1.88439e+218
CAUTION: Gross error found at entry 3.
Found: -nan Expected -1.88439e+218 Error: 1.79769e+308
166, 3, -1.88439e+218
CAUTION: Found non-finite result, when a finite value was expected at entry 4
Found: -nan Expected 7.53144e+74 Error: 1.79769e+308
171, 23, 7.53144e+74
CAUTION: Gross error found at entry 4.
Found: -nan Expected 7.53144e+74 Error: 1.79769e+308
171, 23, 7.53144e+74
CAUTION: Found non-finite result, when a finite value was expected at entry 5
Found: -nan Expected -6.52661e-66 Error: 1.79769e+308
168, 150, -6.52661e-66
CAUTION: Gross error found at entry 5.
Found: -nan Expected -6.52661e-66 Error: 1.79769e+308
168, 150, -6.52661e-66
CAUTION: Found non-finite result, when a finite value was expected at entry 6
Found: -nan Expected 9.2734e-88 Error: 1.79769e+308
169, 202, 9.2734e-88
CAUTION: Gross error found at entry 6.
Found: -nan Expected 9.2734e-88 Error: 1.79769e+308
169, 202, 9.2734e-88
Outside supported domain
20, -9.5, -0.00103076
Outside supported domain
21, -9.5, 4.28582e+26
Outside supported domain
22, -9.5, -0.00419144
```

```
Outside supported domain
23, -9.5, 8.6745e+29
Outside supported domain
24, -9.5, -0.0204825
Outside supported domain
25, -9.5, 2.08188e+33
Outside supported domain
26, -9.5, -0.118403
Outside supported domain
27, -9.5, 5.84592e+36
Outside supported domain
28, -9.5, -0.798969
Outside supported domain
29, -9.5, 1.89875e+40
Outside supported domain
30, -9.5, -6.22245
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Mathematica Data - large negative arguments

```
Outside supported domain
124, -1.5, 7.63705e+240
Outside supported domain
124, -2.5, 7.63705e+240
Outside supported domain
124, -3.5, 7.63705e+240
Outside supported domain
124, -4.5, 7.63705e+240
Outside supported domain
124, -5.5, 7.63705e+240
Outside supported domain
124, -6.5, 7.63705e+240
Outside supported domain
124, -7.5, 7.63705e+240
Outside supported domain
124, -8.5, 7.63705e+240
Outside supported domain
124, -9.5, 7.63705e+240
Outside supported domain
124, -10.5, 7.63705e+240
Outside supported domain
124, -11.5, 7.63705e+240
Outside supported domain
124, -12.5, 7.63705e+240
Outside supported domain
124, -13.5, 7.63705e+240
Outside supported domain
124, -14.5, 7.63705e+240
Outside supported domain
124, -15.5, 7.63705e+240
Outside supported domain
124, -16.5, 7.63705e+240
Outside supported domain
124, -17.5, 7.63705e+240
Outside supported domain
124, -18.5, 7.63705e+240
Outside supported domain
124, -19.5, 7.63705e+240
```

```
Outside supported domain
124, -20.5, 7.63705e+240
Outside supported domain
124, -1.5, -7.63705e+240
Outside supported domain
124, -2.5, -7.63705e+240
Outside supported domain
124, -3.5, -7.63705e+240
Outside supported domain
124, -4.5, -7.63705e+240
Outside supported domain
124, -5.5, -7.63705e+240
Outside supported domain
124, -6.5, -7.63705e+240
Outside supported domain
124, -7.5, -7.63705e+240
Outside supported domain
124, -8.5, -7.63705e+240
Outside supported domain
124, -9.5, -7.63705e+240
Outside supported domain
124, -10.5, -7.63705e+240
Outside supported domain
124, -11.5, -7.63705e+240
Outside supported domain
124, -12.5, -7.63705e+240
Outside supported domain
124, -13.5, -7.63705e+240
Outside supported domain
124, -14.5, -7.63705e+240
Outside supported domain
124, -15.5, -7.63705e+240
Outside supported domain
124, -16.5, -7.63705e+240
Outside supported domain
124, -17.5, -7.63705e+240
Outside supported domain
124, -18.5, -7.63705e+240
Outside supported domain
124, -19.5, -7.63705e+240
Outside supported domain
124, -20.5, -7.63705e+240
Outside supported domain
1, -0.5, 8.9348
Outside supported domain
2, -0.5, -0.828797
Outside supported domain
3, -0.5, 193.409
Outside supported domain
4, -0.5, -3.47425
Outside supported domain
5, -0.5, 15371.1
Outside supported domain
6, -0.5, -43.4579
Outside supported domain
7, -0.5, 2.58068e+06
Outside supported domain
8, -0.5, -1059.96
```

```
Outside supported domain
9, -0.5, 7.43185e+08
Outside supported domain
10, -0.5, -42108.9
Outside supported domain
11, -0.5, 3.26999e+11
Outside supported domain
12, -0.5, -2.46448e+06
Outside supported domain
13, -0.5, 2.04047e+14
Outside supported domain
14, -0.5, -1.9918e+08
Outside supported domain
15, -0.5, 1.71399e+17
Outside supported domain
16, -0.5, -2.12394e+10
Outside supported domain
17, -0.5, 1.86483e+20
Outside supported domain
18, -0.5, -2.88824e+12
Outside supported domain
19, -0.5, 2.55108e+23
Outside supported domain
20, -0.5, -4.87773e+14
Outside supported domain
21, -0.5, 4.28582e+26
Outside supported domain
1, -0.5, 8.9348
Outside supported domain
2, -0.5, -0.828843
Outside supported domain
3, -0.5, 193.409
Outside supported domain
4, -0.5, -3.47791
Outside supported domain
5, -0.5, 15371.1
Outside supported domain
6, -0.5, -44.0732
Outside supported domain
7, -0.5, 2.58068e+06
Outside supported domain
8, -0.5, -1237.15
Outside supported domain
9, -0.5, 7.43185e+08
Outside supported domain
10, -0.5, -120071
Outside supported domain
11, -0.5, 3.26999e+11
Outside supported domain
12, -0.5, -5.11131e+07
Outside supported domain
13, -0.5, 2.04047e+14
Outside supported domain
14, -0.5, -4.1064e+10
Outside supported domain
15, -0.5, 1.71399e+17
Outside supported domain
16, -0.5, -4.44822e+13
```

```
Outside supported domain
17, -0.5, 1.86483e+20
Outside supported domain
18, -0.5, -6.08254e+16
Outside supported domain
19, -0.5, 2.55108e+23
Outside supported domain
20, -0.5, -1.02182e+20
Outside supported domain
21, -0.5, 4.28582e+26
Outside supported domain
1, -0.5, 8.9348
Outside supported domain
2, -0.5, -0.828751
Outside supported domain
3, -0.5, 193.409
Outside supported domain
4, -0.5, -3.47059
Outside supported domain
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Mathematica Data - negative arguments

```
Outside supported domain
1, -12.75, 19.6638
Outside supported domain
1, -12.25, 19.6608
Outside supported domain
1, -11.75, 19.6576
Outside supported domain
1, -11.25, 19.6542
Outside supported domain
1, -10.75, 19.6504
Outside supported domain
1, -10.25, 19.6463
Outside supported domain
1, -9.75, 19.6417
Outside supported domain
1, -9.25, 19.6367
Outside supported domain
1, -8.75, 19.6312
Outside supported domain
1, -8.25, 19.625
Outside supported domain
1, -7.75, 19.6181
Outside supported domain
1, -7.25, 19.6104
Outside supported domain
1, -6.75, 19.6015
Outside supported domain
1, -6.25, 19.5913
Outside supported domain
1, -5.75, 19.5795
Outside supported domain
1, -5.25, 19.5657
Outside supported domain
1, -4.75, 19.5493
```

```
Outside supported domain
1, -4.25, 19.5294
Outside supported domain
1, -3.75, 19.505
Outside supported domain
1, -3.25, 19.4741
Outside supported domain
1, -2.75, 19.4339
Outside supported domain
1, -2.25, 19.3794
Outside supported domain
1, -1.75, 19.3016
Outside supported domain
1, -1.25, 19.1819
Outside supported domain
1, -0.75, 18.9751
Outside supported domain
1, -0.25, 18.5419
Outside supported domain
2, -12.75, -124.031
Outside supported domain
2, -12.25, 124.019
Outside supported domain
2, -11.75, -124.032
Outside supported domain
2, -11.25, 124.018
Outside supported domain
2, -10.75, -124.033
Outside supported domain
2, -10.25, 124.016
Outside supported domain
2, -9.75, -124.035
Outside supported domain
2, -9.25, 124.015
Outside supported domain
2, -8.75, -124.037
Outside supported domain
2, -8.25, 124.012
Outside supported domain
2, -7.75, -124.04
Outside supported domain
2, -7.25, 124.009
Outside supported domain
2, -6.75, -124.044
Outside supported domain
2, -6.25, 124.003
Outside supported domain
2, -5.75, -124.051
Outside supported domain
2, -5.25, 123.995
Outside supported domain
2, -4.75, -124.061
Outside supported domain
2, -4.25, 123.981
Outside supported domain
2, -3.75, -124.08
Outside supported domain
2, -3.25, 123.955
```

```
Outside supported domain
2, -2.75, -124.118
Outside supported domain
2, -2.25, 123.897
Outside supported domain
2, -1.75, -124.214
Outside supported domain
2, -1.25, 123.721
Outside supported domain
2, -0.75, -124.587
Outside supported domain
2, -0.25, 122.697
Outside supported domain
3, -12.75, 1558.54
Outside supported domain
3, -12.25, 1558.54
Outside supported domain
3, -11.75, 1558.54
Outside supported domain
3, -11.25, 1558.54
Outside supported domain
3, -10.75, 1558.54
Outside supported domain
3, -10.25, 1558.54
Outside supported domain
3, -9.75, 1558.54
Outside supported domain
3, -9.25, 1558.54
Outside supported domain
3, -8.75, 1558.54
Outside supported domain
3, -8.25, 1558.54
Outside supported domain
3, -7.75, 1558.54
Outside supported domain
3, -7.25, 1558.54
Outside supported domain
3, -6.75, 1558.54
Outside supported domain
3, -6.25, 1558.54
Outside supported domain
3, -5.75, 1558.54
Outside supported domain
3, -5.25, 1558.54
Outside supported domain
3, -4.75, 1558.53
Outside supported domain
3, -4.25, 1558.53
Outside supported domain
3, -3.75, 1558.52
Outside supported domain
3, -3.25, 1558.51
Outside supported domain
3, -2.75, 1558.49
Outside supported domain
3, -2.25, 1558.46
Outside supported domain
3, -1.75, 1558.38
```

```
Outside supported domain
3, -1.25, 1558.22
Outside supported domain
3, -0.75, 1557.75
Outside supported domain
3, -0.25, 1555.76
Outside supported domain
4, -12.75, -24481.6
Outside supported domain
4, -12.25, 24481.6
Outside supported domain
4, -11.75, -24481.6
Outside supported domain
4, -11.25, 24481.6
Outside supported domain
4, -10.75, -24481.6
Outside supported domain
4, -10.25, 24481.6
Outside supported domain
4, -9.75, -24481.6
Outside supported domain
4, -9.25, 24481.6
Outside supported domain
4, -8.75, -24481.6
Outside supported domain
4, -8.25, 24481.6
Outside supported domain
4, -7.75, -24481.6
Outside supported domain
4, -7.25, 24481.6
Outside supported domain
4, -6.75, -24481.6
Outside supported domain
4, -6.25, 24481.6
Outside supported domain
4, -5.75, -24481.6
Outside supported domain
4, -5.25, 24481.6
Outside supported domain
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library Rmath 3.2.3 and test data Mathematica Data - large arguments

```
CAUTION: Gross error found at entry 211.
Found: -0 Expected -8.44974e-268 Error: 3.79751e+40
30, 8.58993e+09, -8.44974e-268
CAUTION: Gross error found at entry 212.
Found: -0 Expected -7.86943e-277 Error: 3.5367e+31
30, 1.71799e+10, -7.86943e-277
CAUTION: Gross error found at entry 213.
Found: -0 Expected -7.32898e-286 Error: 3.29381e+22
30, 3.43597e+10, -7.32898e-286
CAUTION: Gross error found at entry 214.
Found: -0 Expected -6.82564e-295 Error: 3.0676e+13
30, 6.87195e+10, -6.82564e-295
CAUTION: Gross error found at entry 215.
Found: -0 Expected -6.35687e-304 Error: 28568.3
```

```
30, 1.37439e+11, -6.35687e-304
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Iv: Mathworld Data (large values)

```
domain error
-1, 3.72917e-155, 1.86459e-155
domain error
-1.125, 3.72917e-155, -1.34964e+173
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Iv: Random Data

```
domain error
-80.4919, 24.7501, 4.18698e+28
domain error
-80.4919, 63.7722, 2.03248e+06
domain error
-74.6026, 24.7501, 7.20977e+23
domain error
-74.6026, 63.7722, 8.7549e+08
domain error
-72.9046, 24.7501, 1.04535e+22
domain error
-72.9046, 63.7722, 4.7162e+09
domain error
-62.3236, 24.7501, 3.65147e+14
domain error
-62.3236, 63.7722, 8.56683e+13
domain error
-55.7932, 24.7501, -7.70364e+09
domain error
-55.7932, 63.7722, 1.95969e+16
domain error
-44.3004, 9.50706, 2.93478e+22
domain error
-44.3004, 24.7501, 640.568
domain error
-44.3004, 63.7722, 8.05557e+19
domain error
-38.3666, 5.11399, 2.89105e+27
domain error
-38.3666, 9.50706, 8.80632e+16
domain error
-38.3666, 24.7501, 0.389004
domain error
-38.3666, 63.7722, 3.06303e+21
underflow
81.1584, 0.00177219, 0
underflow
81.1584, 0.00221773, 0
underflow
81.1584, 0.0074445, 6.08857e-319
underflow
82.6752, 0.00177219, 0
underflow
82.6752, 0.00221773, 0
underflow
```

```
82.6752, 0.0074445, 0
underflow
91.5014, 0.00177219, 0
underflow
91.5014, 0.00221773, 0
underflow
91.5014, 0.0074445, 0
underflow
91.5014, 0.014336, 0
underflow
91.5014, 0.0176092, 0
underflow
92.9777, 0.00177219, 0
underflow
92.9777, 0.00221773, 0
underflow
92.9777, 0.0074445, 0
underflow
92.9777, 0.014336, 0
underflow
92.9777, 0.0176092, 0
underflow
93.539, 0.00177219, 0
underflow
93.539, 0.00221773, 0
underflow
93.539, 0.0074445, 0
underflow
93.539, 0.014336, 0
underflow
93.539, 0.0176092, 0
underflow
93.7736, 0.00177219, 0
underflow
93.7736, 0.00221773, 0
underflow
93.7736, 0.0074445, 0
underflow
93.7736, 0.014336, 0
underflow
93.7736, 0.0176092, 0
underflow
98.5763, 0.00177219, 0
underflow
98.5763, 0.00221773, 0
underflow
98.5763, 0.0074445, 0
underflow
98.5763, 0.014336, 0
underflow
98.5763, 0.0176092, 0
underflow
99.2923, 0.00177219, 0
underflow
99.2923, 0.00221773, 0
underflow
99.2923, 0.0074445, 0
underflow
```

```
99.2923, 0.014336, 0
underflow
99.2923, 0.0176092, 0
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel In: Random Data

```
underflow
70, 0.00177219, 1.75887e-314
underflow
73, 0.00177219, 0
underflow
73, 0.00221773, 4.24896e-322
underflow
76, 0.00177219, 0
underflow
76, 0.00221773, 0
underflow
79, 0.00177219, 0
underflow
79, 0.00221773, 0
underflow
79, 0.0074445, 1.38676e-309
underflow
82, 0.00177219, 0
underflow
82, 0.00221773, 0
underflow
82, 0.0074445, 1.33398e-322
underflow
85, 0.00177219, 0
underflow
85, 0.00221773, 0
underflow
85, 0.0074445, 0
underflow
85, 0.014336, 1.81568e-311
underflow
88, 0.00177219, 0
underflow
88, 0.00221773, 0
underflow
88, 0.0074445, 0
underflow
88, 0.014336, 9.88131e-324
underflow
88, 0.0176092, 7.34647e-316
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Iv: Mathworld Data

```
domain error
-4.99902, 2.125, 0.0267921
domain error
-5.5, 10, 597.578
domain error
-5.5, 100, 9.22363e+41
domain error
```

```
-10.0003, 0.000976562, 1.41474e+35
domain error
-10.0003, 50, 1.07153e+20
domain error
-141.4, 100, 2066.28
```

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel In: Mathworld Data (Integer Version)

```
underflow
10, 1e-100, 0
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel In: Mathworld Data

```
domain error
-2, 0, 0
domain error
-5, 100, 9.47009e+41
domain error
-5, -1, -0.000271463
domain error
10, -5, 0.00458004
domain error
-100, -200, 4.35275e+74
underflow
10, 1e-100, 0
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel I1: Mathworld Data

```
domain error
1, -2, -1.59064
domain error
1, -8, -399.873
domain error
1, -10, -2670.99
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel I0: Mathworld Data

```
domain error
0, -2, 2.27959
domain error
0, -7, 168.594
domain error
0, -1, 1.26607
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Iv: Mathworld Data (large values)

```
Bad argument in cyl_bessel_i.
-1, 3.72917e-155, 1.86459e-155
Bad argument in cyl_bessel_i.
-1.125, 3.72917e-155, -1.34964e+173
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Iv: Random Data

```
Bad argument in cyl_bessel_i.
-80.4919, 24.7501, 4.18698e+28
Bad argument in cyl_bessel_i.
-80.4919, 63.7722, 2.03248e+06
Bad argument in cyl_bessel_i.
-74.6026, 24.7501, 7.20977e+23
Bad argument in cyl_bessel_i.
-74.6026, 63.7722, 8.7549e+08
Bad argument in cyl_bessel_i.
-72.9046, 24.7501, 1.04535e+22
Bad argument in cyl_bessel_i.
-72.9046, 63.7722, 4.7162e+09
Bad argument in cyl_bessel_i.
-62.3236, 24.7501, 3.65147e+14
Bad argument in cyl_bessel_i.
-62.3236, 63.7722, 8.56683e+13
Bad argument in cyl_bessel_i.
-55.7932, 24.7501, -7.70364e+09
Bad argument in cyl_bessel_i.
-55.7932, 63.7722, 1.95969e+16
Bad argument in cyl_bessel_i.
-44.3004, 9.50706, 2.93478e+22
Bad argument in cyl_bessel_i.
-44.3004, 24.7501, 640.568
Bad argument in cyl_bessel_i.
-44.3004, 63.7722, 8.05557e+19
Bad argument in cyl_bessel_i.
-38.3666, 5.11399, 2.89105e+27
Bad argument in cyl_bessel_i.
-38.3666, 9.50706, 8.80632e+16
Bad argument in cyl_bessel_i.
-38.3666, 24.7501, 0.389004
Bad argument in cyl_bessel_i.
-38.3666, 63.7722, 3.06303e+21
```

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Iv: Mathworld Data

```
Bad argument in cyl_bessel_i.
-4.99902, 2.125, 0.0267921
Bad argument in cyl_bessel_i.
-5.5, 10, 597.578
Bad argument in cyl_bessel_i.
-5.5, 100, 9.22363e+41
Bad argument in cyl_bessel_i.
-10.0003, 0.000976562, 1.41474e+35
Bad argument in cyl_bessel_i.
-10.0003, 50, 1.07153e+20
Bad argument in cyl_bessel_i.
-141.4, 100, 2066.28
```

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel In: Mathworld Data (Integer Version)

Bad argument in cyl_bessel_i.
-2, 0, 0
Bad argument in cyl_bessel_i.
-5, 100, 9.47009e+41
Bad argument in cyl_bessel_i.
-5, -1, -0.000271463
Bad argument in cyl_bessel_i.
10, -5, 0.00458004
Bad argument in cyl_bessel_i.
-100, -200, 4.35275e+74

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel I1: Mathworld Data (Integer Version)

Bad argument in cyl_bessel_i.
1, -2, -1.59064
Bad argument in cyl_bessel_i.
1, -8, -399.873
Bad argument in cyl_bessel_i.
1, -10, -2670.99

Error Output For cyl_bessel_i (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel I0: Mathworld Data (Integer Version)

Bad argument in cyl_bessel_i.
0, -2, 2.27959
Bad argument in cyl_bessel_i.
0, -7, 168.594
Bad argument in cyl_bessel_i.
0, -1, 1.26607

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel In: Mathworld Data

Bad argument in cyl_bessel_i.
-2, 0, 0
Bad argument in cyl_bessel_i.
-5, 100, 9.47009e+41
Bad argument in cyl_bessel_i.
-5, -1, -0.000271463
Bad argument in cyl_bessel_i.
10, -5, 0.00458004
Bad argument in cyl_bessel_i.
-100, -200, 4.35275e+74

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel I1: Mathworld Data

Bad argument in cyl_bessel_i.
1, -2, -1.59064
Bad argument in cyl_bessel_i.
1, -8, -399.873
Bad argument in cyl_bessel_i.
1, -10, -2670.99

Error Output For cyl_bessel_i with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel I0: Mathworld Data

```
Bad argument in cyl_bessel_i.
0, -2, 2.27959
Bad argument in cyl_bessel_i.
0, -7, 168.594
Bad argument in cyl_bessel_i.
0, -1, 1.26607
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel J: Random Data

```
underflow
63.8868, 5.5381e-05, 0
underflow
63.8868, 6.9304e-05, 0
underflow
63.8868, 0.000232641, 0
underflow
63.8868, 0.000448, 8.39912e-323
underflow
63.8868, 0.000550287, 4.32897e-317
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel J: Mathworld Data (large values)

```
domain error
-0.5, 1.2459e-206, 7.14823e+102
domain error
-256, 8, 0
domain error
-2.5, 4, -0.0145679
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel J: Mathworld Data

```
domain error
-5.5, 3.1416, -2.5582
domain error
-5.5, 10000, 0.00244984
domain error
-5.5, 10000, 0.00244984
domain error
-5.5, 1e+06, 0.000279243
domain error
-0.5, 101, 0.0708185
domain error
-10.0003, 0.000976562, 1.41474e+35
domain error
-10.0003, 15, -0.0902239
domain error
-10.0003, 100, -0.0547614
domain error
-10.0003, 20000, -0.00556869
domain error
-8.5, 12.5664, -0.257087
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel JN: Mathworld Data (Integer Version)

```
underflow
10, 1e-100, 0
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel JN: Mathworld Data

```
domain error
-1, 1.25, -0.510623
domain error
-2, 0, 0
domain error
5, -10, 0.234062
domain error
-5, 1e+06, 0.000725964
domain error
-5, -1, 0.000249758
domain error
10, -10, 0.207486
domain error
10, -5, 0.0014678
domain error
-10, 1e+06, -0.000331079
underflow
10, 1e-100, 0
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel J1: Mathworld Data

```
domain error
1, -2, -0.576725
domain error
1, -8, -0.234636
domain error
1, -10, -0.0434727
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel J0: Mathworld Data

```
domain error
0, -2, 0.223891
domain error
0, -8, 0.171651
domain error
0, -10, -0.245936
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J: Mathworld Data (large values)

```
Bad argument in cyl_bessel_j.
-0.5, 1.2459e-206, 7.14823e+102
Bad argument in cyl_bessel_j.
-256, 8, 1.46866e-353
Bad argument in cyl_bessel_j.
-2.5, 4, -0.0145679
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J: Mathworld Data

```
Bad argument in cyl_bessel_j.
-5.5, 3.1416, -2.5582
Bad argument in cyl_bessel_j.
-5.5, 10000, 0.00244984
Bad argument in cyl_bessel_j.
-5.5, 10000, 0.00244984
Bad argument in cyl_bessel_j.
-5.5, 1e+06, 0.000279243
Bad argument in cyl_bessel_j.
-0.5, 101, 0.0708185
Bad argument in cyl_bessel_j.
-10.0003, 0.000976562, 1.41474e+35
Bad argument in cyl_bessel_j.
-10.0003, 15, -0.0902239
Bad argument in cyl_bessel_j.
-10.0003, 100, -0.0547614
Bad argument in cyl_bessel_j.
-10.0003, 20000, -0.00556869
Bad argument in cyl_bessel_j.
-8.5, 12.5664, -0.257087
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel JN: Mathworld Data (Integer Version)

```
Bad argument in cyl_bessel_j.
-1, 1.25, -0.510623
Bad argument in cyl_bessel_j.
-2, 0, 0
Bad argument in cyl_bessel_j.
5, -10, 0.234062
Bad argument in cyl_bessel_j.
-5, 1e+06, 0.000725964
Bad argument in cyl_bessel_j.
-5, -1, 0.000249758
Bad argument in cyl_bessel_j.
10, -10, 0.207486
Bad argument in cyl_bessel_j.
10, -5, 0.0014678
Bad argument in cyl_bessel_j.
-10, 1e+06, -0.000331079
CAUTION: Gross error found at entry 15.
Found: 0.0042409 Expected 0.00128318 Error: 2.305
1000, 100000, 0.00128318
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J1: Mathworld Data (Integer Version)

```
Bad argument in cyl_bessel_j.
1, -2, -0.576725
Bad argument in cyl_bessel_j.
1, -8, -0.234636
Bad argument in cyl_bessel_j.
1, -10, -0.0434727
```

Error Output For cyl_bessel_j (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J0: Mathworld Data (Integer Version)

```
Bad argument in cyl_bessel_j.
0, -2, 0.223891
Bad argument in cyl_bessel_j.
0, -8, 0.171651
Bad argument in cyl_bessel_j.
0, -10, -0.245936
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel JN: Mathworld Data

```
Bad argument in cyl_bessel_j.
-1, 1.25, -0.510623
Bad argument in cyl_bessel_j.
-2, 0, 0
Bad argument in cyl_bessel_j.
5, -10, 0.234062
Bad argument in cyl_bessel_j.
-5, 1e+06, 0.000725964
Bad argument in cyl_bessel_j.
-5, -1, 0.000249758
Bad argument in cyl_bessel_j.
10, -10, 0.207486
Bad argument in cyl_bessel_j.
10, -5, 0.0014678
Bad argument in cyl_bessel_j.
-10, 1e+06, -0.000331079
CAUTION: Gross error found at entry 15.
Found: 0.0042409 Expected 0.00128318 Error: 2.305
1000, 100000, 0.00128318
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J1: Mathworld Data

```
Bad argument in cyl_bessel_j.
1, -2, -0.576725
Bad argument in cyl_bessel_j.
1, -8, -0.234636
Bad argument in cyl_bessel_j.
1, -10, -0.0434727
```

Error Output For cyl_bessel_j with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel J0: Mathworld Data

```
Bad argument in cyl_bessel_j.
0, -2, 0.223891
Bad argument in cyl_bessel_j.
0, -8, 0.171651
Bad argument in cyl_bessel_j.
0, -10, -0.245936
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Kv: Random Data

```
domain error
```

```
-80.4919, 24.7501, 6.57902e+28
domain error
-80.4919, 63.7722, 2.39552e-09
domain error
-80.4919, 125.28, 3.06904e-45
domain error
-80.4919, 255.547, 2.30343e-107
domain error
-80.4919, 503.011, 1.20315e-217
domain error
-80.4919, 1007.46, 0
domain error
-80.4919, 1185.4, 0
domain error
-80.4919, 3534.52, 0
domain error
-80.4919, 8071.55, 0
domain error
-80.4919, 16229.2, 0
domain error
-80.4919, 32066.2, 0
domain error
-80.4919, 36367.9, 0
domain error
-74.6026, 24.7501, 1.19405e+24
domain error
-74.6026, 63.7722, 5.81897e-12
domain error
-74.6026, 125.28, 9.89214e-47
domain error
-74.6026, 255.547, 3.9726e-108
domain error
-74.6026, 503.011, 4.87462e-218
domain error
-74.6026, 1007.46, 0
domain error
-74.6026, 1185.4, 0
domain error
-74.6026, 3534.52, 0
domain error
-74.6026, 8071.55, 0
domain error
-74.6026, 16229.2, 0
domain error
-74.6026, 32066.2, 0
domain error
-74.6026, 36367.9, 0
domain error
-72.9046, 24.7501, 5.5618e+22
domain error
-72.9046, 63.7722, 1.09452e-12
domain error
-72.9046, 125.28, 3.8393e-47
domain error
-72.9046, 255.547, 2.45173e-108
domain error
-72.9046, 503.011, 3.80454e-218
domain error
```

```
-72.9046, 1007.46, 0
domain error
-72.9046, 1185.4, 0
domain error
-72.9046, 3534.52, 0
domain error
-72.9046, 8071.55, 0
domain error
-72.9046, 16229.2, 0
domain error
-72.9046, 32066.2, 0
domain error
-72.9046, 36367.9, 0
domain error
-62.3236, 24.7501, 6.74518e+14
domain error
-62.3236, 63.7722, 6.54531e-17
domain error
-62.3236, 125.28, 1.65653e-49
domain error
-62.3236, 255.547, 1.54767e-109
domain error
-62.3236, 503.011, 9.22721e-219
domain error
-62.3236, 1007.46, 0
domain error
-62.3236, 1185.4, 0
domain error
-62.3236, 3534.52, 0
domain error
-62.3236, 8071.55, 0
domain error
-62.3236, 16229.2, 0
domain error
-62.3236, 32066.2, 0
domain error
-62.3236, 36367.9, 0
domain error
-55.7932, 24.7501, 2.00028e+10
domain error
-55.7932, 63.7722, 3.01107e-19
domain error
-55.7932, 125.28, 8.54693e-51
domain error
-55.7932, 255.547, 3.47666e-110
domain error
-55.7932, 503.011, 4.29705e-219
domain error
-55.7932, 1007.46, 0
domain error
-55.7932, 1185.4, 0
domain error
-55.7932, 3534.52, 0
domain error
-55.7932, 8071.55, 0
domain error
-55.7932, 16229.2, 0
domain error
```

```
-55.7932, 32066.2, 0
domain error
-55.7932, 36367.9, 0
domain error
-44.3004, 9.50706, 5.6936e+22
domain error
-44.3004, 24.7501, 1242.73
domain error
-44.3004, 63.7722, 7.99341e-23
domain error
-44.3004, 125.28, 9.88149e-53
domain error
-44.3004, 255.547, 3.73007e-111
domain error
-44.3004, 503.011, 1.37367e-219
domain error
-44.3004, 1007.46, 0
domain error
-44.3004, 1185.4, 0
domain error
-44.3004, 3534.52, 0
domain error
-44.3004, 8071.55, 0
domain error
-44.3004, 16229.2, 0
domain error
-44.3004, 32066.2, 0
domain error
-44.3004, 36367.9, 0
domain error
-38.3666, 5.11399, 4.97154e+27
domain error
-38.3666, 9.50706, 1.51436e+17
domain error
-38.3666, 24.7501, 0.639495
domain error
-38.3666, 63.7722, 2.19334e-24
domain error
-38.3666, 125.28, 1.45351e-53
domain error
-38.3666, 255.547, 1.43713e-111
domain error
-38.3666, 503.011, 8.44445e-220
domain error
-38.3666, 1007.46, 0
domain error
-38.3666, 1185.4, 0
domain error
-38.3666, 3534.52, 0
domain error
-38.3666, 8071.55, 0
domain error
-38.3666, 16229.2, 0
domain error
-38.3666, 32066.2, 0
domain error
-38.3666, 36367.9, 0
underflow
```

```
9.3763, 1007.46, 0
underflow
9.3763, 1185.4, 0
underflow
9.3763, 3534.52, 0
underflow
9.3763, 8071.55, 0
underflow
9.3763, 16229.2, 0
underflow
9.3763, 32066.2, 0
underflow
9.3763, 36367.9, 0
underflow
9.44412, 1007.46, 0
underflow
9.44412, 1185.4, 0
underflow
9.44412, 3534.52, 0
underflow
9.44412, 8071.55, 0
underflow
9.44412, 16229.2, 0
underflow
9.44412, 32066.2, 0
underflow
9.44412, 36367.9, 0
underflow
26.4719, 1007.46, 0
underflow
26.4719, 1185.4, 0
underflow
26.4719, 3534.52, 0
underflow
26.4719, 8071.55, 0
underflow
26.4719, 16229.2, 0
underflow
26.4719, 32066.2, 0
underflow
26.4719, 36367.9, 0
underflow
62.9447, 1007.46, 0
underflow
62.9447, 1185.4, 0
underflow
62.9447, 3534.52, 0
underflow
62.9447, 8071.55, 0
underflow
62.9447, 16229.2, 0
underflow
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Kn: Random Data

```
underflow
```

```
0, 1007.46, 0
underflow
0, 1185.4, 0
underflow
0, 3534.52, 0
underflow
0, 8071.55, 0
underflow
0, 16229.2, 0
underflow
0, 32066.2, 0
underflow
0, 36367.9, 0
underflow
1, 1007.46, 0
underflow
1, 1185.4, 0
underflow
1, 3534.52, 0
underflow
1, 8071.55, 0
underflow
1, 16229.2, 0
underflow
1, 32066.2, 0
underflow
1, 36367.9, 0
underflow
4, 1007.46, 0
underflow
4, 1185.4, 0
underflow
4, 3534.52, 0
underflow
4, 8071.55, 0
underflow
4, 16229.2, 0
underflow
4, 32066.2, 0
underflow
4, 36367.9, 0
underflow
7, 1007.46, 0
underflow
7, 1185.4, 0
underflow
7, 3534.52, 0
underflow
7, 8071.55, 0
underflow
7, 16229.2, 0
underflow
7, 32066.2, 0
underflow
7, 36367.9, 0
underflow
10, 1007.46, 0
underflow
```

```
10, 1185.4, 0
underflow
10, 3534.52, 0
underflow
10, 8071.55, 0
underflow
10, 16229.2, 0
underflow
10, 32066.2, 0
underflow
10, 36367.9, 0
underflow
13, 1007.46, 0
underflow
13, 1185.4, 0
underflow
13, 3534.52, 0
underflow
13, 8071.55, 0
underflow
13, 16229.2, 0
underflow
13, 32066.2, 0
underflow
13, 36367.9, 0
underflow
16, 1007.46, 0
underflow
16, 1185.4, 0
underflow
16, 3534.52, 0
underflow
16, 8071.55, 0
underflow
16, 16229.2, 0
underflow
16, 32066.2, 0
underflow
16, 36367.9, 0
underflow
19, 1007.46, 0
underflow
19, 1185.4, 0
underflow
19, 3534.52, 0
underflow
19, 8071.55, 0
underflow
19, 16229.2, 0
underflow
19, 32066.2, 0
underflow
19, 36367.9, 0
underflow
22, 1007.46, 0
underflow
22, 1185.4, 0
underflow
```

```
22, 3534.52, 0
underflow
22, 8071.55, 0
underflow
22, 16229.2, 0
underflow
22, 32066.2, 0
underflow
22, 36367.9, 0
underflow
25, 1007.46, 0
underflow
25, 1185.4, 0
underflow
25, 3534.52, 0
underflow
25, 8071.55, 0
underflow
25, 16229.2, 0
underflow
25, 32066.2, 0
underflow
25, 36367.9, 0
underflow
28, 1007.46, 0
underflow
28, 1185.4, 0
underflow
28, 3534.52, 0
underflow
28, 8071.55, 0
underflow
28, 16229.2, 0
underflow
28, 32066.2, 0
underflow
28, 36367.9, 0
underflow
31, 1007.46, 0
underflow
31, 1185.4, 0
underflow
31, 3534.52, 0
underflow
31, 8071.55, 0
underflow
31, 16229.2, 0
underflow
31, 32066.2, 0
underflow
31, 36367.9, 0
underflow
34, 1007.46, 0
underflow
34, 1185.4, 0
underflow
34, 3534.52, 0
underflow
```

```
34, 8071.55, 0
underflow
34, 16229.2, 0
underflow
34, 32066.2, 0
underflow
34, 36367.9, 0
underflow
37, 1007.46, 0
underflow
37, 1185.4, 0
underflow
37, 3534.52, 0
underflow
37, 8071.55, 0
underflow
37, 16229.2, 0
underflow
37, 32066.2, 0
underflow
37, 36367.9, 0
underflow
40, 1007.46, 0
underflow
40, 1185.4, 0
underflow
40, 3534.52, 0
underflow
40, 8071.55, 0
underflow
40, 16229.2, 0
underflow
40, 32066.2, 0
underflow
40, 36367.9, 0
underflow
43, 1007.46, 0
underflow
43, 1185.4, 0
underflow
43, 3534.52, 0
underflow
43, 8071.55, 0
underflow
43, 16229.2, 0
underflow
43, 32066.2, 0
underflow
43, 36367.9, 0
underflow
46, 1007.46, 0
underflow
46, 1185.4, 0
underflow
46, 3534.52, 0
underflow
46, 8071.55, 0
underflow
```

```
46, 16229.2, 0
underflow
46, 32066.2, 0
underflow
46, 36367.9, 0
underflow
49, 1007.46, 0
underflow
49, 1185.4, 0
underflow
49, 3534.52, 0
underflow
49, 8071.55, 0
underflow
49, 16229.2, 0
underflow
49, 32066.2, 0
underflow
49, 36367.9, 0
underflow
52, 1007.46, 0
underflow
52, 1185.4, 0
underflow
52, 3534.52, 0
underflow
52, 8071.55, 0
underflow
52, 16229.2, 0
underflow
52, 32066.2, 0
underflow
52, 36367.9, 0
underflow
55, 1007.46, 0
underflow
55, 1185.4, 0
underflow
55, 3534.52, 0
underflow
55, 8071.55, 0
underflow
55, 16229.2, 0
underflow
55, 32066.2, 0
underflow
55, 36367.9, 0
underflow
58, 1007.46, 0
underflow
58, 1185.4, 0
underflow
58, 3534.52, 0
underflow
58, 8071.55, 0
underflow
58, 16229.2, 0
underflow
```

```
58, 32066.2, 0
underflow
58, 36367.9, 0
underflow
61, 1007.46, 0
underflow
61, 1185.4, 0
underflow
61, 3534.52, 0
underflow
61, 8071.55, 0
underflow
61, 16229.2, 0
underflow
61, 32066.2, 0
underflow
61, 36367.9, 0
underflow
64, 1007.46, 0
underflow
64, 1185.4, 0
underflow
64, 3534.52, 0
underflow
64, 8071.55, 0
underflow
64, 16229.2, 0
underflow
64, 32066.2, 0
underflow
64, 36367.9, 0
underflow
67, 1007.46, 0
underflow
67, 1185.4, 0
underflow
67, 3534.52, 0
underflow
67, 8071.55, 0
underflow
67, 16229.2, 0
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Kv: Mathworld Data (large values)

```
domain error
-1, 3.72917e-155, 2.68156e+154
domain error
-1.125, 3.72917e-155, 5.53984e+173
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Kv: Mathworld Data

```
domain error
-5.5, 10, 7.33045e-05
domain error
-5.5, 100, 5.41275e-45
```

```
domain error
-141.399, 50, 1.30185e+42
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Bessel Kn: Mathworld Data

```
domain error
-5, 100, 5.27326e-45
domain error
-10, 1, 1.80713e+08
domain error
-1000, 700, 6.51562e-31
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Kv: Random Data

```
Bad argument in cyl_bessel_k.
-80.4919, 24.7501, 6.57902e+28
Bad argument in cyl_bessel_k.
-80.4919, 63.7722, 2.39552e-09
Bad argument in cyl_bessel_k.
-80.4919, 125.28, 3.06904e-45
Bad argument in cyl_bessel_k.
-80.4919, 255.547, 2.30343e-107
Bad argument in cyl_bessel_k.
-80.4919, 503.011, 1.20315e-217
Bad argument in cyl_bessel_k.
-80.4919, 1007.46, 2.86537e-438
Bad argument in cyl_bessel_k.
-80.4919, 1185.4, 8.63263e-516
Bad argument in cyl_bessel_k.
-80.4919, 3534.52, 5.01367e-1537
Bad argument in cyl_bessel_k.
-80.4919, 8071.55, 7.76555e-3508
Bad argument in cyl_bessel_k.
-80.4919, 16229.2, 0
Bad argument in cyl_bessel_k.
-80.4919, 32066.2, 0
Bad argument in cyl_bessel_k.
-80.4919, 36367.9, 0
Bad argument in cyl_bessel_k.
-74.6026, 24.7501, 1.19405e+24
Bad argument in cyl_bessel_k.
-74.6026, 63.7722, 5.81897e-12
Bad argument in cyl_bessel_k.
-74.6026, 125.28, 9.89214e-47
Bad argument in cyl_bessel_k.
-74.6026, 255.547, 3.9726e-108
Bad argument in cyl_bessel_k.
-74.6026, 503.011, 4.87462e-218
Bad argument in cyl_bessel_k.
-74.6026, 1007.46, 1.82221e-438
Bad argument in cyl_bessel_k.
-74.6026, 1185.4, 5.87506e-516
Bad argument in cyl_bessel_k.
-74.6026, 3534.52, 4.40608e-1537
Bad argument in cyl_bessel_k.
-74.6026, 8071.55, 7.3384e-3508
```

```
Bad argument in cyl_bessel_k.
-74.6026, 16229.2, 0
Bad argument in cyl_bessel_k.
-74.6026, 32066.2, 0
Bad argument in cyl_bessel_k.
-74.6026, 36367.9, 0
Bad argument in cyl_bessel_k.
-72.9046, 24.7501, 5.5618e+22
Bad argument in cyl_bessel_k.
-72.9046, 63.7722, 1.09452e-12
Bad argument in cyl_bessel_k.
-72.9046, 125.28, 3.8393e-47
Bad argument in cyl_bessel_k.
-72.9046, 255.547, 2.45173e-108
Bad argument in cyl_bessel_k.
-72.9046, 503.011, 3.80454e-218
Bad argument in cyl_bessel_k.
-72.9046, 1007.46, 1.60949e-438
Bad argument in cyl_bessel_k.
-72.9046, 1185.4, 5.28662e-516
Bad argument in cyl_bessel_k.
-72.9046, 3534.52, 4.25273e-1537
Bad argument in cyl_bessel_k.
-72.9046, 8071.55, 7.22542e-3508
Bad argument in cyl_bessel_k.
-72.9046, 16229.2, 0
Bad argument in cyl_bessel_k.
-72.9046, 32066.2, 0
Bad argument in cyl_bessel_k.
-72.9046, 36367.9, 0
Bad argument in cyl_bessel_k.
-62.3236, 24.7501, 6.74518e+14
Bad argument in cyl_bessel_k.
-62.3236, 63.7722, 6.54531e-17
Bad argument in cyl_bessel_k.
-62.3236, 125.28, 1.65653e-49
Bad argument in cyl_bessel_k.
-62.3236, 255.547, 1.54767e-109
Bad argument in cyl_bessel_k.
-62.3236, 503.011, 9.22721e-219
Bad argument in cyl_bessel_k.
-62.3236, 1007.46, 7.91894e-439
Bad argument in cyl_bessel_k.
-62.3236, 1185.4, 2.89281e-516
Bad argument in cyl_bessel_k.
-62.3236, 3534.52, 3.4736e-1537
Bad argument in cyl_bessel_k.
-62.3236, 8071.55, 6.6126e-3508
Bad argument in cyl_bessel_k.
-62.3236, 16229.2, 0
Bad argument in cyl_bessel_k.
-62.3236, 32066.2, 0
Bad argument in cyl_bessel_k.
-62.3236, 36367.9, 0
Bad argument in cyl_bessel_k.
-55.7932, 24.7501, 2.00028e+10
Bad argument in cyl_bessel_k.
-55.7932, 63.7722, 3.01107e-19
```

Bad argument in cyl_bessel_k.
-55.7932, 125.28, 8.54693e-51
Bad argument in cyl_bessel_k.
-55.7932, 255.547, 3.47666e-110
Bad argument in cyl_bessel_k.
-55.7932, 503.011, 4.29705e-219
Bad argument in cyl_bessel_k.
-55.7932, 1007.46, 5.40242e-439
Bad argument in cyl_bessel_k.
-55.7932, 1185.4, 2.08996e-516
Bad argument in cyl_bessel_k.
-55.7932, 3534.52, 3.11458e-1537
Bad argument in cyl_bessel_k.
-55.7932, 8071.55, 6.30409e-3508
Bad argument in cyl_bessel_k.
-55.7932, 16229.2, 0
Bad argument in cyl_bessel_k.
-55.7932, 32066.2, 0
Bad argument in cyl_bessel_k.
-55.7932, 36367.9, 0
Bad argument in cyl_bessel_k.
-44.3004, 9.50706, 5.6936e+22
Bad argument in cyl_bessel_k.
-44.3004, 24.7501, 1242.73
Bad argument in cyl_bessel_k.
-44.3004, 63.7722, 7.99341e-23
Bad argument in cyl_bessel_k.
-44.3004, 125.28, 9.88149e-53
Bad argument in cyl_bessel_k.
-44.3004, 255.547, 3.73007e-111
Bad argument in cyl_bessel_k.
-44.3004, 503.011, 1.37367e-219
Bad argument in cyl_bessel_k.
-44.3004, 1007.46, 3.05398e-439
Bad argument in cyl_bessel_k.
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Kv: Mathworld Data (large values)

Bad argument in cyl_bessel_k.
-1, 3.72917e-155, 2.68156e+154
Bad argument in cyl_bessel_k.
-1.125, 3.72917e-155, 5.53984e+173

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Kv: Mathworld Data

Bad argument in cyl_bessel_k.
-5.5, 10, 7.33045e-05
Bad argument in cyl_bessel_k.
-5.5, 100, 5.41275e-45
Bad argument in cyl_bessel_k.
-141.399, 50, 1.30185e+42

Error Output For cyl_bessel_k (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Kn: Mathworld Data (Integer Version)

```
Bad argument in cyl_bessel_k.
-5, 100, 5.27326e-45
Bad argument in cyl_bessel_k.
-10, 1, 1.80713e+08
Bad argument in cyl_bessel_k.
-1000, 700, 6.51562e-31
```

Error Output For cyl_bessel_k with compiler GNU C++ version 7.1.0 and library <cmath> and test data Bessel Kn: Mathworld Data

```
Bad argument in cyl_bessel_k.
-5, 100, 5.27326e-45
Bad argument in cyl_bessel_k.
-10, 1, 1.80713e+08
Bad argument in cyl_bessel_k.
-1000, 700, 6.51562e-31
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Yv: Mathworld Data (large values)

```
domain error
-0.5, 1.2459e-206, 8.90598e-104
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Yv: Mathworld Data

```
domain error
-5.5, 3.125, -0.0274994
domain error
-5.5, 10000, -0.00759344
domain error
-10.0003, 0.000976562, -1.50382e+38
domain error
-10.0003, 100, 0.0583042
domain error
-141.75, 100, -3.8101e+09
domain error
-8.5, 12.5664, 0.0436808
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Yn: Mathworld Data

```
domain error
-5, 1e+06, 0.000331052
domain error
-10, 1e+06, 0.000725952
domain error
-1000, 700, -1.88753e+77
domain error
-25, 8, 3.45114e+08
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library <cmath> and test data Yv: Random Data

```
CAUTION: Gross error found at entry 394.
Found: -3.29903 Expected 0.0192842 Error: 1.18973e+4932
125.28, 1007.46, 0.0192842
CAUTION: Gross error found at entry 395.
Found: 1.13543 Expected 0.0230358 Error: 48.2897
125.28, 1185.4, 0.0230358
CAUTION: Gross error found at entry 396.
Found: 0.00119445 Expected 0.00460223 Error: 2.85302
125.28, 3534.52, 0.00460223
CAUTION: Gross error found at entry 403.
Found: 1068 Expected -0.00270959 Error: 1.18973e+4932
255.547, 1007.46, -0.00270959
CAUTION: Gross error found at entry 404.
Found: -395.006 Expected 0.00738845 Error: 1.18973e+4932
255.547, 1185.4, 0.00738845
CAUTION: Gross error found at entry 405.
Found: 1.08701 Expected -0.000407036 Error: 1.18973e+4932
255.547, 3534.52, -0.000407036
CAUTION: Gross error found at entry 406.
Found: 0.0232211 Expected 0.00886946 Error: 1.61809
255.547, 8071.55, 0.00886946
CAUTION: Gross error found at entry 411.
Found: 65895.7 Expected -0.0158467 Error: 1.18973e+4932
503.011, 1007.46, -0.0158467
CAUTION: Gross error found at entry 412.
Found: -123316 Expected 0.00594357 Error: 1.18973e+4932
503.011, 1185.4, 0.00594357
CAUTION: Gross error found at entry 413.
Found: -706.209 Expected 0.010151 Error: 1.18973e+4932
503.011, 3534.52, 0.010151
CAUTION: Gross error found at entry 414.
Found: -21.2081 Expected 0.00888375 Error: 1.18973e+4932
503.011, 8071.55, 0.00888375
CAUTION: Gross error found at entry 415.
Found: 0.0272835 Expected 0.00552287 Error: 3.94008
503.011, 16229.2, 0.00552287
CAUTION: Gross error found at entry 416.
Found: 0.0103324 Expected 0.00445559 Error: 1.31898
503.011, 32066.2, 0.00445559
CAUTION: Gross error found at entry 417.
Found: 0.00540788 Expected -0.00384344 Error: 1.18973e+4932
503.011, 36367.9, -0.00384344
CAUTION: Gross error found at entry 418.
Found: 5.43091e+07 Expected -0.0772843 Error: 1.18973e+4932
1007.46, 1007.46, -0.0772843
CAUTION: Gross error found at entry 419.
Found: -2.84383e+07 Expected 0.0304312 Error: 1.18973e+4932
1007.46, 1185.4, 0.0304312
CAUTION: Gross error found at entry 420.
Found: -61440.2 Expected -0.00474217 Error: 1.29562e+07
1007.46, 3534.52, -0.00474217
CAUTION: Gross error found at entry 421.
Found: -4126.89 Expected -0.0074205 Error: 556146
1007.46, 8071.55, -0.0074205
```

```
CAUTION: Gross error found at entry 422.
Found: -69.2831 Expected -0.00179572 Error: 38581.4
1007.46, 16229.2, -0.00179572
CAUTION: Gross error found at entry 423.
Found: 2.32048 Expected 0.000750053 Error: 3092.76
1007.46, 32066.2, 0.000750053
CAUTION: Gross error found at entry 424.
Found: 3.90724 Expected 0.00305125 Error: 1279.54
1007.46, 36367.9, 0.00305125
CAUTION: Gross error found at entry 425.
Found: -1.83374e+08 Expected -7.25176e+28 Error: 3.95463e+20
1185.4, 1007.46, -7.25176e+28
CAUTION: Gross error found at entry 426.
Found: 1.09822e+08 Expected -0.0732059 Error: 1.18973e+4932
1185.4, 1185.4, -0.0732059
CAUTION: Gross error found at entry 427.
Found: 315632 Expected 0.000479585 Error: 6.58136e+08
1185.4, 3534.52, 0.000479585
CAUTION: Gross error found at entry 428.
Found: 16815.6 Expected 0.00174909 Error: 9.61391e+06
1185.4, 8071.55, 0.00174909
CAUTION: Gross error found at entry 429.
Found: 133.356 Expected 0.00416288 Error: 32033.6
1185.4, 16229.2, 0.00416288
CAUTION: Gross error found at entry 430.
Found: -1.38401 Expected -0.000320056 Error: 4323.27
1185.4, 32066.2, -0.000320056
CAUTION: Gross error found at entry 431.
Found: -17.7085 Expected -0.00417656 Error: 4238.96
1185.4, 36367.9, -0.00417656
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library <cmath> and test data Yv: Mathworld Data (large values)

```
Bad argument in __cyl_neumann_n.
-0.5, 1.2459e-206, 8.90598e-104
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library <cmath> and test data Yv: Mathworld Data

```
Bad argument in __cyl_neumann_n.
-5.5, 3.125, -0.0274994
Bad argument in __cyl_neumann_n.
-5.5, 10000, -0.00759344
Bad argument in __cyl_neumann_n.
-10.0003, 0.000976562, -1.50382e+38
Bad argument in __cyl_neumann_n.
-10.0003, 100, 0.0583042
Bad argument in __cyl_neumann_n.
-141.75, 100, -3.8101e+09
Bad argument in __cyl_neumann_n.
-8.5, 12.5664, 0.0436808
```

Error Output For cyl_neumann (integer orders) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Yn: Mathworld Data (Integer Version)

```
Bad argument in __cyl_neumann_n.
```

```
-5, 1e+06, 0.000331052
Bad argument in __cyl_neumann_n.
-10, 1e+06, 0.000725952
CAUTION: Gross error found at entry 7.
Found: 0.0540745 Expected 0.00217255 Error: 23.8899
1000, 100000, 0.00217255
Bad argument in __cyl_neumann_n.
-1000, 700, -1.88753e+77
Bad argument in __cyl_neumann_n.
-25, 8, 3.45114e+08
```

Error Output For cyl_neumann with compiler GNU C++ version 7.1.0 and library <cmath> and test data Yn: Mathworld Data

```
Bad argument in __cyl_neumann_n.
-5, 1e+06, 0.000331052
Bad argument in __cyl_neumann_n.
-10, 1e+06, 0.000725952
CAUTION: Gross error found at entry 7.
Found: 0.0540745 Expected 0.00217255 Error: 23.8899
1000, 100000, 0.00217255
Bad argument in __cyl_neumann_n.
-1000, 700, -1.88753e+77
Bad argument in __cyl_neumann_n.
-25, 8, 3.45114e+08
```

Error Output For beta with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Beta Function: Small Values

```
CAUTION: Found non-finite result, when a finite value was expected at entry 22
Found: inf Expected 5.69832e+154 Error: 1.79769e+308
2.98334e-154, 1.86459e-155, 5.69832e+154
CAUTION: Gross error found at entry 22.
Found: inf Expected 5.69832e+154 Error: 1.79769e+308
2.98334e-154, 1.86459e-155, 5.69832e+154
```

Error Output For ellint_rj with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data RJ: Random data

```
domain error
1.77787e-31, 1.40657e+18, 10.046, -4.8298e-10, -2.51795e-10
domain error
3.37448e-31, 4.65772e+22, 0.469831, -4.33756e-09, -2.95865e-11
domain error
5.25297e-31, 5.85483e+25, 2.02482e-15, -1.87347e-28, 3.36445e+07
domain error
6.22216e-31, 3.43401e+23, 0.673005, -2.7626e-13, -7.58898e-12
domain error
6.26875e-31, 2.62568e-13, 1.06394e+24, -1.36451e+14, -6.70372e-25
domain error
6.84599e-31, 3.57666e-29, 1.82191e+11, -3.63292e+08, -8.35235e-13
domain error
8.90482e-31, 1.97093e-28, 1.14939e-31, -1.26424e-12, -6.39454e+26
domain error
1.07374e-30, 1.70005e-12, 1.88773e-25, -1.16558e-29, 4.31668e+32
domain error
1.17141e-30, 24.2523, 3.67522e+21, -4.79065e-22, 2.2702e-05
```

```
domain error
1.64143e-30, 2.01978e-22, 2.58942e+12, -8.52649e-12, -2.82629e+06
domain error
1.85141e-30, 0.0386712, 2.37846e-13, -1.57357e+15, -1.38574e-13
domain error
2.70034e-30, 4.43896e-24, 7.54576e+16, -1.1436e-14, -1.10082e+07
domain error
4.01162e-30, 2.73343e+23, 1.32333e+13, -1.86032e-07, -4.16626e-25
domain error
4.13665e-30, 1.08034e-30, 3.13547e-16, -5.58099e-08, -5.14643e+16
domain error
4.3728e-30, 7.79812e+12, 8.58894e+21, -4.58312e-24, 5.28901e-09
domain error
5.6397e-30, 1.64768e+23, 9.64423e-15, -1.82207e+20, -1.62886e-30
domain error
9.89841e-30, 9.69731e+10, 1.03263e+21, -0.00343967, -9.62714e-22
domain error
1.3797e-29, 6.03357e+08, 5.62497e-15, -5.87235e+16, -5.80287e-20
domain error
1.96963e-29, 3.22384e-25, 2.92187e+23, -3.80643e+27, -8.2513e-38
domain error
2.00927e-29, 5.6976e-05, 1.16219e+25, -1.64129e-22, 0.00318397
domain error
7.29506e-29, 5904.94, 9.93922e+10, -19.528, -1.60795e-09
domain error
1.19698e-28, 1.66816e-22, 28472, -1.21137e-19, -5.84699e+17
domain error
1.64095e-28, 2.13421e-21, 7.8914e-15, -1.77584e-07, -1.70156e+15
domain error
2.03475e-28, 4.40987e+15, 28739.1, -9624.5, -1.29418e-12
domain error
2.73113e-28, 1.08457e+19, 4.00674e+08, -5.70043e-11, 1.092e-17
domain error
5.52633e-28, 1.45707e-17, 1.29411e-27, -1.67255e-15, -5.84881e+24
domain error
5.61278e-28, 9.22881e-12, 8.64222e-13, -5.6282e+23, -4.57782e-18
domain error
6.08465e-28, 1.32249e+26, 1.25536e-30, -1.89097e-14, -223.246
domain error
9.50943e-28, 2.49682e-18, 0.000904584, -3.1419e-12, -2.44954e+14
domain error
1.20779e-27, 35383.2, 1.35533e-15, -4.67834e-24, 3.20581e+15
domain error
2.29822e-27, 3.35258e-16, 2.60689e+08, -9.99161e-20, -5.4924e+11
domain error
3.0926e-27, 3.11839e-13, 3.37883e-23, -1.94349e+26, -3.55191e-19
domain error
3.12803e-27, 1.15118e+16, 1.52495e+10, -4.2399e+13, -3.07515e-21
domain error
4.49747e-27, 716.685, 1.69018e-23, -1.32558e-14, -9.2291e+13
domain error
4.84575e-27, 3.44028e-27, 3.42665e+09, -812.399, -2.12767e-06
domain error
5.81424e-27, 3.70845e-15, 3.69338e+11, -4.15794e+06, -2.95944e-11
domain error
6.08654e-27, 1.23742e+08, 1.09124e-26, -2.19946e+16, -4.90896e-19
domain error
7.71967e-27, 9.46115e-26, 1.24324e+25, -522800, -5.83203e-17
```

```
domain error
9.20037e-27, 207550, 2.45782e-17, -6.06901e+29, -2.88945e-31
domain error
1.75502e-26, 5.81507e+16, 8.83063e+21, -1.11214e-21, 1.57697e-11
domain error
2.29965e-26, 2.9716e-21, 1.81059e-25, -5.23972e-08, -6.23302e+18
domain error
2.32628e-26, 0.0655133, 1.62901e-21, -7.15441e-17, -9.88586e+17
domain error
3.49194e-26, 2.53343e+14, 756.217, -1.3359e+10, -1.275e-16
domain error
1.009e-25, 0.0694304, 1.20267e-14, -1.55746e-22, 2.10701e+17
domain error
3.54771e-25, 1.67999e-27, 2.3917e+24, -9.98754e+25, -1.11704e-36
domain error
6.31714e-25, 3.4594e-28, 6.37951e-24, -1.25529e-24, -9.56292e+35
domain error
6.74086e-25, 2.47169e+12, 1.32962e+23, -6.78845e+06, -3.32861e-24
domain error
1.8099e-24, 4.5215e-06, 8.66937e-11, -3.70795e-08, -1.41893e+11
domain error
2.29798e-24, 9.30454e-30, 6.56584e-17, -9890.38, -373149
domain error
2.88161e-24, 8.82377e-05, 1.57747e+21, -4.25068e-24, 2260.61
domain error
3.25991e-24, 1.92923e+29, 3.09752e-05, -1.00986e+11, -1.25485e-24
domain error
6.36705e-24, 2.8074e+22, 1.75569e-13, -1.53152e+24, -4.89823e-34
domain error
7.90772e-24, 2.11611e-30, 1.42682e-07, -0.00296297, -5.38814e+07
domain error
1.05302e-23, 4.83473e+26, 4.43149e-30, -1.56818e+13, -3.6836e-25
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For ellint_1 with compiler GNU C++ version 7.1.0 and library <cmath> and test data Elliptic Integral F: Mathworld Data

```
CAUTION: Gross error found at entry 9.
Found: -7.02862e+09 Expected 1.04181e+20 Error: 1.18973e+4932
1e+20, 0.390625, 1.04181e+20
CAUTION: Gross error found at entry 10.
Found: -9.3866e+09 Expected 1.39133e+50 Error: 1.18973e+4932
1e+50, 0.875, 1.39133e+50
```

Error Output For ellint_2 (complete) with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Elliptic Integral E: Mathworld Data

```
domain error
-1, 1
```

Error Output For ellint_2 with compiler GNU C++ version 7.1.0 and library <cmath> and test data Elliptic Integral E: Mathworld Data

```
CAUTION: Gross error found at entry 7.
Found: -6.3027e+09 Expected 9.34215e+09 Error: 1.18973e+4932
1e+10, -0.5, 9.34215e+09
CAUTION: Gross error found at entry 8.
```

```
Found: -6.48129e+09 Expected 7.08861e+19 Error: 1.18973e+4932
7.3787e+19, 0.390625, 7.08861e+19
CAUTION: Gross error found at entry 9.
Found: -5.13973e+09 Expected 7.1259e+49 Error: 1.18973e+4932
9.35361e+49, 0.878906, 7.1259e+49
```

Error Output For ellint_3 (complete) with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Complete Elliptic Integral PI: Mathworld Data

```
domain error
-4.14952e+180, 0.5, 7.71119e-91
```

Error Output For ellint_3 with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Elliptic Integral PI: Mathworld Data

```
domain error
1.125, 10, 0.25, 0.662468
domain error
1.125, 3, 0.25, -0.142697
domain error
1.00391, 21.5, 0.125, -0.535406
domain error
1, 2, 0.5, -2.87535
domain error
1, -2, 0.5, 2.87535
domain error
1, 2, 6.22302e-61, -2.18504
domain error
1, -2, 6.22302e-61, 2.18504
domain error
20, 3.14257, 0.5, 0.000975941
domain error
20, -3.14257, 0.5, -0.000975941
domain error
1.01562, 1.6958, 0.5, -27.1647
domain error
1.01562, -1.6958, 0.5, 27.1647
```

Error Output For ellint_3 (complete) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Complete Elliptic Integral PI: Random Data

```
Argument too small in ellint_rj
-87.1743, 0.126987, 0.167413
Argument too small in ellint_rj
-87.1743, 0.135477, 0.167431
Argument too small in ellint_rj
-87.1743, 0.221034, 0.167683
Argument too small in ellint_rj
-87.1743, 0.308167, 0.168078
Argument too small in ellint_rj
-87.1743, 0.632359, 0.17122
Argument too small in ellint_rj
-87.1743, 0.814724, 0.175341
Argument too small in ellint_rj
-87.1743, 0.835009, 0.176056
Argument too small in ellint_rj
-87.1743, 0.905792, 0.179501
```

```
Argument too small in ellint_rj
-87.1743, 0.913376, 0.180014
Argument too small in ellint_rj
-87.1743, 0.968868, 0.186162
Argument too small in ellint_rj
-86.3168, 0.126987, 0.168233
Argument too small in ellint_rj
-86.3168, 0.135477, 0.168252
Argument too small in ellint_rj
-86.3168, 0.221034, 0.168506
Argument too small in ellint_rj
-86.3168, 0.308167, 0.168905
Argument too small in ellint_rj
-86.3168, 0.632359, 0.172077
Argument too small in ellint_rj
-86.3168, 0.814724, 0.176237
Argument too small in ellint_rj
-86.3168, 0.835009, 0.176958
Argument too small in ellint_rj
-86.3168, 0.905792, 0.180437
Argument too small in ellint_rj
-86.3168, 0.913376, 0.180955
Argument too small in ellint_rj
-86.3168, 0.968868, 0.187163
Argument too small in ellint_rj
-77.6756, 0.126987, 0.177238
Argument too small in ellint_rj
-77.6756, 0.135477, 0.177258
Argument too small in ellint_rj
-77.6756, 0.221034, 0.17754
Argument too small in ellint_rj
-77.6756, 0.308167, 0.17798
Argument too small in ellint_rj
-77.6756, 0.632359, 0.181485
Argument too small in ellint_rj
-77.6756, 0.814724, 0.186089
Argument too small in ellint_rj
-77.6756, 0.835009, 0.186888
Argument too small in ellint_rj
-77.6756, 0.905792, 0.190742
Argument too small in ellint_rj
-77.6756, 0.913376, 0.191315
Argument too small in ellint_rj
-77.6756, 0.968868, 0.1982
Argument too small in ellint_rj
-68.8751, 0.126987, 0.188077
Argument too small in ellint_rj
-68.8751, 0.135477, 0.188099
Argument too small in ellint_rj
-68.8751, 0.221034, 0.188414
Argument too small in ellint_rj
-68.8751, 0.308167, 0.188907
Argument too small in ellint_rj
-68.8751, 0.632359, 0.192834
Argument too small in ellint_rj
-68.8751, 0.814724, 0.198
Argument too small in ellint_rj
-68.8751, 0.835009, 0.198896
```

```
Argument too small in ellint_rj
-68.8751, 0.905792, 0.203226
Argument too small in ellint_rj
-68.8751, 0.913376, 0.203871
Argument too small in ellint_rj
-68.8751, 0.968868, 0.211615
Argument too small in ellint_rj
-36.1317, 0.126987, 0.258074
Argument too small in ellint_rj
-36.1317, 0.135477, 0.258115
Argument too small in ellint_rj
-36.1317, 0.221034, 0.258686
Argument too small in ellint_rj
-36.1317, 0.308167, 0.259579
Argument too small in ellint_rj
-36.1317, 0.632359, 0.266738
Argument too small in ellint_rj
-36.1317, 0.814724, 0.276242
Argument too small in ellint_rj
-36.1317, 0.835009, 0.2779
Argument too small in ellint_rj
-36.1317, 0.905792, 0.285938
Argument too small in ellint_rj
-36.1317, 0.913376, 0.287139
Argument too small in ellint_rj
-36.1317, 0.968868, 0.301608
Argument too small in ellint_rj
-17.7129, 0.126987, 0.363673
Argument too small in ellint_rj
-17.7129, 0.135477, 0.36375
Argument too small in ellint_rj
-17.7129, 0.221034, 0.364822
Argument too small in ellint_rj
-17.7129, 0.308167, 0.366503
Argument too small in ellint_rj
-17.7129, 0.632359, 0.380066
Argument too small in ellint_rj
-17.7129, 0.814724, 0.398311
Argument too small in ellint_rj
-17.7129, 0.835009, 0.401518
Argument too small in ellint_rj
-17.7129, 0.905792, 0.417145
Argument too small in ellint_rj
-17.7129, 0.913376, 0.41949
Argument too small in ellint_rj
-17.7129, 0.968868, 0.447893
Argument too small in ellint_rj
-15.6641, 0.126987, 0.385409
Argument too small in ellint_rj
-15.6641, 0.135477, 0.385495
Argument too small in ellint_rj
-15.6641, 0.221034, 0.386686
Argument too small in ellint_rj
-15.6641, 0.308167, 0.388553
Argument too small in ellint_rj
-15.6641, 0.632359, 0.403643
Argument too small in ellint_rj
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For ellint_3 (complete) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Complete Elliptic Integral PI: Mathworld Data

```
CAUTION: Gross error found at entry 3.
Found: 1.28255 Expected 2.22144 Error: 0.732051
0.5, 0, 2.22144
Argument too small in ellint_rj
-4, 0.3, 0.712709
Argument too small in ellint_rj
-100000, -0.5, 0.00496945
Argument too small in ellint_rj
-1e+10, -0.75, 1.5708e-05
CAUTION: Gross error found at entry 8.
Found: 1.45615 Expected 101.045 Error: 68.3919
0.999023, -0.875, 101.045
Argument too small in ellint_rj
-4.14952e+180, 0.5, 7.71119e-91
```

Error Output For ellint_3 with compiler GNU C++ version 7.1.0 and library <cmath> and test data Elliptic Integral PI: Large Random Data

```
Argument too small in ellint_rj
-88.2952, -8.04919, 0.814724, -0.874724
Argument too small in ellint_rj
-88.2952, -7.46026, 0.135477, -0.827189
Argument too small in ellint_rj
-88.2952, -7.29046, 0.905792, -0.877476
Argument too small in ellint_rj
-88.2952, -6.23236, 0.835009, -0.652152
Argument too small in ellint_rj
-88.2952, -5.57932, 0.126987, -0.512276
Argument too small in ellint_rj
-88.2952, -4.43004, 0.968868, -0.543324
Argument too small in ellint_rj
-88.2952, -3.83666, 0.913376, -0.513389
Argument too small in ellint_rj
-88.2952, 0.93763, 0.221034, 0.158243
Argument too small in ellint_rj
-88.2952, 0.944412, 0.632359, 0.160101
Argument too small in ellint_rj
-88.2952, 2.64719, 0.308167, 0.188127
Argument too small in ellint_rj
-88.2952, 6.29447, 0.0975404, 0.676465
Argument too small in ellint_rj
-88.2952, 6.70017, 0.547221, 0.817785
Argument too small in ellint_rj
-88.2952, 8.11584, 0.278498, 0.837452
Argument too small in ellint_rj
-88.2952, 8.26752, 0.188382, 0.837571
Argument too small in ellint_rj
-88.2952, 9.15014, 0.546881, 0.885365
Argument too small in ellint_rj
-88.2952, 9.29777, 0.992881, 1.06701
Argument too small in ellint_rj
-88.2952, 9.3539, 0.957507, 1.03573
Argument too small in ellint_rj
-88.2952, 9.37736, 0.996461, 1.13933
```

```
Argument too small in ellint_rj
-88.2952, 9.85763, 0.964889, 1.24906
Argument too small in ellint_rj
-88.2952, 9.92923, 0.967695, 1.25621
Argument too small in ellint_rj
-86.8166, -8.04919, 0.157613, -0.841405
Argument too small in ellint_rj
-86.8166, -7.46026, 0.725839, -0.859877
Argument too small in ellint_rj
-86.8166, -7.29046, 0.970593, -0.914439
Argument too small in ellint_rj
-86.8166, -6.23236, 0.98111, -0.710627
Argument too small in ellint_rj
-86.8166, -5.57932, 0.957167, -0.58106
Argument too small in ellint_rj
-86.8166, -4.43004, 0.109862, -0.499839
Argument too small in ellint_rj
-86.8166, -3.83666, 0.485376, -0.494286
Argument too small in ellint_rj
-86.8166, 0.93763, 0.798106, 0.162644
Argument too small in ellint_rj
-86.8166, 0.944412, 0.80028, 0.16282
Argument too small in ellint_rj
-86.8166, 2.64719, 0.297029, 0.18978
Argument too small in ellint_rj
-86.8166, 6.29447, 0.141886, 0.682392
Argument too small in ellint_rj
-86.8166, 6.70017, 0.00478348, 0.812885
Argument too small in ellint_rj
-86.8166, 8.11584, 0.421761, 0.849249
Argument too small in ellint_rj
-86.8166, 8.26752, 0.112465, 0.843648
Argument too small in ellint_rj
-86.8166, 9.15014, 0.915736, 0.953733
Argument too small in ellint_rj
-86.8166, 9.29777, 0.639763, 0.936743
Argument too small in ellint_rj
-86.8166, 9.3539, 0.792207, 0.987359
Argument too small in ellint_rj
-86.8166, 9.37736, 0.878431, 1.02525
Argument too small in ellint_rj
-86.8166, 9.85763, 0.959492, 1.25508
Argument too small in ellint_rj
-86.8166, 9.92923, 0.503663, 1.16735
Argument too small in ellint_rj
-84.7616, -8.04919, 0.655741, -0.873305
Argument too small in ellint_rj
-84.7616, -7.46026, 0.797929, -0.879044
Argument too small in ellint_rj
-84.7616, -7.29046, 0.0357117, -0.840785
Argument too small in ellint_rj
-84.7616, -6.23236, 0.361294, -0.635502
Argument too small in ellint_rj
-84.7616, -5.57932, 0.849129, -0.558231
Argument too small in ellint_rj
-84.7616, -4.43004, 0.211924, -0.506533
Argument too small in ellint_rj
-84.7616, -3.83666, 0.933993, -0.527681
```

```
Argument too small in ellint_rj
-84.7616, 0.93763, 0.68136, 0.163458
Argument too small in ellint_rj
-84.7616, 0.944412, 0.678735, 0.163582
Argument too small in ellint_rj
-84.7616, 2.64719, 0.398739, 0.193458
Argument too small in ellint_rj
-84.7616, 6.29447, 0.75774, 0.716086
Argument too small in ellint_rj
-84.7616, 6.70017, 0.740647, 0.847849
Argument too small in ellint_rj
-84.7616, 8.11584, 0.743132, 0.883827
Argument too small in ellint_rj
-84.7616, 8.26752, 0.474759, 0.864181
Argument too small in ellint_rj
-84.7616, 9.15014, 0.392227, 0.895646
Argument too small in ellint_rj
-84.7616, 9.29777, 0.422088, 0.933423
Argument too small in ellint_rj
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For ellint_3 with compiler GNU C++ version 7.1.0 and library <cmath> and test data Elliptic Integral PI: Random Data

```
CAUTION: Gross error found at entry 150.
Found: 1.09748 Expected 1.76311 Error: 0.606506
0.546881, 1.27977, 0.349984, 1.76311
CAUTION: Gross error found at entry 151.
Found: 1.39529 Expected 2.4686 Error: 0.769232
0.546881, 1.31163, 0.907365, 2.4686
CAUTION: Gross error found at entry 152.
Found: 1.17627 Expected 2.03097 Error: 0.726615
0.546881, 1.42281, 0.196595, 2.03097
CAUTION: Gross error found at entry 153.
Found: 1.47192 Expected 2.76894 Error: 0.881179
0.546881, 1.43473, 0.848468, 2.76894
CAUTION: Gross error found at entry 154.
Found: 1.23674 Expected 2.22733 Error: 0.800966
0.546881, 1.50405, 0.251084, 2.22733
CAUTION: Gross error found at entry 155.
Found: 1.87704 Expected 3.98415 Error: 1.12257
0.546881, 1.51564, 0.955018, 3.98415
CAUTION: Gross error found at entry 156.
Found: 1.35817 Expected 2.53989 Error: 0.870091
0.546881, 1.52005, 0.616045, 2.53989
CAUTION: Gross error found at entry 157.
Found: 1.48427 Expected 2.87082 Error: 0.934166
0.546881, 1.52189, 0.778898, 2.87082
CAUTION: Gross error found at entry 158.
Found: 1.32687 Expected 2.48679 Error: 0.874176
0.546881, 1.55961, 0.473289, 2.48679
CAUTION: Gross error found at entry 159.
Found: 2.37485 Expected 5.58805 Error: 1.35301
0.546881, 1.56524, 0.98746, 5.58805
CAUTION: Gross error found at entry 170.
Found: 1.08889 Expected 1.74565 Error: 0.603142
0.547221, 1.27977, 0.285839, 1.74565
CAUTION: Gross error found at entry 171.
```

```
Found: 1.21346 Expected 2.03956 Error: 0.680778
0.547221, 1.31163, 0.67982, 2.03956
CAUTION: Gross error found at entry 172.
Found: 1.36407 Expected 2.48392 Error: 0.820965
0.547221, 1.42281, 0.7572, 2.48392
CAUTION: Gross error found at entry 173.
Found: 1.21442 Expected 2.12881 Error: 0.752947
0.547221, 1.43473, 0.39232, 2.12881
CAUTION: Gross error found at entry 174.
Found: 1.4409 Expected 2.74399 Error: 0.904352
0.547221, 1.50405, 0.753729, 2.74399
CAUTION: Gross error found at entry 175.
Found: 1.32796 Expected 2.46156 Error: 0.853642
0.547221, 1.51564, 0.561557, 2.46156
CAUTION: Gross error found at entry 176.
Found: 1.27163 Expected 2.32413 Error: 0.82767
0.547221, 1.52005, 0.380446, 2.32413
CAUTION: Gross error found at entry 177.
Found: 1.24298 Expected 2.25511 Error: 0.814274
0.547221, 1.52189, 0.208068, 2.25511
CAUTION: Gross error found at entry 178.
Found: 1.36528 Expected 2.58635 Error: 0.894379
0.547221, 1.55961, 0.567822, 2.58635
CAUTION: Gross error found at entry 179.
Found: 1.35151 Expected 2.55463 Error: 0.890206
0.547221, 1.56524, 0.527371, 2.55463
CAUTION: Gross error found at entry 189.
Found: 1.01047 Expected 1.52344 Error: 0.507658
0.632359, 0.993308, 0.964966, 1.52344
CAUTION: Gross error found at entry 190.
Found: 1.05231 Expected 1.84135 Error: 0.749817
0.632359, 1.27977, 0.129906, 1.84135
CAUTION: Gross error found at entry 191.
Found: 1.07393 Expected 1.92224 Error: 0.789918
0.632359, 1.31163, 0.154438, 1.92224
CAUTION: Gross error found at entry 192.
Found: 1.22616 Expected 2.43657 Error: 0.987156
0.632359, 1.42281, 0.568824, 2.43657
CAUTION: Gross error found at entry 193.
Found: 1.18462 Expected 2.33142 Error: 0.968083
0.632359, 1.43473, 0.394908, 2.33142
CAUTION: Gross error found at entry 194.
Found: 1.25094 Expected 2.59169 Error: 1.0718
0.632359, 1.50405, 0.469391, 2.59169
CAUTION: Gross error found at entry 195.
Found: 1.23693 Expected 2.56158 Error: 1.07091
0.632359, 1.51564, 0.387296, 2.56158
CAUTION: Gross error found at entry 196.
Found: 1.19839 Expected 2.45293 Error: 1.04685
0.632359, 1.52005, 0.0119021, 2.45293
CAUTION: Gross error found at entry 197.
Found: 1.39415 Expected 3.05228 Error: 1.18935
0.632359, 1.52189, 0.726955, 3.05228
CAUTION: Gross error found at entry 198.
Found: 1.25489 Expected 2.6569 Error: 1.11723
0.632359, 1.55961, 0.337123, 2.6569
CAUTION: Gross error found at entry 199.
Found: 1.27021 Expected 2.70857 Error: 1.13237
```

```
0.632359, 1.56524, 0.38857, 2.70857
CAUTION: Gross error found at entry 209.
Found: 0.83304 Expected 1.35947 Error: 0.631944
0.814724, 0.993308, 0.119547, 1.35947
CAUTION: Gross error found at entry 210.
Found: 1.07764 Expected 2.50291 Error: 1.32258
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For ellint_3 with compiler GNU C++ version 7.1.0 and library <cmath> and test data Elliptic Integral PI: Mathworld Data

```
CAUTION: Gross error found at entry 0.
Found: -0.809353 Expected -1.55741 Error: 0.924263
1, -1, 0, -1.55741
CAUTION: Gross error found at entry 11.
Found: 1.07555 Expected 13.2822 Error: 11.3492
0.999023, 1.5, 0, 13.2822
CAUTION: Gross error found at entry 13.
Found: -5.86896e+09 Expected 1.53659e+10 Error: 1.18973e+4932
0.5, 1e+10, 0.5, 1.53659e+10
Argument too small in ellint_rj
-100000, 10, 0.75, 0.0347926
Argument too small in ellint_rj
-1e+10, 10, 0.875, 0.000109956
Argument too small in ellint_rj
-1e+10, 1e+20, 0.875, 1.00001e+15
Argument too small in ellint_rj
-1e+10, 1.57031, 0.875, 1.57081e-05
CAUTION: Gross error found at entry 18.
Found: -6.25413e+09 Expected 6.43274e+21 Error: 1.18973e+4932
0.999023, 1e+20, 0.875, 6.43274e+21
CAUTION: Gross error found at entry 19.
Found: 0.102424 Expected 0.196321 Error: 0.916748
50, 0.125, 0.25, 0.196321
CAUTION: Gross error found at entry 20.
Found: 0.798807 Expected 1.773 Error: 1.21956
1.125, 1, 0.25, 1.773
CAUTION: Gross error found at entry 21.
Found: 7.07138 Expected 0.662468 Error: 9.6743
1.125, 10, 0.25, 0.662468
CAUTION: Gross error found at entry 22.
Found: 2.04288 Expected -0.142697 Error: 1.18973e+4932
1.125, 3, 0.25, -0.142697
CAUTION: Gross error found at entry 23.
Found: 1.07762 Expected 22.2699 Error: 19.6659
1.00391, 1.5, 0.125, 22.2699
CAUTION: Gross error found at entry 24.
Found: 15.1275 Expected -0.535406 Error: 1.18973e+4932
1.00391, 21.5, 0.125, -0.535406
CAUTION: Gross error found at entry 41.
Found: 1.57454 Expected 3.0338 Error: 0.926787
0.5, 2, 0, 3.0338
CAUTION: Gross error found at entry 42.
Found: 3.0338 Expected 1.57454 Error: 0.926787
-0.5, 2, 0, 1.57454
CAUTION: Gross error found at entry 43.
Found: -1.57454 Expected -3.0338 Error: 0.926787
0.5, -2, 0, -3.0338
```

```
CAUTION: Gross error found at entry 44.
Found: -3.0338 Expected -1.57454 Error: 0.926787
-0.5, -2, 0, -1.57454
CAUTION: Found non-finite result, when a finite value was expected at entry 51
Found: inf Expected -2.87535 Error: 1.18973e+4932
1, 2, 0.5, -2.87535
CAUTION: Gross error found at entry 51.
Found: inf Expected -2.87535 Error: 1.18973e+4932
1, 2, 0.5, -2.87535
CAUTION: Found non-finite result, when a finite value was expected at entry 52
Found: -inf Expected 2.87535 Error: 1.18973e+4932
1, -2, 0.5, 2.87535
CAUTION: Gross error found at entry 52.
Found: -inf Expected 2.87535 Error: 1.18973e+4932
1, -2, 0.5, 2.87535
CAUTION: Found non-finite result, when a finite value was expected at entry 53
Found: inf Expected -2.18504 Error: 1.18973e+4932
1, 2, 6.22302e-61, -2.18504
CAUTION: Gross error found at entry 53.
Found: inf Expected -2.18504 Error: 1.18973e+4932
1, 2, 6.22302e-61, -2.18504
CAUTION: Found non-finite result, when a finite value was expected at entry 54
Found: -inf Expected 2.18504 Error: 1.18973e+4932
1, -2, 6.22302e-61, 2.18504
CAUTION: Gross error found at entry 54.
Found: -inf Expected 2.18504 Error: 1.18973e+4932
1, -2, 6.22302e-61, 2.18504
CAUTION: Gross error found at entry 57.
Found: 0.703907 Expected 0.000975941 Error: 720.259
20, 3.14257, 0.5, 0.000975941
CAUTION: Gross error found at entry 58.
Found: -0.703907 Expected -0.000975941 Error: 720.259
20, -3.14257, 0.5, -0.000975941
CAUTION: Gross error found at entry 59.
Found: 1.24445 Expected -27.1647 Error: 1.18973e+4932
1.01562, 1.6958, 0.5, -27.1647
CAUTION: Gross error found at entry 60.
Found: -1.24445 Expected 27.1647 Error: 1.18973e+4932
1.01562, -1.6958, 0.5, 27.1647
```

Error Output For expint (Ei) with compiler GNU C++ version 7.1.0 and library <cmath> and test data Exponential Integral Ei

```
Continued fraction failed in __expint_En_cont_frac.
-1.30539, -0.134326
```

Error Output For ibeta with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Incomplete Beta Function: Large and Diverse Values

```
underflow
1.04761e-05, 39078.2, 0.913384, 95444.4, 0, 1, 0
underflow
1.2158e-05, 24110.5, 0.135563, 82239.7, 0, 1, 0
underflow
1.30342e-05, 26168.3, 0.127074, 76710.7, 0, 1, 0
underflow
1.51962e-05, 16177.5, 0.814742, 65795.4, 0, 1, 0
underflow
```

```
1.64873e-05, 470997, 0.127074, 60639.1, 0, 1, 0
underflow
1.66259e-05, 147819, 0.632396, 60134.5, 0, 1, 0
underflow
1.78638e-05, 439.387, 0.835025, 55972.4, 0, 1, 0
underflow
2.00434e-05, 482.007, 0.905801, 49885.1, 0, 1, 0
underflow
2.05189e-05, 236088, 0.835025, 48722.7, 0, 1, 0
underflow
2.14336e-05, 3719.28, 0.814742, 46647, 0, 1, 0
underflow
2.24486e-05, 445071, 0.221112, 44532.6, 0, 1, 0
underflow
2.34849e-05, 25542.8, 0.968871, 42569.8, 0, 1, 0
underflow
2.39993e-05, 462.946, 0.814742, 41661.1, 0, 1, 0
underflow
2.52178e-05, 1832.27, 0.913384, 39646.4, 0, 1, 0
underflow
2.87756e-05, 25491.8, 0.905801, 34740.9, 0, 1, 0
underflow
2.89316e-05, 494.984, 0.968871, 34557.6, 0, 1, 0
underflow
3.11413e-05, 348144, 0.308236, 32098.3, 0, 1, 0
underflow
3.12319e-05, 33713, 0.221112, 32007.5, 0, 1, 0
underflow
3.19889e-05, 3931.19, 0.308236, 31251.9, 0, 1, 0
underflow
3.27129e-05, 3109.49, 0.968871, 30560.4, 0, 1, 0
underflow
3.27529e-05, 25796.3, 0.835025, 30520.9, 0, 1, 0
underflow
3.34106e-05, 3378.01, 0.221112, 29922, 0, 1, 0
underflow
3.40793e-05, 288783, 0.814742, 29330.2, 0, 1, 0
underflow
3.46418e-05, 411.559, 0.913384, 28860.3, 0, 1, 0
underflow
3.61632e-05, 311937, 0.905801, 27639.2, 0, 1, 0
underflow
3.75686e-05, 386440, 0.913384, 26604.5, 0, 1, 0
underflow
3.99261e-05, 495352, 0.968871, 25032.6, 0, 1, 0
underflow
4.01492e-05, 3246.23, 0.905801, 24898.5, 0, 1, 0
underflow
4.0288e-05, 2569.28, 0.835025, 24812.9, 0, 1, 0
underflow
4.11667e-05, 24253.8, 0.308236, 24280.8, 0, 1, 0
underflow
4.17714e-05, 274447, 0.135563, 23926.7, 0, 1, 0
underflow
4.66877e-05, 3780.93, 0.632396, 21410.1, 0, 1, 0
underflow
4.73604e-05, 48598.7, 0.632396, 21103.3, 0, 1, 0
underflow
```

```
0.00013245, 251.768, 0.968871, 7543.9, 0, 1, 0
underflow
0.000168283, 195801, 0.905801, 5929.61, 0, 1, 0
underflow
0.000177906, 276489, 0.814742, 5607.86, 0, 1, 0
underflow
0.000183097, 316055, 0.127074, 5448.36, 0, 1, 0
underflow
0.000190369, 159132, 0.835025, 5240.42, 0, 1, 0
underflow
0.000191066, 419861, 0.913384, 5220.29, 0, 1, 0
underflow
0.000192195, 177798, 0.308236, 5190.39, 0, 1, 0
underflow
0.000220499, 107380, 0.135563, 4523.03, 0, 1, 0
underflow
0.00022254, 1432.25, 0.814742, 4485.74, 0, 1, 0
underflow
0.000240291, 49604.4, 0.632396, 4150.25, 0, 1, 0
underflow
0.000251444, 15605.8, 0.135563, 3966.81, 0, 1, 0
underflow
0.000274279, 289206, 0.968871, 3632.79, 0, 1, 0
underflow
0.000274343, 2954.47, 0.308236, 3636.51, 0, 1, 0
underflow
0.000278714, 4023.16, 0.632396, 3579.05, 0, 1, 0
underflow
0.000288369, 460073, 0.221112, 3454.19, 0, 1, 0
underflow
0.000294717, 4642.26, 0.221112, 3384.08, 0, 1, 0
underflow
0.000303403, 2574.36, 0.835025, 3287.52, 0, 1, 0
underflow
0.000304309, 4480.75, 0.905801, 3277.17, 0, 1, 0
underflow
0.00031313, 47957, 0.308236, 3182.22, 0, 1, 0
underflow
0.000320063, 25544.6, 0.905801, 3113.68, 0, 1, 0
underflow
0.000334818, 29065.5, 0.968871, 2975.86, 0, 1, 0
underflow
0.00034899, 41187.6, 0.913384, 2854.23, 0, 1, 0
underflow
0.000350247, 426.308, 0.905801, 2848.5, 0, 1, 0
underflow
0.000357727, 31752.2, 0.127074, 2784.5, 0, 1, 0
underflow
0.000412091, 367.714, 0.913384, 2420.17, 0, 1, 0
underflow
0.000417933, 4668.47, 0.968871, 2383.72, 0, 1, 0
underflow
0.000424632, 17994.9, 0.221112, 2344.63, 0, 1, 0
underflow
0.000427051, 2443.44, 0.913384, 2333.28, 0, 1, 0
underflow
0.000437724, 454399, 0.632396, 2270.98, 0, 1, 0
underflow
```

```
0.000450377, 10660.8, 0.835025, 2210.53, 0, 1, 0
underflow
0.000475601, 19603, 0.814742, 2092.17, 0, 1, 0
underflow
0.00116972, 4487.22, 0.221112, 845.964, 0, 1, 0
underflow
0.00124188, 211066, 0.632396, 792.493, 0, 1, 0
underflow
0.00128578, 4738.41, 0.308236, 768.75, 0, 1, 0
underflow
0.00133388, 46277.8, 0.913384, 738.46, 0, 1, 0
underflow
0.00138692, 2158.76, 0.814742, 712.816, 0, 1, 0
underflow
0.00153268, 13060.2, 0.968871, 642.474, 0, 1, 0
underflow
0.00159946, 1780.43, 0.968871, 617.202, 0, 1, 0
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For jacobi_dn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Modulus near 1

```
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357365, 0.0357231
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357497, 0.0357097
|m| > 1.0
-4.0246, 1, -0.99936, 0.0357762, 0.0356829
|m| > 1.0
-4.0246, 1, -0.999359, 0.0357895, 0.0356695
|m| > 1.0
-4.0246, 1.00001, -0.999354, 0.0359354, 0.0355222
|m| > 1.0
-4.0246, 1.00003, -0.999347, 0.0361343, 0.0353212
|m| > 1.0
-4.0246, 1.00004, -0.999343, 0.036247, 0.0352073
|m| > 1.0
-4.0246, 1.0001, -0.999311, 0.0371157, 0.0343296
|m| > 1.0
-4.0246, 1.00016, -0.99928, 0.0379513, 0.0334851
|m| > 1.0
-4.0246, 1.00027, -0.999221, 0.0394571, 0.0319634
|m| > 1.0
-4.0246, 1.00076, -0.99893, 0.0462407, 0.0251046
|m| > 1.0
-4.0246, 1.00125, -0.998589, 0.0531109, 0.0181532
|m| > 1.0
-4.0246, 1.00232, -0.99768, 0.0680761, 0.0029944
|m| > 1.0
-4.0246, 1.00604, -0.992752, 0.120179, -0.049966
|m| > 1.0
-4.0246, 1.01557, -0.967356, 0.25342, -0.186698
|m| > 1.0
-4.0246, 1.03059, -0.890373, 0.455232, -0.397492
|m| > 1.0
-4.0246, 1.06239, -0.607191, 0.794556, -0.76412
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479567, 0.0479467
```

```
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479665, 0.0479367
|m| > 1.0
-3.73013, 1, -0.998848, 0.0479862, 0.0479167
|m| > 1.0
-3.73013, 1, -0.998848, 0.047996, 0.0479067
|m| > 1.0
-3.73013, 1.00001, -0.998842, 0.0481042, 0.0477966
|m| > 1.0
-3.73013, 1.00003, -0.998835, 0.0482517, 0.0476465
|m| > 1.0
-3.73013, 1.00004, -0.998831, 0.0483354, 0.0475615
|m| > 1.0
-3.73013, 1.0001, -0.9988, 0.0489797, 0.0469059
|m| > 1.0
-3.73013, 1.00016, -0.998769, 0.0495995, 0.0462752
|m| > 1.0
-3.73013, 1.00027, -0.998713, 0.0507164, 0.0451386
|m| > 1.0
-3.73013, 1.00076, -0.998445, 0.0557477, 0.0400164
|m| > 1.0
-3.73013, 1.00125, -0.998147, 0.0608429, 0.0348257
|m| > 1.0
-3.73013, 1.00232, -0.997409, 0.0719406, 0.0235071
|m| > 1.0
-3.73013, 1.00604, -0.993866, 0.110593, -0.016048
|m| > 1.0
-3.73013, 1.01557, -0.977708, 0.209971, -0.118704
|m| > 1.0
-3.73013, 1.03059, -0.931162, 0.364606, -0.281224
|m| > 1.0
-3.73013, 1.06239, -0.753495, 0.657453, -0.599326
|m| > 1.0
-3.64523, 1, -0.998637, 0.0521997, 0.0521906
|m| > 1.0
-3.64523, 1, -0.998636, 0.0522087, 0.0521814
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522268, 0.052163
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522358, 0.0521538
|m| > 1.0
-3.64523, 1.00001, -0.99863, 0.052335, 0.0520526
|m| > 1.0
-3.64523, 1.00003, -0.998622, 0.0524703, 0.0519145
|m| > 1.0
-3.64523, 1.00004, -0.998618, 0.052547, 0.0518363
|m| > 1.0
-3.64523, 1.0001, -0.998587, 0.0531379, 0.0512335
|m| > 1.0
-3.64523, 1.00016, -0.998557, 0.0537063, 0.0506536
|m| > 1.0
-3.64523, 1.00027, -0.998501, 0.0547305, 0.0496084
|m| > 1.0
-3.64523, 1.00076, -0.998238, 0.0593443, 0.0448986
|m| > 1.0
-3.64523, 1.00125, -0.997949, 0.0640165, 0.0401258
|m| > 1.0
-3.64523, 1.00232, -0.997244, 0.0741927, 0.0297191
```

```
|m| > 1.0
-3.64523, 1.00604, -0.993972, 0.109636, -0.00664888
|m| > 1.0
-3.64523, 1.01557, -0.979623, 0.200844, -0.101111
|m| > 1.0
-3.64523, 1.03059, -0.939163, 0.343472, -0.251382
|m| > 1.0
-3.64523, 1.06239, -0.784719, 0.619852, -0.552253
|m| > 1.0
-3.11618, 1, -0.996078, 0.0884811, 0.0884757
|m| > 1.0
-3.11618, 1, -0.996077, 0.0884863, 0.0884702
|m| > 1.0
-3.11618, 1, -0.996076, 0.0884967, 0.0884593
|m| > 1.0
-3.11618, 1, -0.996076, 0.0885019, 0.0884538
|m| > 1.0
-3.11618, 1.00001, -0.996071, 0.0885593, 0.0883936
|m| > 1.0
-3.11618, 1.00003, -0.996064, 0.0886376, 0.0883114
|m| > 1.0
-3.11618, 1.00004, -0.99606, 0.0886819, 0.0882648
|m| > 1.0
-3.11618, 1.0001, -0.99603, 0.0890236, 0.0879059
|m| > 1.0
-3.11618, 1.00016, -0.996, 0.0893523, 0.0875607
|m| > 1.0
-3.11618, 1.00027, -0.995947, 0.0899445, 0.0869386
|m| > 1.0
-3.11618, 1.00076, -0.995702, 0.092612, 0.0841353
|m| > 1.0
-3.11618, 1.00125, -0.995447, 0.0953126, 0.0812953
|m| > 1.0
-3.11618, 1.00232, -0.994867, 0.101193, 0.0751049
|m| > 1.0
-3.11618, 1.00604, -0.992571, 0.121667, 0.0534858
|m| > 1.0
-3.11618, 1.01557, -0.984666, 0.174451, -0.00273723
|m| > 1.0
-3.11618, 1.03059, -0.966077, 0.258253, -0.0934336
|m| > 1.0
-3.11618, 1.06239, -0.901067, 0.433681, -0.289151
|m| > 1.0
-2.78966, 1, -0.992478, 0.122424, 0.12242
|m| > 1.0
-2.78966, 1, -0.992477, 0.122428, 0.122416
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For jacobi_cn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Modulus near 1

```
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357365, 0.0357231
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357497, 0.0357097
|m| > 1.0
-4.0246, 1, -0.99936, 0.0357762, 0.0356829
|m| > 1.0
```

```
-4.0246, 1, -0.999359, 0.0357895, 0.0356695
|m| > 1.0
-4.0246, 1.00001, -0.999354, 0.0359354, 0.0355222
|m| > 1.0
-4.0246, 1.00003, -0.999347, 0.0361343, 0.0353212
|m| > 1.0
-4.0246, 1.00004, -0.999343, 0.036247, 0.0352073
|m| > 1.0
-4.0246, 1.0001, -0.999311, 0.0371157, 0.0343296
|m| > 1.0
-4.0246, 1.00016, -0.99928, 0.0379513, 0.0334851
|m| > 1.0
-4.0246, 1.00027, -0.999221, 0.0394571, 0.0319634
|m| > 1.0
-4.0246, 1.00076, -0.99893, 0.0462407, 0.0251046
|m| > 1.0
-4.0246, 1.00125, -0.998589, 0.0531109, 0.0181532
|m| > 1.0
-4.0246, 1.00232, -0.99768, 0.0680761, 0.0029944
|m| > 1.0
-4.0246, 1.00604, -0.992752, 0.120179, -0.049966
|m| > 1.0
-4.0246, 1.01557, -0.967356, 0.25342, -0.186698
|m| > 1.0
-4.0246, 1.03059, -0.890373, 0.455232, -0.397492
|m| > 1.0
-4.0246, 1.06239, -0.607191, 0.794556, -0.76412
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479567, 0.0479467
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479665, 0.0479367
|m| > 1.0
-3.73013, 1, -0.998848, 0.0479862, 0.0479167
|m| > 1.0
-3.73013, 1, -0.998848, 0.047996, 0.0479067
|m| > 1.0
-3.73013, 1.00001, -0.998842, 0.0481042, 0.0477966
|m| > 1.0
-3.73013, 1.00003, -0.998835, 0.0482517, 0.0476465
|m| > 1.0
-3.73013, 1.00004, -0.998831, 0.0483354, 0.0475615
|m| > 1.0
-3.73013, 1.0001, -0.9988, 0.0489797, 0.0469059
|m| > 1.0
-3.73013, 1.00016, -0.998769, 0.0495995, 0.0462752
|m| > 1.0
-3.73013, 1.00027, -0.998713, 0.0507164, 0.0451386
|m| > 1.0
-3.73013, 1.00076, -0.998445, 0.0557477, 0.0400164
|m| > 1.0
-3.73013, 1.00125, -0.998147, 0.0608429, 0.0348257
|m| > 1.0
-3.73013, 1.00232, -0.997409, 0.0719406, 0.0235071
|m| > 1.0
-3.73013, 1.00604, -0.993866, 0.110593, -0.016048
|m| > 1.0
-3.73013, 1.01557, -0.977708, 0.209971, -0.118704
|m| > 1.0
```

```
-3.73013, 1.03059, -0.931162, 0.364606, -0.281224
|m| > 1.0
-3.73013, 1.06239, -0.753495, 0.657453, -0.599326
|m| > 1.0
-3.64523, 1, -0.998637, 0.0521997, 0.0521906
|m| > 1.0
-3.64523, 1, -0.998636, 0.0522087, 0.0521814
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522268, 0.052163
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522358, 0.0521538
|m| > 1.0
-3.64523, 1.00001, -0.99863, 0.052335, 0.0520526
|m| > 1.0
-3.64523, 1.00003, -0.998622, 0.0524703, 0.0519145
|m| > 1.0
-3.64523, 1.00004, -0.998618, 0.052547, 0.0518363
|m| > 1.0
-3.64523, 1.0001, -0.998587, 0.0531379, 0.0512335
|m| > 1.0
-3.64523, 1.00016, -0.998557, 0.0537063, 0.0506536
|m| > 1.0
-3.64523, 1.00027, -0.998501, 0.0547305, 0.0496084
|m| > 1.0
-3.64523, 1.00076, -0.998238, 0.0593443, 0.0448986
|m| > 1.0
-3.64523, 1.00125, -0.997949, 0.0640165, 0.0401258
|m| > 1.0
-3.64523, 1.00232, -0.997244, 0.0741927, 0.0297191
|m| > 1.0
-3.64523, 1.00604, -0.993972, 0.109636, -0.00664888
|m| > 1.0
-3.64523, 1.01557, -0.979623, 0.200844, -0.101111
|m| > 1.0
-3.64523, 1.03059, -0.939163, 0.343472, -0.251382
|m| > 1.0
-3.64523, 1.06239, -0.784719, 0.619852, -0.552253
|m| > 1.0
-3.11618, 1, -0.996078, 0.0884811, 0.0884757
|m| > 1.0
-3.11618, 1, -0.996077, 0.0884863, 0.0884702
|m| > 1.0
-3.11618, 1, -0.996076, 0.0884967, 0.0884593
|m| > 1.0
-3.11618, 1, -0.996076, 0.0885019, 0.0884538
|m| > 1.0
-3.11618, 1.00001, -0.996071, 0.0885593, 0.0883936
|m| > 1.0
-3.11618, 1.00003, -0.996064, 0.0886376, 0.0883114
|m| > 1.0
-3.11618, 1.00004, -0.99606, 0.0886819, 0.0882648
|m| > 1.0
-3.11618, 1.0001, -0.99603, 0.0890236, 0.0879059
|m| > 1.0
-3.11618, 1.00016, -0.996, 0.0893523, 0.0875607
|m| > 1.0
-3.11618, 1.00027, -0.995947, 0.0899445, 0.0869386
|m| > 1.0
```

```
-3.11618, 1.00076, -0.995702, 0.092612, 0.0841353
|m| > 1.0
-3.11618, 1.00125, -0.995447, 0.0953126, 0.0812953
|m| > 1.0
-3.11618, 1.00232, -0.994867, 0.101193, 0.0751049
|m| > 1.0
-3.11618, 1.00604, -0.992571, 0.121667, 0.0534858
|m| > 1.0
-3.11618, 1.01557, -0.984666, 0.174451, -0.00273723
|m| > 1.0
-3.11618, 1.03059, -0.966077, 0.258253, -0.0934336
|m| > 1.0
-3.11618, 1.06239, -0.901067, 0.433681, -0.289151
|m| > 1.0
-2.78966, 1, -0.992478, 0.122424, 0.12242
|m| > 1.0
-2.78966, 1, -0.992477, 0.122428, 0.122416
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For jacobi_sn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Modulus near 1

```
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357365, 0.0357231
|m| > 1.0
-4.0246, 1, -0.999361, 0.0357497, 0.0357097
|m| > 1.0
-4.0246, 1, -0.99936, 0.0357762, 0.0356829
|m| > 1.0
-4.0246, 1, -0.999359, 0.0357895, 0.0356695
|m| > 1.0
-4.0246, 1.00001, -0.999354, 0.0359354, 0.0355222
|m| > 1.0
-4.0246, 1.00003, -0.999347, 0.0361343, 0.0353212
|m| > 1.0
-4.0246, 1.00004, -0.999343, 0.036247, 0.0352073
|m| > 1.0
-4.0246, 1.0001, -0.999311, 0.0371157, 0.0343296
|m| > 1.0
-4.0246, 1.00016, -0.99928, 0.0379513, 0.0334851
|m| > 1.0
-4.0246, 1.00027, -0.999221, 0.0394571, 0.0319634
|m| > 1.0
-4.0246, 1.00076, -0.99893, 0.0462407, 0.0251046
|m| > 1.0
-4.0246, 1.00125, -0.998589, 0.0531109, 0.0181532
|m| > 1.0
-4.0246, 1.00232, -0.99768, 0.0680761, 0.0029944
|m| > 1.0
-4.0246, 1.00604, -0.992752, 0.120179, -0.049966
|m| > 1.0
-4.0246, 1.01557, -0.967356, 0.25342, -0.186698
|m| > 1.0
-4.0246, 1.03059, -0.890373, 0.455232, -0.397492
|m| > 1.0
-4.0246, 1.06239, -0.607191, 0.794556, -0.76412
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479567, 0.0479467
```

```
|m| > 1.0
-3.73013, 1, -0.998849, 0.0479665, 0.0479367
|m| > 1.0
-3.73013, 1, -0.998848, 0.0479862, 0.0479167
|m| > 1.0
-3.73013, 1, -0.998848, 0.047996, 0.0479067
|m| > 1.0
-3.73013, 1.00001, -0.998842, 0.0481042, 0.0477966
|m| > 1.0
-3.73013, 1.00003, -0.998835, 0.0482517, 0.0476465
|m| > 1.0
-3.73013, 1.00004, -0.998831, 0.0483354, 0.0475615
|m| > 1.0
-3.73013, 1.0001, -0.9988, 0.0489797, 0.0469059
|m| > 1.0
-3.73013, 1.00016, -0.998769, 0.0495995, 0.0462752
|m| > 1.0
-3.73013, 1.00027, -0.998713, 0.0507164, 0.0451386
|m| > 1.0
-3.73013, 1.00076, -0.998445, 0.0557477, 0.0400164
|m| > 1.0
-3.73013, 1.00125, -0.998147, 0.0608429, 0.0348257
|m| > 1.0
-3.73013, 1.00232, -0.997409, 0.0719406, 0.0235071
|m| > 1.0
-3.73013, 1.00604, -0.993866, 0.110593, -0.016048
|m| > 1.0
-3.73013, 1.01557, -0.977708, 0.209971, -0.118704
|m| > 1.0
-3.73013, 1.03059, -0.931162, 0.364606, -0.281224
|m| > 1.0
-3.73013, 1.06239, -0.753495, 0.657453, -0.599326
|m| > 1.0
-3.64523, 1, -0.998637, 0.0521997, 0.0521906
|m| > 1.0
-3.64523, 1, -0.998636, 0.0522087, 0.0521814
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522268, 0.052163
|m| > 1.0
-3.64523, 1, -0.998635, 0.0522358, 0.0521538
|m| > 1.0
-3.64523, 1.00001, -0.99863, 0.052335, 0.0520526
|m| > 1.0
-3.64523, 1.00003, -0.998622, 0.0524703, 0.0519145
|m| > 1.0
-3.64523, 1.00004, -0.998618, 0.052547, 0.0518363
|m| > 1.0
-3.64523, 1.0001, -0.998587, 0.0531379, 0.0512335
|m| > 1.0
-3.64523, 1.00016, -0.998557, 0.0537063, 0.0506536
|m| > 1.0
-3.64523, 1.00027, -0.998501, 0.0547305, 0.0496084
|m| > 1.0
-3.64523, 1.00076, -0.998238, 0.0593443, 0.0448986
|m| > 1.0
-3.64523, 1.00125, -0.997949, 0.0640165, 0.0401258
|m| > 1.0
-3.64523, 1.00232, -0.997244, 0.0741927, 0.0297191
```

```
|m| > 1.0
-3.64523, 1.00604, -0.993972, 0.109636, -0.00664888
|m| > 1.0
-3.64523, 1.01557, -0.979623, 0.200844, -0.101111
|m| > 1.0
-3.64523, 1.03059, -0.939163, 0.343472, -0.251382
|m| > 1.0
-3.64523, 1.06239, -0.784719, 0.619852, -0.552253
|m| > 1.0
-3.11618, 1, -0.996078, 0.0884811, 0.0884757
|m| > 1.0
-3.11618, 1, -0.996077, 0.0884863, 0.0884702
|m| > 1.0
-3.11618, 1, -0.996076, 0.0884967, 0.0884593
|m| > 1.0
-3.11618, 1, -0.996076, 0.0885019, 0.0884538
|m| > 1.0
-3.11618, 1.00001, -0.996071, 0.0885593, 0.0883936
|m| > 1.0
-3.11618, 1.00003, -0.996064, 0.0886376, 0.0883114
|m| > 1.0
-3.11618, 1.00004, -0.99606, 0.0886819, 0.0882648
|m| > 1.0
-3.11618, 1.0001, -0.99603, 0.0890236, 0.0879059
|m| > 1.0
-3.11618, 1.00016, -0.996, 0.0893523, 0.0875607
|m| > 1.0
-3.11618, 1.00027, -0.995947, 0.0899445, 0.0869386
|m| > 1.0
-3.11618, 1.00076, -0.995702, 0.092612, 0.0841353
|m| > 1.0
-3.11618, 1.00125, -0.995447, 0.0953126, 0.0812953
|m| > 1.0
-3.11618, 1.00232, -0.994867, 0.101193, 0.0751049
|m| > 1.0
-3.11618, 1.00604, -0.992571, 0.121667, 0.0534858
|m| > 1.0
-3.11618, 1.01557, -0.984666, 0.174451, -0.00273723
|m| > 1.0
-3.11618, 1.03059, -0.966077, 0.258253, -0.0934336
|m| > 1.0
-3.11618, 1.06239, -0.901067, 0.433681, -0.289151
|m| > 1.0
-2.78966, 1, -0.992478, 0.122424, 0.12242
|m| > 1.0
-2.78966, 1, -0.992477, 0.122428, 0.122416
*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***
```

Error Output For jacobi_dn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Random Small Values

```
|m| > 1.0
1.65048e-12, 1.65574, 1.65048e-12, 1, 1
|m| > 1.0
2.06542e-12, 1.65574, 2.06542e-12, 1, 1
|m| > 1.0
6.93323e-12, 1.65574, 6.93323e-12, 1, 1
|m| > 1.0
```

```
1.33514e-11, 1.65574, 1.33514e-11, 1, 1
|m| > 1.0
1.63998e-11, 1.65574, 1.63998e-11, 1, 1
|m| > 1.0
5.73016e-11, 1.65574, 5.73016e-11, 1, 1
|m| > 1.0
1.11373e-10, 1.65574, 1.11373e-10, 1, 1
|m| > 1.0
1.42147e-10, 1.65574, 1.42147e-10, 1, 1
|m| > 1.0
3.80063e-10, 1.65574, 3.80063e-10, 1, 1
|m| > 1.0
6.09163e-10, 1.65574, 6.09163e-10, 1, 1
|m| > 1.0
1.02216e-09, 1.65574, 1.02216e-09, 1, 1
|m| > 1.0
2.88192e-09, 1.65574, 2.88192e-09, 1, 1
|m| > 1.0
4.76278e-09, 1.65574, 4.76278e-09, 1, 1
|m| > 1.0
8.85413e-09, 1.65574, 8.85413e-09, 1, 1
|m| > 1.0
2.30503e-08, 1.65574, 2.30503e-08, 1, 1
|m| > 1.0
5.93925e-08, 1.65574, 5.93925e-08, 1, 1
|m| > 1.0
1.16676e-07, 1.65574, 1.16676e-07, 1, 1
|m| > 1.0
2.37997e-07, 1.65574, 2.37997e-07, 1, 1
|m| > 1.0
4.68466e-07, 1.65574, 4.68466e-07, 1, 1
|m| > 1.0
9.3827e-07, 1.65574, 9.3827e-07, 1, 1
|m| > 1.0
1.10399e-06, 1.65574, 1.10399e-06, 1, 1
|m| > 1.0
3.29178e-06, 1.65574, 3.29178e-06, 1, 1
|m| > 1.0
7.51721e-06, 1.65574, 7.51721e-06, 1, 1
|m| > 1.0
1.51147e-05, 1.65574, 1.51147e-05, 1, 1
|m| > 1.0
2.9864e-05, 1.65574, 2.9864e-05, 1, 1
|m| > 1.0
3.38703e-05, 1.65574, 3.38703e-05, 1, 1
|m| > 1.0
9.06601e-05, 1.65574, 9.06601e-05, 1, 1
|m| > 1.0
0.000219495, 1.65574, 0.000219495, 1, 1
|m| > 1.0
0.000439522, 1.65574, 0.000439521, 1, 1
|m| > 1.0
0.000633315, 1.65574, 0.000633315, 1, 0.999999
|m| > 1.0
0.00111512, 1.65574, 0.00111512, 0.999999, 0.999998
|m| > 1.0
0.00196247, 1.65574, 0.00196246, 0.999998, 0.999995
|m| > 1.0
```

```
0.00555375, 1.65574, 0.00555365, 0.999985, 0.999958
|m| > 1.0
0.00869113, 1.65574, 0.00869072, 0.999962, 0.999896
|m| > 1.0
0.0299334, 1.65574, 0.0299166, 0.999552, 0.998772
|m| > 1.0
0.0512426, 1.65574, 0.0511588, 0.998691, 0.996406
|m| > 1.0
0.112013, 1.65574, 0.111143, 0.993804, 0.982922
|m| > 1.0
0.234804, 1.65574, 0.227, 0.973895, 0.926679
|m| > 1.0
0.489873, 1.65574, 0.425971, 0.904737, 0.708912
|m| > 1.0
0.751831, 1.65574, 0.553446, 0.832885, 0.400346
|m| > 1.0
1.65574, 1.65574, 0.408154, 0.912913, -0.737088
```

Error Output For jacobi_cn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Random Small Values

```
|m| > 1.0
1.65048e-12, 1.65574, 1.65048e-12, 1, 1
|m| > 1.0
2.06542e-12, 1.65574, 2.06542e-12, 1, 1
|m| > 1.0
6.93323e-12, 1.65574, 6.93323e-12, 1, 1
|m| > 1.0
1.33514e-11, 1.65574, 1.33514e-11, 1, 1
|m| > 1.0
1.63998e-11, 1.65574, 1.63998e-11, 1, 1
|m| > 1.0
5.73016e-11, 1.65574, 5.73016e-11, 1, 1
|m| > 1.0
1.11373e-10, 1.65574, 1.11373e-10, 1, 1
|m| > 1.0
1.42147e-10, 1.65574, 1.42147e-10, 1, 1
|m| > 1.0
3.80063e-10, 1.65574, 3.80063e-10, 1, 1
|m| > 1.0
6.09163e-10, 1.65574, 6.09163e-10, 1, 1
|m| > 1.0
1.02216e-09, 1.65574, 1.02216e-09, 1, 1
|m| > 1.0
2.88192e-09, 1.65574, 2.88192e-09, 1, 1
|m| > 1.0
4.76278e-09, 1.65574, 4.76278e-09, 1, 1
|m| > 1.0
8.85413e-09, 1.65574, 8.85413e-09, 1, 1
|m| > 1.0
2.30503e-08, 1.65574, 2.30503e-08, 1, 1
|m| > 1.0
5.93925e-08, 1.65574, 5.93925e-08, 1, 1
|m| > 1.0
1.16676e-07, 1.65574, 1.16676e-07, 1, 1
|m| > 1.0
2.37997e-07, 1.65574, 2.37997e-07, 1, 1
|m| > 1.0
```

```
4.68466e-07, 1.65574, 4.68466e-07, 1, 1
|m| > 1.0
9.3827e-07, 1.65574, 9.3827e-07, 1, 1
|m| > 1.0
1.10399e-06, 1.65574, 1.10399e-06, 1, 1
|m| > 1.0
3.29178e-06, 1.65574, 3.29178e-06, 1, 1
|m| > 1.0
7.51721e-06, 1.65574, 7.51721e-06, 1, 1
|m| > 1.0
1.51147e-05, 1.65574, 1.51147e-05, 1, 1
|m| > 1.0
2.9864e-05, 1.65574, 2.9864e-05, 1, 1
|m| > 1.0
3.38703e-05, 1.65574, 3.38703e-05, 1, 1
|m| > 1.0
9.06601e-05, 1.65574, 9.06601e-05, 1, 1
|m| > 1.0
0.000219495, 1.65574, 0.000219495, 1, 1
|m| > 1.0
0.000439522, 1.65574, 0.000439521, 1, 1
|m| > 1.0
0.000633315, 1.65574, 0.000633315, 1, 0.999999
|m| > 1.0
0.00111512, 1.65574, 0.00111512, 0.999999, 0.999998
|m| > 1.0
0.00196247, 1.65574, 0.00196246, 0.999998, 0.999995
|m| > 1.0
0.00555375, 1.65574, 0.00555365, 0.999985, 0.999958
|m| > 1.0
0.00869113, 1.65574, 0.00869072, 0.999962, 0.999896
|m| > 1.0
0.0299334, 1.65574, 0.0299166, 0.999552, 0.998772
|m| > 1.0
0.0512426, 1.65574, 0.0511588, 0.998691, 0.996406
|m| > 1.0
0.112013, 1.65574, 0.111143, 0.993804, 0.982922
|m| > 1.0
0.234804, 1.65574, 0.227, 0.973895, 0.926679
|m| > 1.0
0.489873, 1.65574, 0.425971, 0.904737, 0.708912
|m| > 1.0
0.751831, 1.65574, 0.553446, 0.832885, 0.400346
|m| > 1.0
1.65574, 1.65574, 0.408154, 0.912913, -0.737088
```

Error Output For jacobi_sn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Random Small Values

```
|m| > 1.0
1.65048e-12, 1.65574, 1.65048e-12, 1, 1
|m| > 1.0
2.06542e-12, 1.65574, 2.06542e-12, 1, 1
|m| > 1.0
6.93323e-12, 1.65574, 6.93323e-12, 1, 1
|m| > 1.0
1.33514e-11, 1.65574, 1.33514e-11, 1, 1
|m| > 1.0
```

```
1.63998e-11, 1.65574, 1.63998e-11, 1, 1
|m| > 1.0
5.73016e-11, 1.65574, 5.73016e-11, 1, 1
|m| > 1.0
1.11373e-10, 1.65574, 1.11373e-10, 1, 1
|m| > 1.0
1.42147e-10, 1.65574, 1.42147e-10, 1, 1
|m| > 1.0
3.80063e-10, 1.65574, 3.80063e-10, 1, 1
|m| > 1.0
6.09163e-10, 1.65574, 6.09163e-10, 1, 1
|m| > 1.0
1.02216e-09, 1.65574, 1.02216e-09, 1, 1
|m| > 1.0
2.88192e-09, 1.65574, 2.88192e-09, 1, 1
|m| > 1.0
4.76278e-09, 1.65574, 4.76278e-09, 1, 1
|m| > 1.0
8.85413e-09, 1.65574, 8.85413e-09, 1, 1
|m| > 1.0
2.30503e-08, 1.65574, 2.30503e-08, 1, 1
|m| > 1.0
5.93925e-08, 1.65574, 5.93925e-08, 1, 1
|m| > 1.0
1.16676e-07, 1.65574, 1.16676e-07, 1, 1
|m| > 1.0
2.37997e-07, 1.65574, 2.37997e-07, 1, 1
|m| > 1.0
4.68466e-07, 1.65574, 4.68466e-07, 1, 1
|m| > 1.0
9.3827e-07, 1.65574, 9.3827e-07, 1, 1
|m| > 1.0
1.10399e-06, 1.65574, 1.10399e-06, 1, 1
|m| > 1.0
3.29178e-06, 1.65574, 3.29178e-06, 1, 1
|m| > 1.0
7.51721e-06, 1.65574, 7.51721e-06, 1, 1
|m| > 1.0
1.51147e-05, 1.65574, 1.51147e-05, 1, 1
|m| > 1.0
2.9864e-05, 1.65574, 2.9864e-05, 1, 1
|m| > 1.0
3.38703e-05, 1.65574, 3.38703e-05, 1, 1
|m| > 1.0
9.06601e-05, 1.65574, 9.06601e-05, 1, 1
|m| > 1.0
0.000219495, 1.65574, 0.000219495, 1, 1
|m| > 1.0
0.000439522, 1.65574, 0.000439521, 1, 1
|m| > 1.0
0.000633315, 1.65574, 0.000633315, 1, 0.999999
|m| > 1.0
0.00111512, 1.65574, 0.00111512, 0.999999, 0.999998
|m| > 1.0
0.00196247, 1.65574, 0.00196246, 0.999998, 0.999995
|m| > 1.0
0.00555375, 1.65574, 0.00555365, 0.999985, 0.999958
|m| > 1.0
```

```
0.00869113, 1.65574, 0.00869072, 0.999962, 0.999896
|m| > 1.0
0.0299334, 1.65574, 0.0299166, 0.999552, 0.998772
|m| > 1.0
0.0512426, 1.65574, 0.0511588, 0.998691, 0.996406
|m| > 1.0
0.112013, 1.65574, 0.111143, 0.993804, 0.982922
|m| > 1.0
0.234804, 1.65574, 0.227, 0.973895, 0.926679
|m| > 1.0
0.489873, 1.65574, 0.425971, 0.904737, 0.708912
|m| > 1.0
0.751831, 1.65574, 0.553446, 0.832885, 0.400346
|m| > 1.0
1.65574, 1.65574, 0.408154, 0.912913, -0.737088
```

Error Output For jacobi_dn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Mathworld Data

```
|m| > 1.0
2.98023e-08, 1.5, 2.98023e-08, 1, 1
|m| > 1.0
-2.98023e-08, 1.5, -2.98023e-08, 1, 1
|m| > 1.0
0.25, 1.5, 0.24183, 0.970319, 0.931888
|m| > 1.0
-0.25, 1.5, -0.24183, 0.970319, 0.931888
|m| > 1.0
1.25, 1.5, 0.665876, 0.746063, -0.0486921
|m| > 1.0
-1.25, 1.5, -0.665876, 0.746063, -0.0486921
|m| > 1.0
25, 1.5, 0.618665, 0.785655, 0.372585
|m| > 1.0
-25, 1.5, -0.618665, 0.785655, 0.372585
```

Error Output For jacobi_cn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Mathworld Data

```
|m| > 1.0
2.98023e-08, 1.5, 2.98023e-08, 1, 1
|m| > 1.0
-2.98023e-08, 1.5, -2.98023e-08, 1, 1
|m| > 1.0
0.25, 1.5, 0.24183, 0.970319, 0.931888
|m| > 1.0
-0.25, 1.5, -0.24183, 0.970319, 0.931888
|m| > 1.0
1.25, 1.5, 0.665876, 0.746063, -0.0486921
|m| > 1.0
-1.25, 1.5, -0.665876, 0.746063, -0.0486921
|m| > 1.0
25, 1.5, 0.618665, 0.785655, 0.372585
|m| > 1.0
-25, 1.5, -0.618665, 0.785655, 0.372585
```

Error Output For jacobi_sn with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Jacobi Elliptic: Mathworld Data

```
|m| > 1.0
2.98023e-08, 1.5, 2.98023e-08, 1, 1
|m| > 1.0
-2.98023e-08, 1.5, -2.98023e-08, 1, 1
|m| > 1.0
0.25, 1.5, 0.24183, 0.970319, 0.931888
|m| > 1.0
-0.25, 1.5, -0.24183, 0.970319, 0.931888
|m| > 1.0
1.25, 1.5, 0.665876, 0.746063, -0.0486921
|m| > 1.0
-1.25, 1.5, -0.665876, 0.746063, -0.0486921
|m| > 1.0
25, 1.5, 0.618665, 0.785655, 0.372585
|m| > 1.0
-25, 1.5, -0.618665, 0.785655, 0.372585
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Mathematica Data - Large orders and other bug cases

```
underflow
168, 150, -6.52661e-66
underflow
169, 202, 9.2734e-88
domain error
20, -9.5, -0.00103076
domain error
21, -9.5, 4.28582e+26
domain error
22, -9.5, -0.00419144
domain error
23, -9.5, 8.6745e+29
domain error
24, -9.5, -0.0204825
domain error
25, -9.5, 2.08188e+33
domain error
26, -9.5, -0.118403
domain error
27, -9.5, 5.84592e+36
domain error
28, -9.5, -0.798969
domain error
29, -9.5, 1.89875e+40
domain error
30, -9.5, -6.22245
underflow
10, 1.32923e+36, -0
underflow
15, 1.32923e+36, 0
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Mathematica Data - large negative arguments

```
domain error
124, -1.5, 7.63705e+240
domain error
124, -2.5, 7.63705e+240
domain error
124, -3.5, 7.63705e+240
domain error
124, -4.5, 7.63705e+240
domain error
124, -5.5, 7.63705e+240
domain error
124, -6.5, 7.63705e+240
domain error
124, -7.5, 7.63705e+240
domain error
124, -8.5, 7.63705e+240
domain error
124, -9.5, 7.63705e+240
domain error
124, -10.5, 7.63705e+240
domain error
124, -11.5, 7.63705e+240
domain error
124, -12.5, 7.63705e+240
domain error
124, -13.5, 7.63705e+240
domain error
124, -14.5, 7.63705e+240
domain error
124, -15.5, 7.63705e+240
domain error
124, -16.5, 7.63705e+240
domain error
124, -17.5, 7.63705e+240
domain error
124, -18.5, 7.63705e+240
domain error
124, -19.5, 7.63705e+240
domain error
124, -20.5, 7.63705e+240
domain error
124, -1.5, -7.63705e+240
domain error
124, -2.5, -7.63705e+240
domain error
124, -3.5, -7.63705e+240
domain error
124, -4.5, -7.63705e+240
domain error
124, -5.5, -7.63705e+240
domain error
124, -6.5, -7.63705e+240
domain error
124, -7.5, -7.63705e+240
```

```
domain error
124, -8.5, -7.63705e+240
domain error
124, -9.5, -7.63705e+240
domain error
124, -10.5, -7.63705e+240
domain error
124, -11.5, -7.63705e+240
domain error
124, -12.5, -7.63705e+240
domain error
124, -13.5, -7.63705e+240
domain error
124, -14.5, -7.63705e+240
domain error
124, -15.5, -7.63705e+240
domain error
124, -16.5, -7.63705e+240
domain error
124, -17.5, -7.63705e+240
domain error
124, -18.5, -7.63705e+240
domain error
124, -19.5, -7.63705e+240
domain error
124, -20.5, -7.63705e+240
domain error
2, -0.5, -0.828797
domain error
3, -0.5, 193.409
domain error
4, -0.5, -3.47425
domain error
5, -0.5, 15371.1
domain error
6, -0.5, -43.4579
domain error
7, -0.5, 2.58068e+06
domain error
8, -0.5, -1059.96
domain error
9, -0.5, 7.43185e+08
domain error
10, -0.5, -42108.9
domain error
11, -0.5, 3.26999e+11
domain error
12, -0.5, -2.46448e+06
domain error
13, -0.5, 2.04047e+14
domain error
14, -0.5, -1.9918e+08
domain error
15, -0.5, 1.71399e+17
domain error
16, -0.5, -2.12394e+10
domain error
17, -0.5, 1.86483e+20
```

```
domain error
18, -0.5, -2.88824e+12
domain error
19, -0.5, 2.55108e+23
domain error
20, -0.5, -4.87773e+14
domain error
21, -0.5, 4.28582e+26
domain error
2, -0.5, -0.828843
domain error
3, -0.5, 193.409
domain error
4, -0.5, -3.47791
domain error
5, -0.5, 15371.1
domain error
6, -0.5, -44.0732
domain error
7, -0.5, 2.58068e+06
domain error
8, -0.5, -1237.15
domain error
9, -0.5, 7.43185e+08
domain error
10, -0.5, -120071
domain error
11, -0.5, 3.26999e+11
domain error
12, -0.5, -5.11131e+07
domain error
13, -0.5, 2.04047e+14
domain error
14, -0.5, -4.1064e+10
domain error
15, -0.5, 1.71399e+17
domain error
16, -0.5, -4.44822e+13
domain error
17, -0.5, 1.86483e+20
domain error
18, -0.5, -6.08254e+16
domain error
19, -0.5, 2.55108e+23
domain error
20, -0.5, -1.02182e+20
domain error
21, -0.5, 4.28582e+26
domain error
2, -0.5, -0.828751
domain error
3, -0.5, 193.409
domain error
4, -0.5, -3.47059
domain error
5, -0.5, 15371.1
domain error
6, -0.5, -42.8426
```

```
domain error
7, -0.5, 2.58068e+06
domain error
8, -0.5, -882.773
domain error
9, -0.5, 7.43185e+08
domain error
10, -0.5, 35853.7
domain error
11, -0.5, 3.26999e+11
domain error
12, -0.5, 4.61841e+07
domain error
13, -0.5, 2.04047e+14
domain error
14, -0.5, 4.06656e+10
domain error
15, -0.5, 1.71399e+17
domain error
16, -0.5, 4.44397e+13
domain error
17, -0.5, 1.86483e+20
domain error
18, -0.5, 6.08197e+16
domain error
19, -0.5, 2.55108e+23
domain error
20, -0.5, 1.02181e+20
domain error
21, -0.5, 4.28582e+26
```

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Mathematica Data - negative arguments

```
domain error
2, -12.75, -124.031
domain error
2, -12.25, 124.019
domain error
2, -11.75, -124.032
domain error
2, -11.25, 124.018
domain error
2, -10.75, -124.033
domain error
2, -10.25, 124.016
domain error
2, -9.75, -124.035
domain error
2, -9.25, 124.015
domain error
2, -8.75, -124.037
domain error
2, -8.25, 124.012
domain error
2, -7.75, -124.04
domain error
2, -7.25, 124.009
```

```
domain error
2, -6.75, -124.044
domain error
2, -6.25, 124.003
domain error
2, -5.75, -124.051
domain error
2, -5.25, 123.995
domain error
2, -4.75, -124.061
domain error
2, -4.25, 123.981
domain error
2, -3.75, -124.08
domain error
2, -3.25, 123.955
domain error
2, -2.75, -124.118
domain error
2, -2.25, 123.897
domain error
2, -1.75, -124.214
domain error
2, -1.25, 123.721
domain error
2, -0.75, -124.587
domain error
2, -0.25, 122.697
domain error
3, -12.75, 1558.54
domain error
3, -12.25, 1558.54
domain error
3, -11.75, 1558.54
domain error
3, -11.25, 1558.54
domain error
3, -10.75, 1558.54
domain error
3, -10.25, 1558.54
domain error
3, -9.75, 1558.54
domain error
3, -9.25, 1558.54
domain error
3, -8.75, 1558.54
domain error
3, -8.25, 1558.54
domain error
3, -7.75, 1558.54
domain error
3, -7.25, 1558.54
domain error
3, -6.75, 1558.54
domain error
3, -6.25, 1558.54
domain error
3, -5.75, 1558.54
```

```
domain error
3, -5.25, 1558.54
domain error
3, -4.75, 1558.53
domain error
3, -4.25, 1558.53
domain error
3, -3.75, 1558.52
domain error
3, -3.25, 1558.51
domain error
3, -2.75, 1558.49
domain error
3, -2.25, 1558.46
domain error
3, -1.75, 1558.38
domain error
3, -1.25, 1558.22
domain error
3, -0.75, 1557.75
domain error
3, -0.25, 1555.76
domain error
4, -12.75, -24481.6
domain error
4, -12.25, 24481.6
domain error
4, -11.75, -24481.6
domain error
4, -11.25, 24481.6
domain error
4, -10.75, -24481.6
domain error
4, -10.25, 24481.6
domain error
4, -9.75, -24481.6
domain error
4, -9.25, 24481.6
domain error
4, -8.75, -24481.6
domain error
4, -8.25, 24481.6
domain error
4, -7.75, -24481.6
domain error
4, -7.25, 24481.6
domain error
4, -6.75, -24481.6
domain error
4, -6.25, 24481.6
domain error
4, -5.75, -24481.6
domain error
4, -5.25, 24481.6
domain error
4, -4.75, -24481.6
domain error
4, -4.25, 24481.6
```

```
domain error
4, -3.75, -24481.6
domain error
4, -3.25, 24481.5
domain error
4, -2.75, -24481.6
domain error
4, -2.25, 24481.5
domain error
4, -1.75, -24481.8
domain error
4, -1.25, 24481.1
domain error
4, -0.75, -24483.2
domain error
4, -0.25, 24473.2
domain error
5, -12.75, 492231
domain error
5, -12.25, 492231
domain error
5, -11.75, 492231
domain error
5, -11.25, 492231
domain error
5, -10.75, 492231
domain error
5, -10.25, 492231
domain error
5, -9.75, 492231
domain error
5, -9.25, 492231
domain error
5, -8.75, 492231
domain error
5, -8.25, 492231
domain error
5, -7.75, 492231
domain error
5, -7.25, 492231
domain error
5, -6.75, 492231
domain error
5, -6.25, 492231
domain error
5, -5.75, 492231
domain error
5, -5.25, 492231
domain error
5, -4.75, 492231
domain error
5, -4.25, 492231
domain error
5, -3.75, 492231
domain error
5, -3.25, 492231
domain error
5, -2.75, 492231
```

```
domain error
5, -2.25, 492231
domain error
5, -1.75, 492231
domain error
5, -1.25, 492230
domain error
5, -0.75, 492227
domain error
5, -0.25, 492199
domain error
6, -12.75, -1.17912e+07
domain error
6, -12.25, 1.17912e+07
domain error
6, -11.75, -1.17912e+07
domain error
6, -11.25, 1.17912e+07
domain error
6, -10.75, -1.17912e+07
domain error
6, -10.25, 1.17912e+07
domain error
6, -9.75, -1.17912e+07
domain error
6, -9.25, 1.17912e+07
domain error
6, -8.75, -1.17912e+07
domain error
6, -8.25, 1.17912e+07
domain error
6, -7.75, -1.17912e+07
domain error
6, -7.25, 1.17912e+07
domain error
6, -6.75, -1.17912e+07
domain error
6, -6.25, 1.17912e+07
domain error
6, -5.75, -1.17912e+07
domain error
6, -5.25, 1.17912e+07
domain error
6, -4.75, -1.17912e+07
domain error
6, -4.25, 1.17912e+07
domain error
6, -3.75, -1.17912e+07
domain error
6, -3.25, 1.17912e+07
domain error
6, -2.75, -1.17912e+07
domain error
6, -2.25, 1.17912e+07
domain error
6, -1.75, -1.17912e+07
domain error
6, -1.25, 1.17912e+07
```

*** FURTHER CONTENT HAS BEEN TRUNCATED FOR BREVITY ***

Error Output For polygamma with compiler GNU C++ version 7.1.0 and library GSL 2.1 and test data Mathematica Data - large arguments

```
underflow
30, 8.58993e+09, -8.44974e-268
underflow
30, 1.71799e+10, -7.86943e-277
underflow
30, 3.43597e+10, -7.32898e-286
underflow
30, 6.87195e+10, -6.82564e-295
underflow
30, 1.37439e+11, -6.35687e-304
underflow
30, 2.74878e+11, -5.9203e-313
underflow
30, 5.49756e+11, -5.53354e-322
underflow
30, 1.09951e+12, -0
underflow
30, 2.19902e+12, -0
underflow
30, 4.39805e+12, -0
underflow
30, 8.79609e+12, -0
underflow
30, 1.75922e+13, -0
underflow
30, 3.51844e+13, -0
underflow
30, 7.03687e+13, -0
underflow
30, 1.40737e+14, -0
underflow
30, 2.81475e+14, -0
underflow
30, 5.6295e+14, -0
underflow
30, 1.1259e+15, -0
underflow
30, 2.2518e+15, -0
underflow
30, 4.5036e+15, -0
underflow
30, 9.0072e+15, -0
underflow
30, 1.80144e+16, -0
underflow
30, 3.60288e+16, -0
underflow
30, 7.20576e+16, -0
underflow
30, 1.44115e+17, -0
underflow
30, 2.8823e+17, -0
underflow
```

```
30, 5.76461e+17, -0
underflow
30, 1.15292e+18, -0
underflow
30, 2.30584e+18, -0
underflow
30, 4.61169e+18, -0
underflow
30, 9.22337e+18, -0
underflow
30, 1.84467e+19, -0
underflow
30, 3.68935e+19, -0
underflow
30, 7.3787e+19, -0
underflow
30, 1.47574e+20, -0
underflow
30, 2.95148e+20, -0
underflow
30, 5.90296e+20, -0
underflow
30, 1.18059e+21, -0
underflow
30, 2.36118e+21, -0
underflow
30, 4.72237e+21, -0
underflow
30, 9.44473e+21, -0
underflow
30, 1.88895e+22, -0
underflow
30, 3.77789e+22, -0
underflow
30, 7.55579e+22, -0
underflow
30, 1.51116e+23, -0
underflow
30, 3.02231e+23, -0
underflow
30, 6.04463e+23, -0
underflow
30, 1.20893e+24, -0
underflow
30, 2.41785e+24, -0
underflow
30, 4.8357e+24, -0
underflow
30, 9.67141e+24, -0
underflow
30, 1.93428e+25, -0
underflow
30, 3.86856e+25, -0
underflow
30, 7.73713e+25, -0
underflow
30, 1.54743e+26, -0
underflow
```

```
30, 3.09485e+26, -0
underflow
30, 6.1897e+26, -0
underflow
30, 1.23794e+27, -0
underflow
30, 2.47588e+27, -0
underflow
30, 4.95176e+27, -0
underflow
30, 9.90352e+27, -0
underflow
30, 1.9807e+28, -0
underflow
30, 3.96141e+28, -0
underflow
30, 7.92282e+28, -0
underflow
30, 1.58456e+29, -0
underflow
30, 3.16913e+29, -0
underflow
30, 6.33825e+29, -0
underflow
30, 1.26765e+30, -0
```

Library Status

History and What's New

Currently open bug reports can be viewed here.

All bug reports including closed ones can be viewed here and here.

Math-2.10.0 (Boost-1.71)

- Catmull-Rom interpolator now works in C++11
- Cardinal quadratic B-spline interpolation
- Domain of elliptic integrals extended
- sin_pi and cos_pi performance improvements
- Forward-mode automatic differentiation
- Vector valued barycentric rational interpolation
- Ooura's method for evaluation of Fourier integrals
- Multiple compatibility issues with Multiprecision fixed
- Lambert-W fixed on a rare architecture

Math-2.9.0 (Boost-1.70)

- Add Lanczos smoothing derivatives
- Move `numerical_differentiation.hpp` from `boost/math/tools/` to `boost/math/differentiation/finite_difference.hpp`.
- Add mean, variance, skewness, kurtosis, median, Gini coefficient, and median absolute deviation to `tools/univariate_statistics.hpp`.
- Add correlation coefficients and covariance to `tools/bivariate_statistics.hpp`
- Add absolute Gini coefficient, Hoyer sparsity, oracle SNR, and the M_2M_4 SNR estimator to `tools/signal_statistics.hpp`.
- Add total variation, l0, l1, l2, and sup norms, as well as corresponding distance functions to `tools/norms.hpp`.
- Add move constructors for polynomials, support complex coefficients, add `.prime()` and `.integrate()` methods.
- Add `quadratic_roots` to `tools/roots.hpp`.
- Add support for complex-valued functions to Newton's method in `roots.hpp`.
- Add Catmull-Rom interpolator.
- Fix bug in `newton_raphson_iterate` where we could terminate incorrectly under certain exceptional cases.
- Suppressed warnings from use of `Q` suffix on constants when compiling with `g++ -Wall -pedantic`.

Math-2.8.0 (Boost-1.69)

- Add LambertW functions.
- Update integration routines to support complex valued integrands and contour integrals.
- Added the derivative of the Barycentric rational approximation.

- Updated continued fraction and series evaluation code to support complex types.
- Minor fixes to better support variable precision floating point types.
- Removed use of deprecated Boost.Endian in favour of Predef.
- Prevent logic error leading to infinite loop in toms748_solve. See #138.
- Fix mean and standard_deviation for extreme_value_distribution. See #139.
- Improve heuristics used in newton_raphson_iterate. See #145.
- Fix result of `erf(NaN)`. See #141.
- Big push to reduce GCC warnings. See #136.
- Refactor polynomial addition. See PR132.
- Fix for vxWorks having a `real` function in the global namespace. See PR131.
- Improve `sinc` approximations and add better tests.
- Fix typo in Student's T hypothesis testing documentation, see #143.

Math-2.7.1 (Boost-1.68)

- Continue to improve numerical integration routines, and in particular add support for contour integrals.
- Improve accuracy of erfc function's rational approximations.

Math-2.7.0 (Boost-1.66)

- Add Gauss and Gauss-Kronrod quadrature routines.
- Add double-exponential (tanh-sinh, exp-sinh and sinh-sinh) quadrature routines.
- Add Chebyshev polynomial evaluation, roots, integration, differentiation, and interpolation routines.

Math-2.6.0 (Boost-1.65)

New Features:

- Add cubic B-Spline interpolation functions, with thanks to Nick Thompson.
- Add barycentric rational interpolation functions, with thanks to Nick Thompson.
- Add adaptive trapezoidal quadrature, with thanks to Nick Thompson.
- Add support for the zeros and derivatives of Legendre polynomials, with thanks to Nick Thompson.

Patches:

- Improve polynomial GCD algorithms, with thanks to Jeremy Murphy.
- Removed gcd/lcm routines from Boost.Math - these are now in Boost.Integer.

Math-2.5.2 (Boost-1.64)

Patches:

- Big push to ensure all functions in also in C99 are compatible with Annex F.

- Improved accuracy of the Bessel functions I0, I1, K0 and K1, see 12066.

Math-2.5.1 (Boost-1.63)

Patches:

- Fixed evaluation of zero polynomial in polynomial.hpp, see 12532.
- Fixed missing header include in boost/math/tools/tuple.hpp, see 12537.

Math-2.5.0 (Boost-1.62)

New Features:

- Enabled all the special function code to work correctly with types whose precision can change at runtime: for example type `mpfr_float` from Boost.Multiprecision.

Patches:

- Fix tgamma_delta_ratio for cases where the delta is small compared to the base.
- Fix misc GCC-4.4 test failures.

Math-2.4.0 (Boost-1.61)

New Features:

- Polynomial arithmetic added to tools.

Math-2.3.0 (Boost-1.60)

New Features:

- Promote the root finding and function minimization code to first class citizens - these are now officially supported as part of the library.
- Allow the library to be used and tested with compiler exception handling support turned off. To better facilitate this the default error handling policies have been changed to *errno_on_error* but only when the compiler has no exception handling support.

Patches:

- Fix behaviour of the non-central chi-squared distribution when the non-centrality parameter is zero to match the chi-squared, see 11557.
- Fix comments in code for the hypergeometric to match what it actually does, also fixes the parameter access functions to return the correct values. See 11556.
- Stopped using hidden visiblity library build with the Oracle compiler as it leads to unresolved externals from the C++ standard library. See 11547.
- Fix unintended use of __declspec when building with Oracle C++. See 11546.
- Fix corner case bug in root bracketing code, see 11532.
- Add some missing typecasts in arguments to std::max in Bernoulli code. See 11453.
- Fix mistaken assumptions about the possible values for FLT_EVAL_METHOD. See 11429.
- Completely revamped performance testing and error-rate measuring code so we can more easily document how well (or not!) we're doing. This information will hopefully get more frequently updated in future as it's more or less automatically generated - see the *reporting* sub-directory for more information.

- Fix some corner cases in the beta, incomplete beta, and incomplete beta derivative. With thanks to Rocco Romeo.
- Reorganized the Bessel functions internally to improve the performance of Jn and Yn.
- Fixed skewness formula for triangular distribution, see #11768.
- Fixed some examples so they compile on Unix platforms which have an ::exception struct declared, see #11827.
- Correct mistake in triangular distribution skewness formula 11768, reported by Juan Leni.

Math-2.2.1

Patch release for Boost-1.58:

- Minor patch for Haiku support.
- Fix the decimal digit count for 128-bit floating point types.
- Fix a few documentation typos.

Math-2.2.0 (boost-1.58.0)

- Added two new special functions - trigamma and polygamma.
- Fixed namespace scope constants so they are constexpr on conforming compilers, see https://svn.boost.org/trac/boost/ticket/10901.
- Fixed various cases of spurious under/overflow in the incomplete beta and gamma functions, plus the elliptic integrals, with thanks to Rocco Romeo.
- Fix 3-arg legendre_p and legendre_q functions to not call the policy based overload if the final argument is not actually a policy.
- Cleaned up some dead code in the incomplete beta function, see #10985.
- Fixed extreme-value pdf for large valued inputs, see #10938.
- Large update to the Elliptic integral code to use Carlson's latest algorithms - these should be more stable, more accurate and slightly faster than before. Also added support for Carlson's RG integral.
- Added ellint_d, jacobi_zeta and heuman_lambda elliptic integrals.
- Switched documentation to use SVG rather than PNG graphs and equations - browsers seem to have finally caught up!

Math-2.1.0 (boost-1.57.0)

- Added Hyperexponential Distribution.
- Fix some spurious overflows in the incomplete gamma functions (with thanks to Rocco Romeo).
- Fix bug in derivative of incomplete beta when a = b = 0.5 - this also effects several non-central distributions, see 10480.
- Fixed some corner cases in round.
- Don't support 80-bit floats in cstdfloat.hpp if standard library support is broken.

Math-2.0.0 (Boost-1.56.0)

- **Breaking change**: moved a number of non-core headers that are predominantly used for internal maintenance into `libs/math/include_private`. The headers effected are `boost/math/tools/test_data.hpp`, `boost/math/tools/remez.hpp`, `boost/math/constants/generate.hpp`, `boost/math/tools/solve.hpp`, `boost/math/tools/test.hpp`. You can continue to use these headers by adding `libs/math/include_private` to your compiler's include path.

- **Breaking change**: A number of distributions and special functions were returning the maximum finite value rather than raising an overflow_error, this has now been fixed, which means these functions now behave as documented. However, since the default behavior on raising an overflow_error is to throw a `std::overflow_error` exception, applications which have come to reply rely on these functions not throwing may experience exceptions where they did not before. The special functions involved are gamma_p_inva, gamma_q_inva, ibeta_inva, ibetac_inva, ibeta_invb, ibetac_invb, gamma_p_inv, gamma_q_inv. The distributions involved are Pareto Distribution, Beta Distribution, Geometric Distribution, Negative Binomial Distribution, Binomial Distribution, Chi Squared Distribution, Gamma Distribution, Inverse chi squared Distribution, Inverse Gamma Distribution. See #10111.
- Fix round and trunc functions so they can be used with integer arguments, see #10066.
- Fix Halley iteration to handle zero derivative (with non-zero second derivative), see #10046.

Math-1.9.1

- Fix Geometric distribution use of Policies, see #9833.
- Fix corner cases in the negative binomial distribution, see #9834.
- Fix compilation failures on Mac OS.

Math-1.9.0

- Changed version number to new Boost.Math specific version now that we're in the modular Boost world.
- Added Bernoulli numbers, changed arbitrary precision tgamma/lgamma to use Sterling's approximation (from Nikhar Agrawal).
- Added first derivatives of the Bessel functions: cyl_bessel_j_prime, cyl_neumann_prime, cyl_bessel_i_prime, cyl_bessel_k_prime, sph_bessel_prime and sph_neumann_prime (from Anton Bikineev).
- Fixed buggy Student's t example code, along with docs for testing sample means for equivalence.
- Documented `max_iter` parameter in root finding code better, see #9225.
- Add option to explicitly enable/disable use of __float128 in constants code, see #9240.
- Cleaned up handling of negative values in Bessel I0 and I1 code (removed dead code), see #9512.
- Fixed handling of very small values passed to tgamma and lgamma so they don't generate spurious overflows (thanks to Rocco Romeo).
- #9672 PDF and CDF of a Laplace distribution throwing domain_error Random variate can now be infinite.
- Fixed several corner cases in rising_factorial, falling_factorial and tgamma_delta_ratio with thanks to Rocco Romeo.
- Fixed several corner cases in rising_factorial, falling_factorial and tgamma_delta_ratio (thanks to Rocco Romeo).
- Removed constant `pow23_four_minus_pi` whose value did not match the name (and was unused by Boost.Math), see #9712.

Boost-1.55

- Suppress numerous warnings (mostly from GCC-4.8 and MSVC) #8384, #8855, #9107, #9109..
- Fixed PGI compilation issue #8333.
- Fixed PGI constant value initialization issue that caused erf to generate incorrect results #8621.
- Prevent macro expansion of some C99 macros that are also C++ functions #8732 and #8733..
- Fixed Student's T distribution to behave correctly with huge degrees of freedom (larger than the largest representable integer) #8837.

- Make some core functions usable with `long double` even when the platform has no standard library `long double` support #8940.
- Fix error handling of distributions to catch invalid scale and location parameters when the random variable is infinite #9042 and #9126.
- Add workaround for broken <tuple> in Intel C++ 14 #9087.
- Improve consistency of argument reduction in the elliptic integrals #9104.
- Fix bug in inverse incomplete beta that results in cancellation errors when the beta function is really an arcsine or Student's T distribution.
- Fix issue in Bessel I and K function continued fractions that causes spurious over/underflow.
- Add improvement to non-central chi squared distribution quantile due to Thomas Luu, Fast and accurate parallel computation of quantile functions for random number generation, Doctorial Thesis 2016. Efficient and Accurate Parallel Inversion of the Gamma Distribution, Thomas Luu

Boost-1.54

- Major reorganization to incorporate other Boost.Math like Integer Utilities Integer Utilities (Greatest Common Divisor and Least Common Multiple), quaternions and octonions. Making new chapter headings.
- Added many references to Boost.Multiprecision and `cpp_dec_float_50` as an example of a User-defined Type (UDT).
- Added Clang to list of supported compilers.
- Fixed constants to use a thread-safe cache of computed values when used at arbitrary precision.
- Added finding zeros of Bessel functions `cyl_bessel_j_zero`, `cyl_neumann_zero`, `airy_ai_zero` and `airy_bi_zero`(by Christopher Kormanyos).
- More accuracy improvements to the Bessel J and Y functions from Rocco Romeo.
- Fixed nasty cyclic dependency bug that caused some headers to not compile #7999.
- Fixed bug in tgamma that caused spurious overflow for arguments between 142.5 and 143.
- Fixed bug in raise_rounding_error that caused it to return an incorrect result when throwing an exception is turned off #7905.
- Added minimal __float128 support.
- Fixed bug in edge-cases of poisson quantile #8308.
- Adjusted heuristics used in Halley iteration to cope with inverting the incomplete beta in tricky regions where the derivative is flatlining. Example is computing the quantile of the Fisher F distribution for probabilities smaller than machine epsilon. See ticket #8314.

Boost-1.53

- Fixed issues #7325, #7415 and #7416, #7183, #7649, #7694, #4445, #7492, #7891, #7429.
- Fixed mistake in calculating pooled standard deviation in two-sample students t example #7402.
- Improve complex acos/asin/atan, see #7290, #7291.
- Improve accuracy in some corner cases of cyl_bessel_j and gamma_p/gamma_q thanks to suggestions from Rocco Romeo.
- Improve accuracy of Bessel J and Y for integer orders thanks to suggestions from Rocco Romeo.

Boost-1.52

- Corrected moments for small degrees of freedom #7177 (reported by Thomas Mang).
- Added Airy functions and Jacobi Elliptic functions.
- Corrected failure to detect bad parameters in many distributions #6934 (reported by Florian Schoppmann) by adding a function check_out_of_range to test many possible bad parameters. This test revealed several distributions where the checks for bad parameters were ineffective, and these have been rectified.
- Fixed issue in Hankel functions that causes incorrect values to be returned for $x < 0$ and ν odd, see #7135.
- Fixed issues #6517, #6362, #7053, #2693, #6937, #7099.
- Permitted infinite degrees of freedom #7259 implemented using the normal distribution (requested by Thomas Mang).
- Much enhanced accuracy for large degrees of freedom ν and/or large non-centrality δ by switching to use the Students t distribution (or Normal distribution for infinite degrees of freedom) centered at delta, when δ / (4 * ν) < epsilon for the floating-point type in use. #7259. It was found that the incomplete beta was suffering from serious cancellation errors when degrees of freedom was very large. (That has now been fixed in our code, but any code based on Didonato and Morris's original papers (probably every implementation out there actually) will have the same issue).

Boost-1.51

See Boost-1.52 - some items were added but not listed in time for the release.

Boost-1.50

- Promoted math constants to be 1st class citizens, including convenient access to the most widely used built-in float, double, long double via three namespaces.
- Added the Owen's T function and Skew Normal distribution written by Benjamin Sobotta: see Owens T and skew_normal_distrib.
- Added Hankel functions cyl_hankel_1, cyl_hankel_2, sph_hankel_1 and sph_hankel_2.
- Corrected issue #6627 nonfinite_num_put formatting of 0.0 is incorrect based on a patch submitted by K R Walker.
- Changed constant initialization mechanism so that it is thread safe even for user-defined types, also so that user defined types get the full precision of the constant, even when `long double` does not. So for example 128-bit rational approximations will work with UDT's and do the right thing, even though `long double` may be only 64 or 80 bits.
- Fixed issue in `bessel_jy` which causes $Y_{8.5}(4\pi)$ to yield a NaN.

Boost-1.49

- Deprecated wrongly named `twothirds` math constant in favour of `two_thirds` (with underscore separator). (issue #6199).
- Refactored test data and some special function code to improve support for arbitary precision and/or expression-template-enabled types.
- Added new faster zeta function evaluation method.

Fixed issues:

- Corrected CDF complement for Laplace distribution (issue #6151).
- Corrected branch cuts on the complex inverse trig functions, to handle signed zeros (issue #6171).
- Fixed bug in `bessel_yn` which caused incorrect overflow errors to be raised for negative n (issue #6367).
- Also fixed minor/cosmetic/configuration issues #6120, #6191, #5982, #6130, #6234, #6307, #6192.

Boost-1.48

- Added new series evaluation methods to the cyclic Bessel I, J, K and Y functions. Also taken great care to avoid spurious over and underflow of these functions. Fixes issue #5560
- Added an example of using Inverse Chi-Squared distribution for Bayesian statistics, provided by Thomas Mang.
- Added tests to use improved version of lexical_cast which handles C99 nonfinites without using globale facets.
- Corrected wrong out-of-bound uniform distribution CDF complement values #5733.
- Enabled long double support on OpenBSD (issue #6014).
- Changed nextafter and related functions to behave in the same way as other implementations - so that nextafter(+INF, 0) is a finite value (issue #5832).
- Changed tuple include configuration to fix issue when using in conjunction with Boost.Tr1 (issue #5934).
- Changed class eps_tolerance to behave correctly when both ends of the range are zero (issue #6001).
- Fixed missing include guards on prime.hpp (issue #5927).
- Removed unused/undocumented constants from constants.hpp (issue #5982).
- Fixed missing std:: prefix in nonfinite_num_facets.hpp (issue #5914).
- Minor patches for Cray compiler compatibility.

Boost-1.47

- Added changesign function to sign.hpp to facilitate addition of nonfinite facets.
- Addition of nonfinite facets from Johan Rade, with tests, examples of use for C99 format infinity and NaN, and documentation.
- Added tests and documentation of changesign from Johan Rade.

Boost-1.46.1

- Fixed issues #5095, #5113.

Boost-1.46.0

- Added Wald, Inverse Gaussian and geometric distributions.
- Added information about configuration macros.
- Added support for mpreal as a real-numbered type.

Boost-1.45.0

- Added warnings about potential ambiguity with std random library in distribution and function names.
- Added inverse gamma distribution and inverse chi_square and scaled inverse chi_square.
- Editorial revision of documentation, and added FAQ.

Boost-1.44.0

- Fixed incorrect range and support for Rayleigh distribution.

- Fixed numerical error in the quantile of the Student's T distribution: the function was returning garbage values for non-integer degrees of freedom between 2 and 3.

Boost-1.41.0

- Significantly improved performance for the incomplete gamma function and its inverse.

Boost-1.40.0

- Added support for MPFR as a bignum type.
- Added some full specializations of the policy classes to reduce compile times.
- Added logistic and hypergeometric distributions, from Gautam Sewani's Google Summer of Code project.
- Added Laplace distribution submitted by Thijs van den Berg.
- Updated performance test code to include new distributions, and improved the performance of the non-central distributions.
- Added SSE2 optimised Lanczos approximation code, from Gautam Sewani's Google Summer of Code project.
- Fixed bug in cyl_bessel_i that used an incorrect approximation for ν = 0.5, also effects the non-central Chi Square Distribution when ν = 3, see bug report #2877.
- Fixed minor bugs #2873.

Boost-1.38.0

- Added Johan Råde's optimised floating point classification routines.
- Fixed code so that it compiles in GCC's -pedantic mode (bug report #1451).

Boost-1.37.0

- Improved accuracy and testing of the inverse hypergeometric functions.

Boost-1.36.0

- Added Noncentral Chi Squared Distribution.
- Added Noncentral Beta Distribution.
- Added Noncentral F Distribution.
- Added Noncentral T Distribution.
- Added Exponential Integral Functions.
- Added Zeta Function.
- Added Rounding and Truncation functions.
- Added Compile time powers of runtime bases.
- Added SSE2 optimizations for Lanczos evaluation.

Boost-1.35.0: Post Review First Official Release

- Added Policy based framework that allows fine grained control over function behaviour.

- **Breaking change:** Changed default behaviour for domain, pole and overflow errors to throw an exception (based on review feedback), this behaviour can be customised using Policy's.
- **Breaking change:** Changed exception thrown when an internal evaluation error occurs to boost::math::evaluation_error.
- **Breaking change:** Changed discrete quantiles to return an integer result: this is anything up to 20 times faster than finding the true root, this behaviour can be customised using Policy's.
- Polynomial/rational function evaluation is now customisable and hopefully faster than before.
- Added performance test program.

Milestone 4: Second Review Candidate (1st March 2007)

- Moved Xiaogang Zhang's Bessel Functions code into the library, and brought them into line with the rest of the code.
- Added C# "Distribution Explorer" demo application.

Milestone 3: First Review Candidate (31st Dec 2006)

- Implemented the main probability distribution and density functions.
- Implemented digamma.
- Added more factorial functions.
- Implemented the Hermite, Legendre and Laguerre polynomials plus the spherical harmonic functions from TR1.
- Moved Xiaogang Zhang's elliptic integral code into the library, and brought them into line with the rest of the code.
- Moved Hubert Holin's existing Boost.Math special functions into this library and brought them into line with the rest of the code.

Milestone 2: Released September 10th 2006

- Implement preview release of the statistical distributions.
- Added statistical distributions tutorial.
- Implemented root finding algorithms.
- Implemented the inverses of the incomplete gamma and beta functions.
- Rewrite erf/erfc as rational approximations (valid to 128-bit precision).
- Integrated the statistical results generated from the test data with Boost.Test: uses a database of expected results, indexed by test, floating point type, platform, and compiler.
- Improved lgamma near 1 and 2 (rational approximations).
- Improved erf/erfc inverses (rational approximations).
- Implemented Rational function generation (the Remez method).

Milestone 1: Released March 31st 2006

- Implement gamma/beta/erf functions along with their incomplete counterparts.
- Generate high quality test data, against which future improvements can be judged.
- Provide tools for the evaluation of infinite series, continued fractions, and rational functions.

- Provide tools for testing against tabulated test data, and collecting statistics on error rates.
- Provide sufficient docs for people to be able to find their way around the library.

SVN Revisions:

Sandbox and trunk last synchonised at revision: .

Known Issues, and TODO List

Predominantly this is a TODO list, or a list of possible future enhancements. Items labled "High Priority" effect the proper functioning of the component, and should be fixed as soon as possible. Items labled "Medium Priority" are desirable enhancements, often pertaining to the performance of the component, but do not effect it's accuracy or functionality. Items labled "Low Priority" should probably be investigated at some point. Such classifications are obviously highly subjective.

If you don't see a component listed here, then we don't have any known issues with it.

tgamma

- Can the Lanczos approximation be optimized any further? (low priority)

Incomplete Beta

- Investigate Didonato and Morris' asymptotic expansion for large a and b (medium priority).

Inverse Gamma

- Investigate whether we can skip iteration altogether if the first approximation is good enough (Medium Priority).

Polynomials

- The Legendre and Laguerre Polynomials have surprisingly different error rates on different platforms, considering they are evaluated with only basic arithmetic operations. Maybe this is telling us something, or maybe not (Low Priority).

Elliptic Integrals

- [para Carlson's algorithms (mainly R_J) are somewhat prone to internal overflow/underflow when the arguments are very large or small. The homogeneity relations:] [para R_F(ka, kb, kc) = $k^{-1/2}$ R_F(a, b, c)] [para and] [para R_J(ka, kb, kc, kr) = $k^{-3/2}$ R_J(a, b, c, r)] [para could be used to sidestep trouble here: provided the problem domains can be accurately identified. (Medium Priority).]

- There are a several other integrals: Bulirsch's *el* functions that could be implemented using Carlson's integrals (Low Priority).

- The integrals K(k) and E(k) could be implemented using rational approximations (both for efficiency and accuracy), assuming we can find them. (Medium Priority).

Owen's T Function

There is a problem area at arbitrary precision when *a* is very close to 1. However, note that the value for *T(h, 1)* is well known and easy to compute, and if we replaced the a^k terms in series T1, T2 or T4 by $(a^k - 1)$ then we would have the difference between *T(h, a)* and *T(h, 1)*. Unfortunately this doesn't improve the convergence of those series in that area. It certainly looks as though a new series in terms of $(1-a)^k$ is both possible and desirable in this area, but it remains elusive at present.

Statistical distributions

- Student's t Perhaps switch to normal distribution as a better approximation for very large degrees of freedom?

Feature Requests

The following table lists distributions that are found in other packages but which are not yet present here, the more frequently the distribution is found, the higher the priority for implementing it:

Distribution	R	Mathematica 6	NIST	Regress+	Matlab
Geometric	X	X	-	-	X
Multinomial	X	-	-	-	X
Tukey Lambda	X	-	X	-	-
Half Normal / Folded Normal	-	X	-	X	-
Chi	-	X	-	X	-
Gumbel	-	X	-	X	-
Discrete Uniform	-	X	-	-	X
Log Series	-	X	-	X	-
Nakagami (generalised Chi)	-	-	-	X	X
Log Logistic	-	-	-	-	X
Tukey (Studentized range)	X	-	-	-	-
Wilcoxon rank sum	X	-	-	-	-
Wincoxon signed rank	X	-	-	-	-
Non-central Beta	X	-	-	-	-
Maxwell	-	X	-	-	-
Beta-Binomial	-	X	-	-	-
Beta-negative Binomial	-	X	-	-	-
Zipf	-	X	-	-	-
Birnbaum-Saunders / Fatigue Life	-	-	X	-	-
Double Exponential	-	-	X	-	-
Power Normal	-	-	X	-	-
Power Lognormal	-	-	X	-	-
Cosine	-	-	-	X	-
Double Gamma	-	-	-	X	-

Distribution	R	Mathematica 6	NIST	Regress+	Matlab
Double Weibul	-	-	-	X	-
Hyperbolic Secant	-	-	-	X	-
Semicircular	-	-	-	X	-
Bradford	-	-	-	X	-
Birr / Fisk	-	-	-	X	-
Reciprocal	-	-	-	X	-
Kolmogorov Distribution	-	-	-	-	-

Also asked for more than once:

- Add support for interpolated distributions, possibly combine with numeric integration and differentiation.
- Add support for bivariate and multivariate distributions: most especially the normal.
- Add support for the log of the cdf and pdf: this is mainly a performance optimisation since we can avoid some special function calls for some distributions by returning the log of the result.

Credits and Acknowledgements

Hubert Holin started the Boost.Math library. The Quaternions, Octonions, inverse hyperbolic functions, and the sinus cardinal functions are his.

Daryle Walker wrote the integer gcd and lcm functions.

John Maddock started the special functions, the beta, gamma, erf, polynomial, and factorial functions are his, as is the "Toolkit" section, and many of the statistical distributions.

Paul A. Bristow threw down the challenge in A Proposal to add Mathematical Functions for Statistics to the C++ Standard Library to add the key math functions, especially those essential for statistics. After JM accepted and solved the difficult problems, not only numerically, but in full C++ template style, PAB implemented a few of the statistical distributions. PAB also tirelessly proof-read everything that JM threw at him (so that all remaining editorial mistakes are his fault).

Xiaogang Zhang worked on the Bessel functions and elliptic integrals for his Google Summer of Code project 2006.

Bruno Lalande submitted the "compile time power of a runtime base" code.

Johan Råde wrote the optimised floating-point classification and manipulation code, and nonfinite facets to permit C99 output of infinities and NaNs. (nonfinite facets were not added until Boost 1.47 but had been in use with Boost.Spirit). This library was based on a suggestion from Robert Ramey, author of Boost.Serialization. Paul A. Bristow expressed the need for better handling of Input & Output of NaN and infinity for the C++ Standard Library and suggested following the C99 format.

Antony Polukhin improved lexical cast avoiding stringstream so that it was no longer necessary to use a globale C99 facet to handle nonfinites.

Håkan Ardö, Boris Gubenko, John Maddock, Markus Schöpflin and Olivier Verdier tested the floating-point library and Martin Bonner, Peter Dimov and John Maddock provided valuable advice.

Gautam Sewani coded the logistic distribution as part of a Google Summer of Code project 2008.

M. A. (Thijs) van den Berg coded the Laplace distribution. (Thijs has also threatened to implement some multivariate distributions).

Thomas Mang requested the inverse gamma in chi squared distributions for Bayesian applications and helped in their implementation, and provided a nice example of their use.

Professor Nico Temme for advice on the inverse incomplete beta function.

Victor Shoup for NTL, without which it would have much more difficult to produce high accuracy constants, and especially the tables of accurate values for testing.

We are grateful to Joel Guzman for helping us stress-test his Boost.Quickbook program used to generate the html and pdf versions of this document, adding several new features en route.

Plots of the functions and distributions were prepared in W3C standard Scalable Vector Graphic (SVG) format using a program created by Jacob Voytko during a Google Summer of Code (2007). From 2012, the latest versions of all Internet Browsers have support for rendering SVG (with varying quality). Older versions, especially (Microsoft Internet Explorer (before IE 9) lack native SVG support but can be made to work with Adobe's free SVG viewer plugin). The SVG files can be converted to JPEG or PNG using Inkscape.

We are also indebted to Matthias Schabel for managing the formal Boost-review of this library, and to all the reviewers - including Guillaume Melquiond, Arnaldur Gylfason, John Phillips, Stephan Tolksdorf and Jeff Garland - for their many helpful comments.

Thanks to Mark Coleman and Georgi Boshnakov for spot test values from Wolfram Mathematica, and of course, to Eric Weisstein for nurturing Wolfram MathWorld, an invaluable resource.

The Skew-normal distribution and Owen's t function were written by Benjamin Sobotta.

We thank Thomas Mang for persuading us to allow t distributions to have infinite degrees of freedom and contributing to some long discussions about how to improve accuracy for large non-centrality and/or large degrees of freedom.

Christopher Kormanyos wrote the e_float multiprecision library TOMS Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations which formed the basis for the Boost.Multiprecision library which now can be used to allow most functions and distributions to be computed up to a precision of the users' choice, no longer restricted to built-in floating-point types like double. (And thanks to Topher Cooper for bring Christopher's e_float to our attention).

Christopher Kormanyos wrote some examples for using Boost.Multiprecision, and added methods for finding zeros of Bessel Functions.

Marco Guazzone provided the hyper-geometric distribution.

Rocco Romeo has found numerous small bugs and generally stress tested the special functions code to near destruction!

Jeremy William Murphy added polynomial arithmetic tools.

Thomas Luu provided improvements to the quantile of the non-central chi squared distribution quantile. and his thesis * Fast and accurate parallel computation of quantile functions for random number generation, 2016.

and his paper

Luu, Thomas; (2015), Efficient and Accurate Parallel Inversion of the Gamma Distribution, SIAM Journal on Scientific Computing , 37 (1) C122 - C141, https://doi.org/10.1137/14095875X.

These also promise to help improve algorithms for computation of quantile of several disitributions, especially for parallel computation using GPUs.

Indexes

Function Index

Class Index

Typedef Index

Macro Index

Index