Math Toolkit

John Maddock
Paul A. Bristow
Hubert Holin
Xiaogang Zhang

Table of Contents

Overview

About the Math Toolkit

This library is divided into three interconnected parts:

Statistical Distributions

Provide 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, although they are optimised for use with types with known-about significand (or mantissa) sizes: typically `float`, `double` or `long double`.

Implementation Toolkit

Provides many of the tools required to implement mathematical special functions: hopefully the presence of these will encourage other authors to contribute more special function implementations in the future. These tools are currently considered experimental: they are "exposed implementation details" whose interfaces and/or implementations may change.

There are helpers for the evaluation of infinite series, continued fractions and rational approximations.

There is a fairly comprehensive set of root finding and function minimisation algorithms: the root finding algorithms are both with and without derivative support.

A Remez algorithm implementation allows for the locating of minimax rational approximations.

There are also (experimental) classes for the manipulation of polynomials, for testing a special function against tabulated test data, and for the rapid generation of test data and/or data for output to an external graphing application.

Navigation

Used in combination with the configured browser key (usually Alt), the following keys act as handy shortcuts for common navigation tasks.

Shortcuts

p - Previous page

n - Next page

h - home

u - Up

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 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 various .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);
```

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`.

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?

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.

Evaluation Error — Occurs when an internal error occured 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.

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 table shows all the permutations of errors and actions, with the default action for each error shown in bold:

Table 1. Possible Error Conditions, and their Defaults

Error Type	**throw_on_error**	**errno_on_error**	**ignore_error**	**user_error**
Domain Error	**Throws `std::domain_error`**	Sets `::errno` to `EDOM` and returns `std::numeric_limits<T>::quiet_NaN()`	Returns `std::numeric_limits<T>::quiet_NaN()`	Returns the result `boost::math::policies::user_` `main_error`: this function must be defined b user.
Pole Error	**Throws `std::domain_error`**	Sets `::errno` to `EDOM` and returns `std::numeric_limits<T>::quiet_NaN()`	Returns `std::numeric_limits<T>::quiet_NaN()`	Returns the result `boost::math::policies::user_pole_` `ror`: this function must be defined by the user.
Overflow Error	**Throws `std::overflow_error`**	Sets `::errno` to `ERANGE` and returns `std::numeric_limits<T>::infinity()`	Returns `std::numeric_limits<T>::infinity()`	Returns the result `boost::math::policies::user_ov` `flow_error`: this function must be defined b user.
Underflow Error	Throws `std::underflow_error`	Sets `::errno` to `ERANGE` and returns 0.	**Returns 0**	Returns the result `boost::math::policies::user_und` `flow_error`: this function must be defined b user.
Denorm Error	Throws `std::underflow_error`	Sets `::errno` to `ERANGE` and returns the denormalised value.	**Returns the denormalised value.**	Returns the result `boost::math::policies::user_` `norm_error`: this function must be defined b user.
Evaluation Error	**Throws `boost::math::evaluation_error`**	Sets `::errno` to `EDOM` and returns `std::numeric_limits<T>::infinity()`.	Returns `std::numeric_limits<T>::infinity()`.	Returns the result `boost::math::policies::user_eva` `ation_error`: this function must be defined b user.

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.

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.

[hreading 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.

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.

Configuration and 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 prefered method for installation wide policies.
- At your chosen namespace scope for distributions and/or functions: this is the prefered 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 provided the function and class templates in the library are instantiated with built-in floating point types: i.e. the types `float`, `double` and `long double`.

However, the library **is not thread safe** when used with user-defined (i.e. class type) numeric types.

The reason for the latter limitation is the need to initialise symbolic constants using constructs such as:

```
static const T coefficient_array = { ... list of values ... };
```

Which is always thread safe when T is a built-in floating point type, but not when T is a user defined type: as in this case there is a need for T's constructors to be run, leading to potential race conditions.

This limitation may be addressed in a future release.

Performance

By and large the performance of this library should be acceptable for most needs. However, you should note that the library's primary emphasis is on accuracy and numerical stability, and *not* speed.

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 FORTAN 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 two 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.

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.

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

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/distributions/my_distribution.hpp> or the "include everything" header: <boost/math/distributions.hpp>.

For example, to use the Students-t distribution include either <boost/math/distributions/students_t.hpp> or <boost/math/distributions.hpp>

Distributions are Objects

Each kind of distribution in this library is a class type.

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;

// 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 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

Random variates and distribution parameters are conventionally distinguished (for example in Wikipedia and Wolfram MathWorld by placing a semi-colon (or sometimes vertical bar) after the random variate (whose value you 'choose'), to separate the variate from the parameter(s) that defines the shape of the distribution.

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 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. TODO: fix this sentence: Mathematically, the functions may make sense with an (+ or -) infinite value, but this implementation limits random variates to finite values from the `max` to `min` for the `RealType`. (See Handling of Floating-Point Infinity for rationale).

The range of random variate values that is permitted and supported can be tested by using two functions `range` and `support`.

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

Often you don't want the value of the CDF, but its complement, which is to say `1-p` rather than `p`. You could calculate the CDF and subtract it from `1`, but if `p` is very close to `1` then cancellation error will cause you to lose significant digits. In extreme cases, `p` may actually be equal to `1`, even though the true value of the complement is non-zero.

See also *"Why 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, 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.

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 Example

See distribution_construction.cpp for full source code.

The structure of distributions is rather different from some other statistical libraries, for example in less object-oriented language like FORTRAN and C, that provide a few arguments to each free function. This library 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.

First we need some includes to access the negative binomial distribution (and the binomial, beta and gamma too).

```
#include <boost/math/distributions/negative_binomial.hpp> // for negative_binomial_distributi
  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.
```

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 by writing

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

or

```
using boost::math::negative_binomial_distribution;
```

we can 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 do
```

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 takes place, so you can use integers and floats:

```
negative_binomial mydist11(5, 0.4); // Using provided typedef double, int and double argument
```

This is probably the most common usage.

```
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. 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:
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(8., 0.25);

// Explicit long double precision:
negative_binomial_distribution<long double>   mydist7(8., 0.25);
```

And if you have your own RealType called MyFPType, for example NTL RR (an arbitrary precision type), then we can write:

```
negative_binomial_distribution<MyFPType>  mydist6(8, 1); // Integer arguments -> MyFPType.
```

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, 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 and standard deviation thus:

```
normal_distribution(RealType mean = 0, RealType sd = 1);
```

So in this case we can write:

```
  using boost::math::normal;

  normal norm1;       // Standard normal distribution.
  normal norm2(2);    // Mean = 2, std deviation = 1.
  normal norm3(2, 3); // Mean = 2, std deviation = 3.

  return 0;
}  // int main()
```

There is no useful output from this program, of course.

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, *[alpha]* 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 > significance level: `cdf(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(complement(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             REJECTED
   Mean  > 38.900             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)
_______________________________________________________________
    50.000               2               3
    75.000               4               5
    90.000               8              10
    95.000              12              14
    99.000              21              23
    99.900              36              38
    99.990              51              54
    99.999              67              69
```

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:

$$t = \frac{\mathrm{Sm}_1 - \mathrm{Sm}_2}{s_p \sqrt{\frac{1}{\mathrm{Sn}_1} + \frac{1}{\mathrm{Sn}_2}}}$$

$$s_p = \sqrt{\frac{(\mathrm{Sn}_1 - 1)\,\mathrm{Sd}_1^2 + (\mathrm{Sn}_2 - 1)\,\mathrm{Sd}_2^2}{\mathrm{Sn}_1 + \mathrm{Sn}_2 - 2}}$$

$$\nu = \mathrm{Sn}_1 + \mathrm{Sn}_2 - 2$$

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" << "=  " << v << "\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 less 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 te 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                                          =  20.14458
Sample 1 Standard Deviation                            =  6.41470
Number of Observations (Sample 2)                      =  79
Sample 2 Mean                                          =  30.48101
Sample 2 Standard Deviation                            =  6.10771
Degrees of Freedom                                     =  326.00000
Pooled Standard Deviation                              =  326.00000
T Statistic                                            =  -12.62059
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 the Sample 1 Mean is greater than that for Sample 2: in this case Sample 1 represents the miles per gallon for US cars, and Sample 2 the miles per gallon for Japanese 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:

$$\nu = \frac{\left(\frac{Sd_1^2}{Sn_1} + \frac{Sd_2^2}{Sn_2}\right)^2}{\frac{\left(\frac{Sd_1^2}{Sn_1}\right)^2}{(Sn_1 - 1)} + \frac{\left(\frac{Sd_2^2}{Sn_2}\right)^2}{(Sn_2 - 1)}}$$

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 is in chi_square_std_deviation_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.

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 < $\chi^2_{(1\text{-alpha}/2;\ N\text{-}1)}$ or T > $\chi^2_{(\text{alpha}/2;\ N\text{-}1)}$
The alternative hypothesis: there is a difference in standard deviation from the specified value	Reject if $\chi^2_{(1\text{-alpha}/2;\ N\text{-}1)}$ >= T >= $\chi^2_{(\text{alpha}/2;\ N\text{-}1)}$
The alternative hypothesis: the standard deviation is less than the specified value	Reject if $\chi^2_{(1\text{-alpha};\ N\text{-}1)}$ <= T
The alternative hypothesis: the standard deviation is greater than the specified value	Reject if $\chi^2_{(\text{alpha};\ N\text{-}1)}$ >= T

Where $\chi^2_{(\text{alpha};\ N\text{-}1)}$ is the upper critical value of the Chi Squared distribution, and $\chi^2_{(1\text{-alpha};\ N\text{-}1)}$ 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 <= $F_{(1-alpha/2; N1-1, N2-1)}$ or F >= $F_{(alpha/2; N1-1, N2-1)}$
The alternative hypothesis: there is a difference in means (two sided test)	Reject if $F_{(1-alpha/2; N1-1, N2-1)}$ <= F <= $F_{(alpha/2; N1-1, N2-1)}$
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.

```
}
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;
```

The number of correct answers, X, is distributed as a binomial random variable with binomial distribution parameters: questions n = 16 and success fraction probability p = 0.25. 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 = (double)answers / (double)questions; // If a random guess.
// Caution:  = answers / questions would be zero (because they are integers)!
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, 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 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 complement function. Why complements?

```
cout << "Probability of getting > 10 right (to pass) is " << cdf(complement(quiz, 10)) << end
```

```
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 i
```

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 gues
  <<  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 gues
  <<  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 gues
  <<  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 gues
  <<  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 answers.
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 discrete 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 Understanding Discrete Quantile Policy.

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
```

#define BOOST_MATH_DISCRETE_QUANTILE_POLICY real

```
at the head of the program would make this policy apply
```

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_distribution<>::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(successes)
```

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 αis 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).

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 occurence 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(successe

   // 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
  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 fracti

  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 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.

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 = " << failures;
  cout << ",   Success fraction = " << 100 * p << "%" << endl;
   // Print table header:
   cout << "\n\n"
           "____________________________\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 http://en.wikipedia.org/wiki/Bernoulli_distribution 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 = successes + 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 mass (density) function (pdf or pmf) 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. 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 fini
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 (int 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 - sa
}
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 tabl
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::noshowpo
#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.
```

```
// 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 (int i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
  cout << setw(15) << alpha[i] << setw(15) << alpha[i] /2 << setw(10) << quantile(complement(
  // Use quantile(complement(s, alpha[i]/2)) to avoid potential loss of accuracy from quantil
}
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." << en
```

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) 1.64
// Rearranging, we can directly calculate the required standard deviation:
double sd95 = 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;
```

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:

```
// 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>
 using std::exception;
```

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;
      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;
         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!
errno is set to: 0
```

And finally let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

Which gives the output:

```
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;
  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 complement of our probability, we should not even think of writing `find_location<normal>(z, 1 - p, sd)`, but, to avoid loss of accuracy, use the complement version.

```
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 instead, use the complements version.

```
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,
   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::noshowp
#include <iomanip>
  using std::setw; using std::setprecision;
#include <limits>
  using std::numeric_limits;
```

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.

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 <<
// 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 meamn 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. 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())); " << ss
// 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.

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 it's 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. This 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

- cdf.
- cdf complement.
- chf.
- hazard.
- kurtosis.
- kurtosis_excess
- mean.
- median.
- mode.
- pdf.
- range.
- quantile.

- quantile from the complement.
- skewness.
- standard_deviation.
- support.
- variance.

Conceptual Index

- Complement of the Cumulative Distribution Function.
- Cumulative Distribution Function.
- Cumulative Hazard Function.
- Inverse Cumulative Distribution Function.
- Inverse Survival Function.
- Hazard Function
- Lower Critical Value.
- 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.

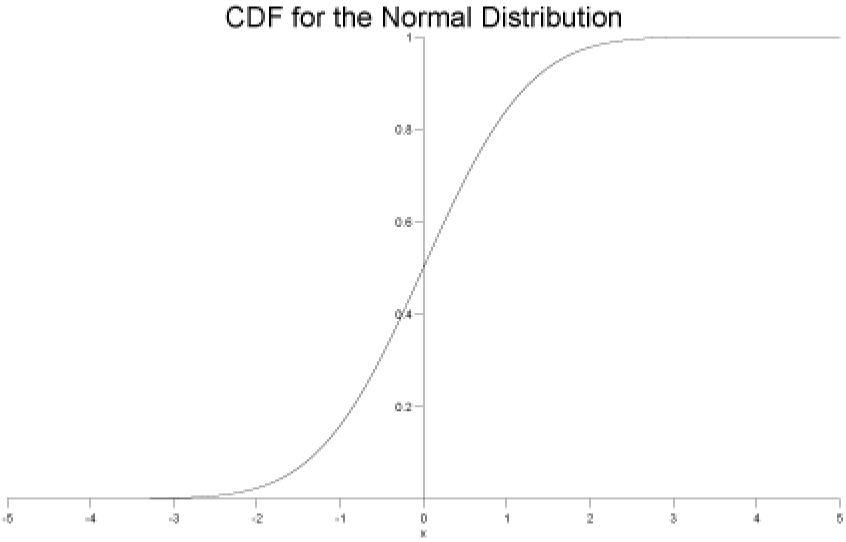
Cumulative Distribution Function

```
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.

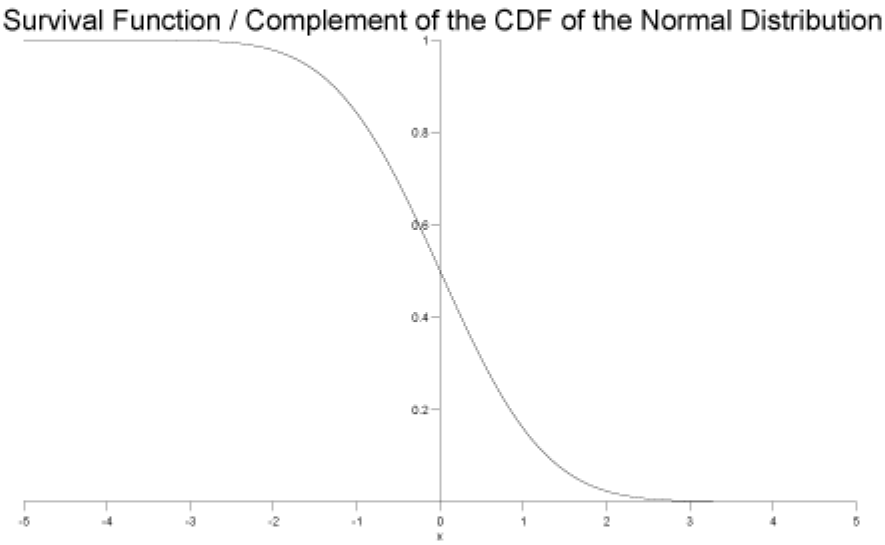
This is also known as the survival function.

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:



Survival Function / Complement of the CDF of 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.

$$\mathrm{hazard}(x) \quad = \quad h(x) \quad = \quad \frac{\mathrm{pdf}(x)}{1 - \mathrm{cdf}(x)}$$

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.

$$\mathrm{chf}(dist, x) \quad = \quad H(x) \quad = \quad \int_{-\infty}^{x} h(\mu)\, d\mu$$

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

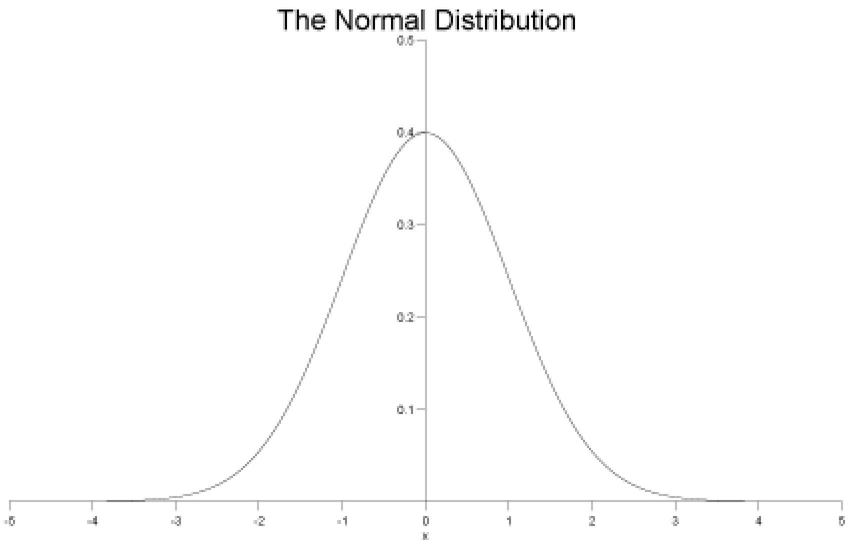
```
template <class RealType, class Policy>
RealType pdf(const Distribution-Type<RealType, Policy>& dist, const RealType& x);
```

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

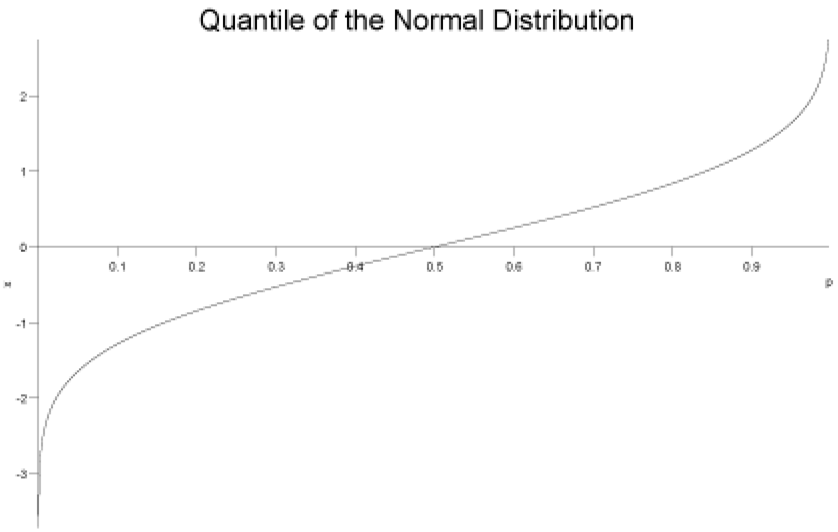
```
template <class RealType, class Policy>
RealType quantile(const Distribution-Type<RealType, Policy>& dist, const RealType& p);
```

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 a *percentile*, 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.

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;
```

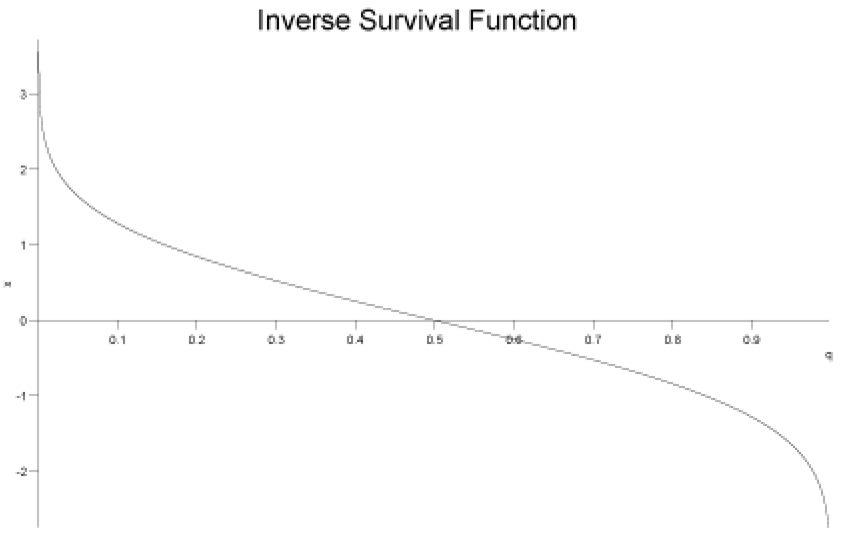
The function computes a value *x* such that `cdf(complement(dist, x)) == q` where *q* is complement of the probability.

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 = β_2= μ_4 / μ_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 = γ_2= μ_4 / μ_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

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

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
```

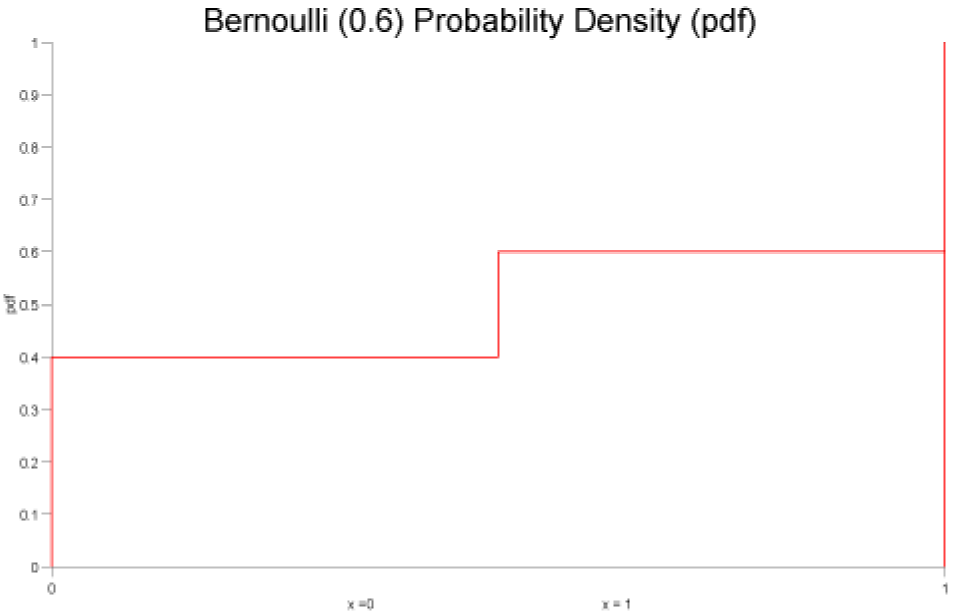
The Bernoulli distribution is a discrete distribution of the outcome of a single trial with only two results, 0 (failure) or 1 (success), with a probability of success p.

The Bernoulli distribution is the simplest building block on which other discrete distributions of sequences of independent Bernoulli trials can be based.

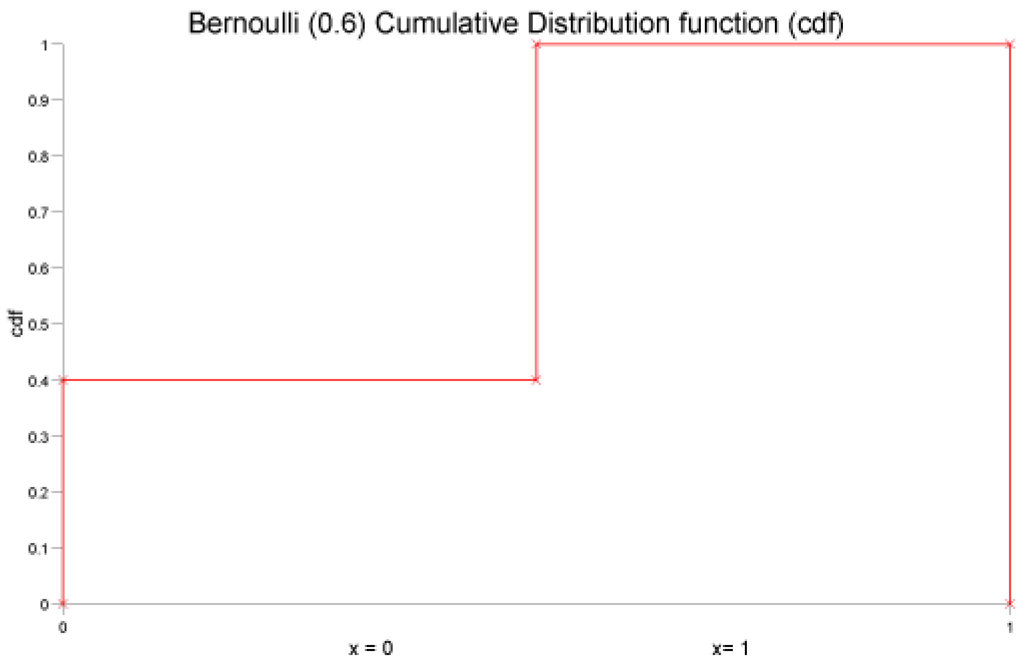
The Bernoulli is the binomial distribution (k = 1, p) with only one trial.

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 $q = 1\text{-}p$. *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 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.

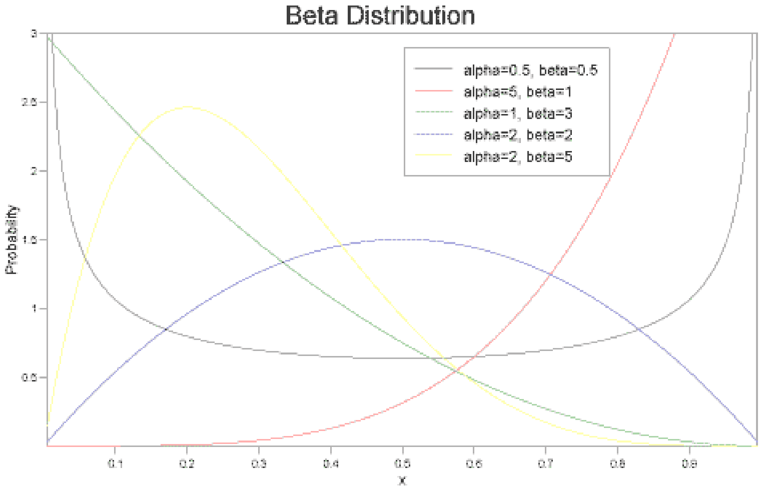
How the beta distribution is used for Bayesian analysis of one parameter models is discussed by Jeff Grynaviski.

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 $B(\alpha, \beta)$ 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 $\alpha = \beta = 2$ (blue line) is dome-shaped, and might be approximated by a symmetrical triangular distribution.



If α = β = 1, then it is a uniform distribution, equal to unity in the entire interval x = 0 to 1. If α and β 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 or β from presumed-known mean and variance.

The other pair estimates either α or β from the cdf and x.

It is also possible to estimate α and β 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 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 and β are integers.

With integer values of α and β 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 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 cloppper_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.

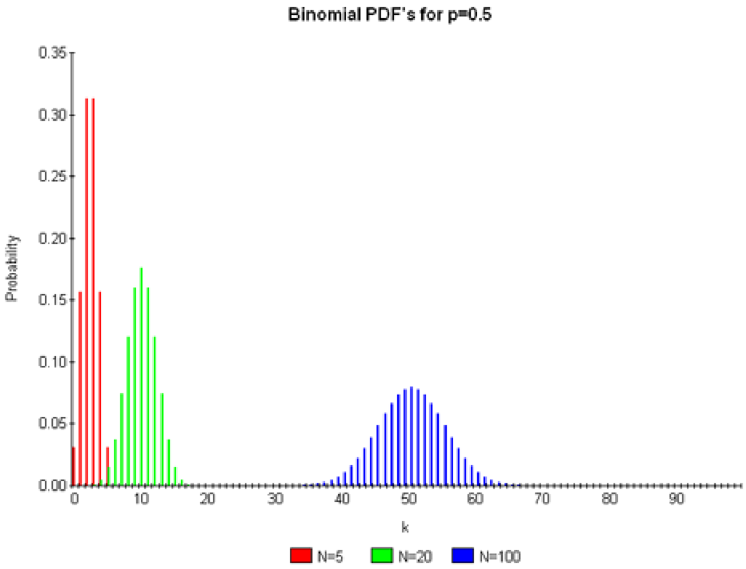
Note

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.

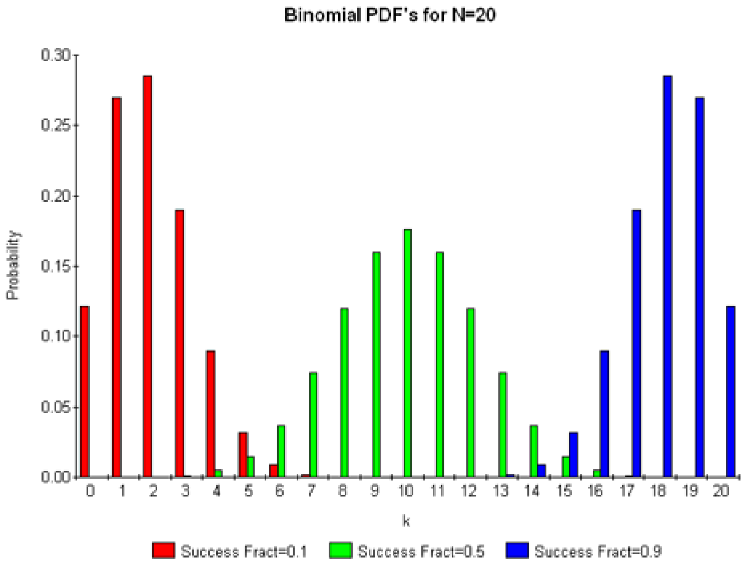
The PDF for the binomial distribution is given by:

$$\begin{aligned} f(k; n, p) &= {}_nC_k\, p^k (1-p)^{n-k} \\ &= \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} \end{aligned}$$

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 2. 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	The **greatest** number of successes that may be observed from n 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 result. For example: `quantile(binomial(n, p), P)`
Quantile from the complement of the probability	The **smallest** number of successes that may be observed from n 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 result. For example: ``quantile(complement(binomial(n, p), P))` ``

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: $\begin{aligned} \mathrm{f}(k; n, p) &= {}_nC_k\, p^k (1-p)^{n-k} \\ &= \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} \\ &= \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)} p^k (1-p)^{n-k} \\ &= \frac{p^k (1-p)^{n-k}}{B(k+1, n-k+1)(n+1)} \end{aligned}$ Which can be evaluated as `ibeta_derivative(k+1, n-k+1, p) / (n+1)`

Function	Implementation Notes
	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 Algorithm 748).
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)`
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;
};
```

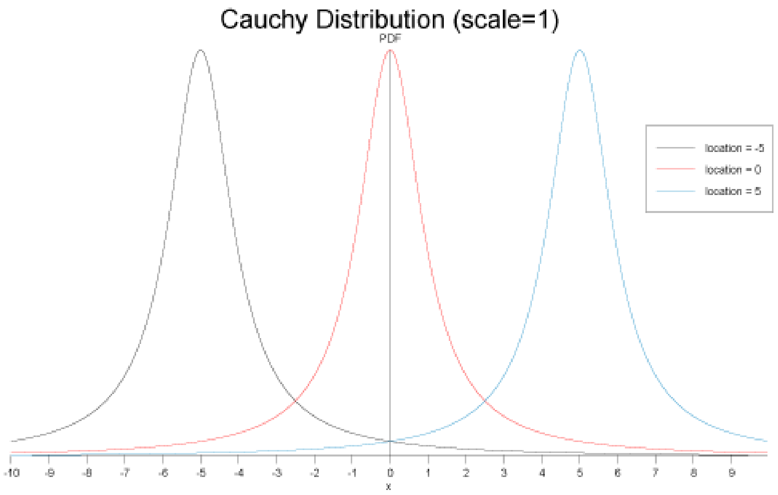
The Cauchy-Lorentz distribution is named after Augustin Cauchy and Hendrik Lorentz. It is a continuous probability distribution with probability distribution function PDF given by:

$$\mathrm{f}\left(x;\, x_0,\, \gamma\right) \quad = \quad \frac{1}{\pi}\left(\frac{\gamma}{(x - x_0)^2 + \gamma^2}\right)$$

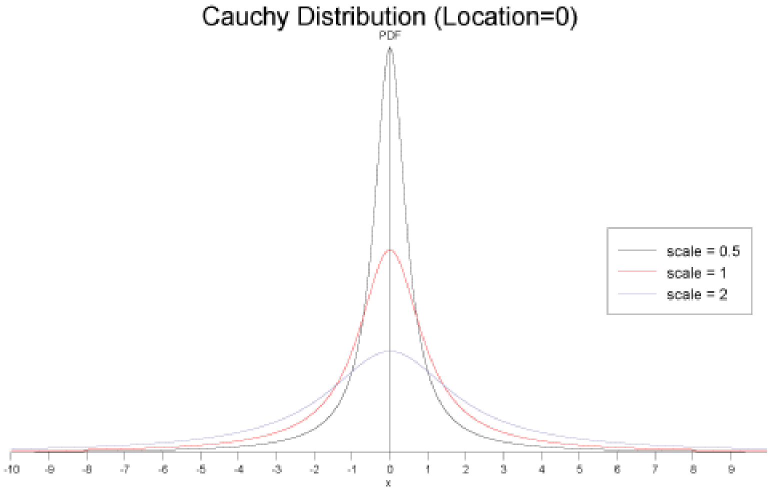
The location parameter x_0 is the location of the peak of the distribution (the mode of the distribution), while the scale parameter γ specifies half the width of the PDF at half the maximum height. If the location is zero, and the scale 1, then the result is a standard Cauchy distribution.

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 = $1 / (\pi * \gamma * (1 + ((x - x_0) / \gamma)^2)$
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 $(x - x_0) / \gamma$ 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 it's complement. First the argument *p* is reduced to the range [-0.5, 0.5], then the relation $x = x_0 \pm \gamma / \tan(\pi * p)$ 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

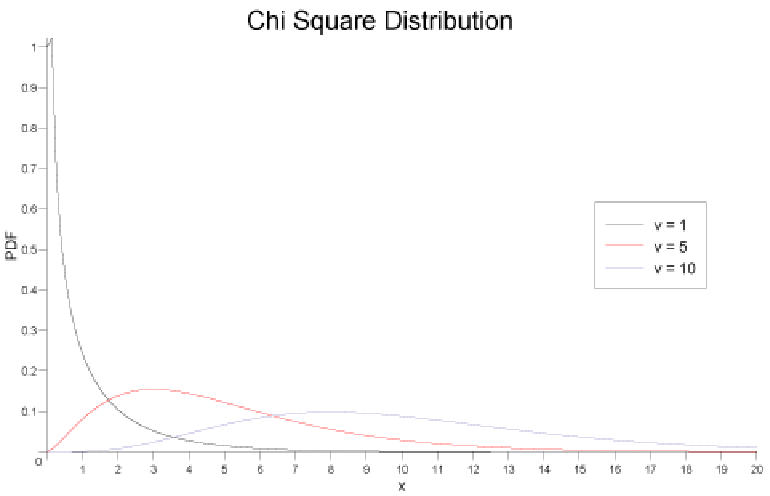
```
#include <boost/math/distributions/chi_squared.hpp>


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.

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>
```

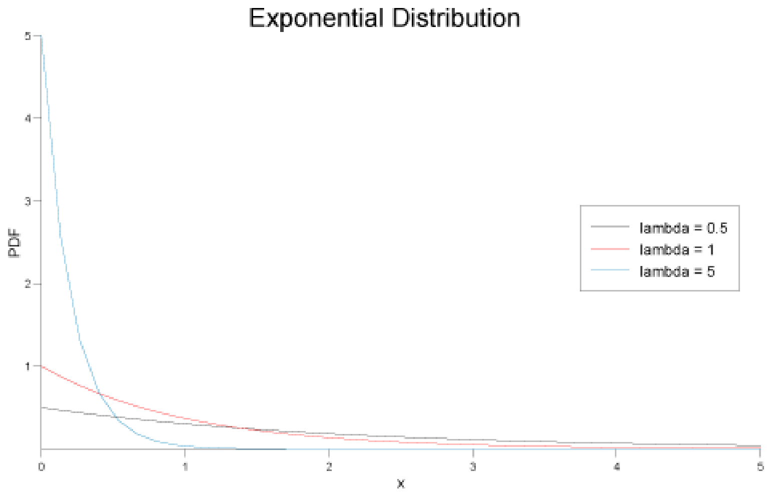
```
template <class RealType = double,
          class Policy   = policies::policy<> >
class exponential_distribution;

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:

$$f(x) \quad = \quad \lambda e^{-\lambda x}$$

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

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.

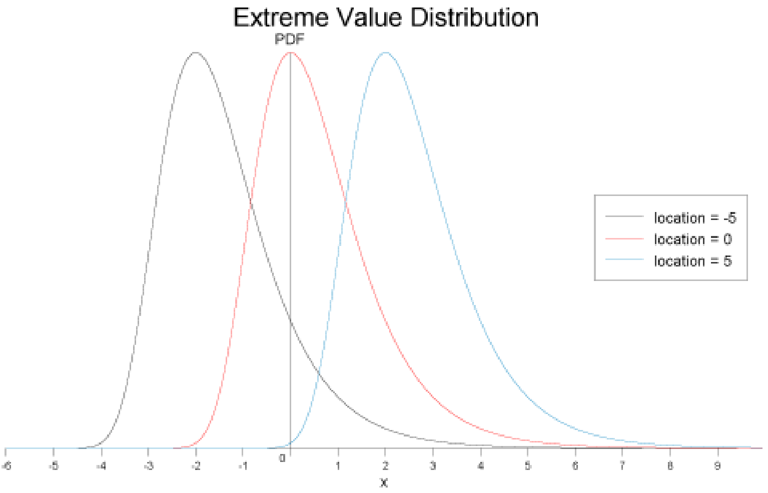
The distribution has a PDF given by:

$f(x) = (1/scale)\ e^{-(x-location)/scale}\ e^{-e^{-(x-location)/scale}}$

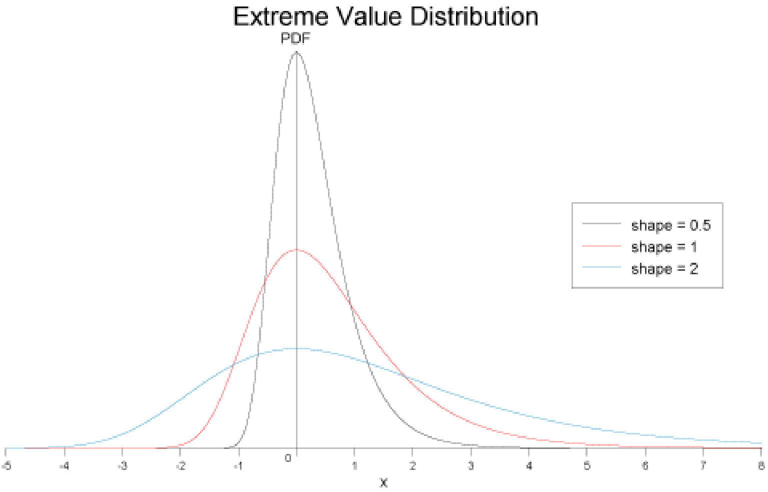
Which in the standard case (scale = 1, location = 0) reduces to:

$f(x) = e^{-x}e^{-e^{-x}}$

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
```

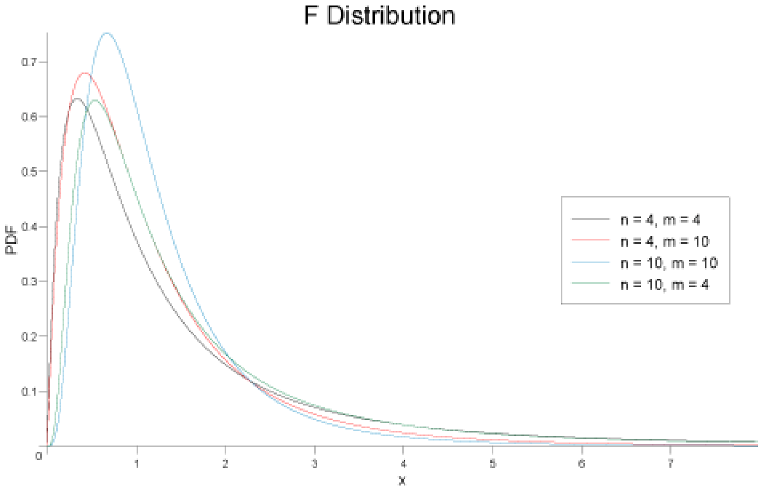
The F distribution is a continuous distribution that arises when testing whether two samples have the same variance. If χ^2_m and χ^2_n are independent variates each distributed as Chi-Squared with *m* and *n* degrees of freedom, then the test statistic:

$F_{n,m} = (\chi^2_n / n) / (\chi^2_m / m)$

Is distributed over the range [0, ∞] with an F distribution, and has the PDF:

$$f_{n,m}(x) = \frac{m^{\frac{m}{2}} n^{\frac{n}{2}} x^{\frac{n}{2}-1}}{(m+n\,x)^{\frac{(n+m)}{2}} B\left(\frac{n}{2}, \frac{m}{2}\right)}$$

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 it's 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.
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)

Function	Implementation Notes
	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_distibution<double> gamma;
```

Instead if you want a double precision gamma distribution you can use

```
boost::gamma_distribution<>
```

For shape parameter *k* and scale parameter θ it is defined by the probability density function:

$$\mathrm{f}(x;\, k,\, \theta) \quad = \quad x^{k-1}\frac{e^{-\frac{x}{\theta}}}{\theta^{k}\Gamma(k)}$$

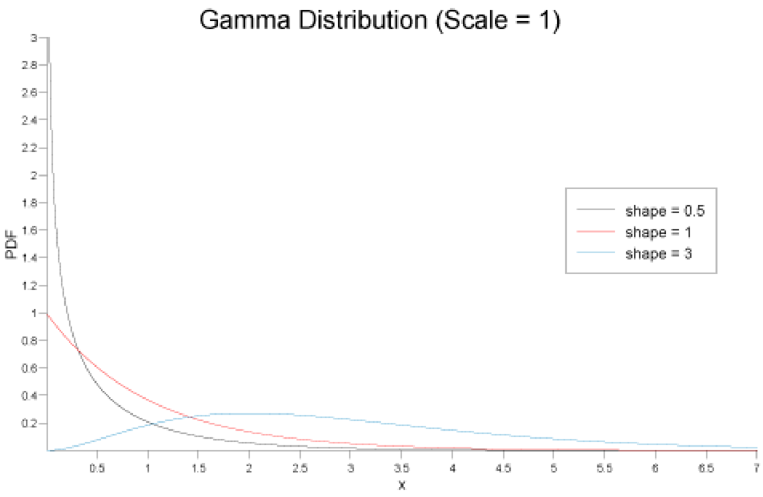
Sometimes an alternative formulation is used: given parameters α= k and β= 1 / θ, then the distribution can be defined by the PDF:

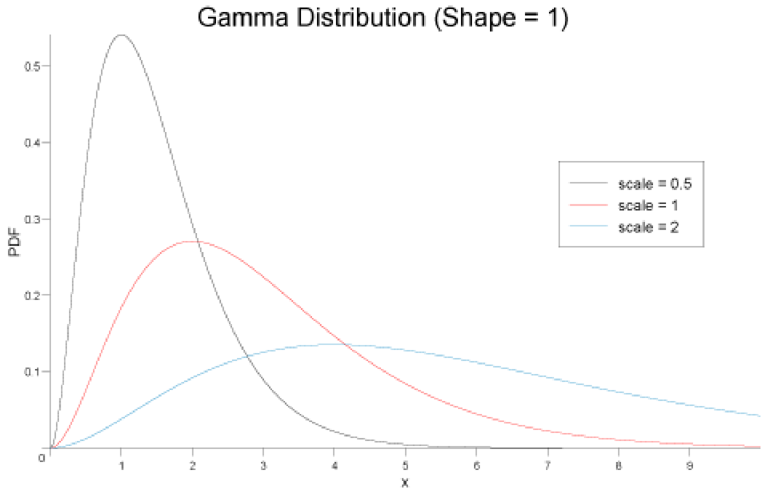
$$\mathrm{f}(x;\, \alpha,\, \beta) \quad = \quad x^{\alpha-1}\frac{\beta^{\alpha}e^{-\beta x}}{\Gamma(\alpha)}$$

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 lognormal 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 it's 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

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
```

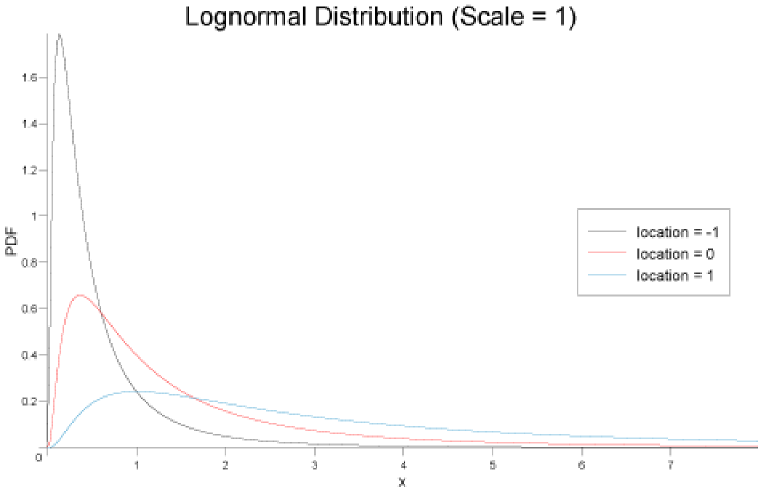
The lognormal distribution is the distribution that arises when the logarithm of the random variable is normally distributed. A lognormal distribution results when the variable is the product of a large number of independent, identically-distributed variables.

For location and scale parameters *m* and *s* it is defined by the probability density function:

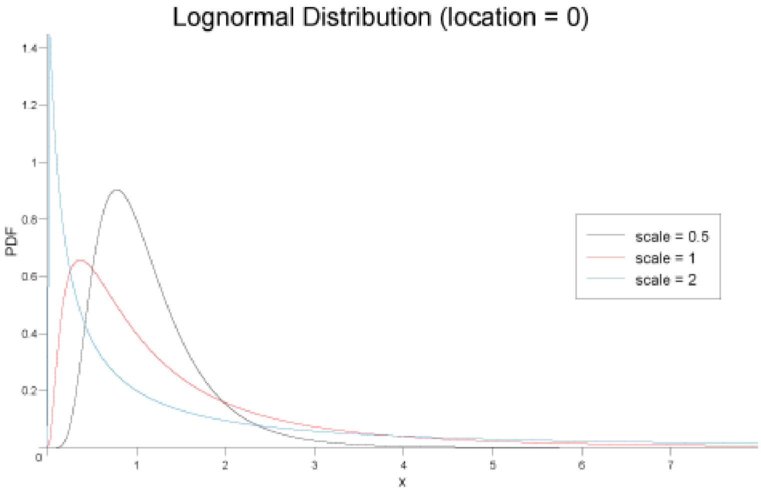
$$f(x) = \frac{1}{x s \sqrt{2\pi}} e^{\frac{-(\ln x - m)^2}{2 s^2}}$$

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 it's 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.

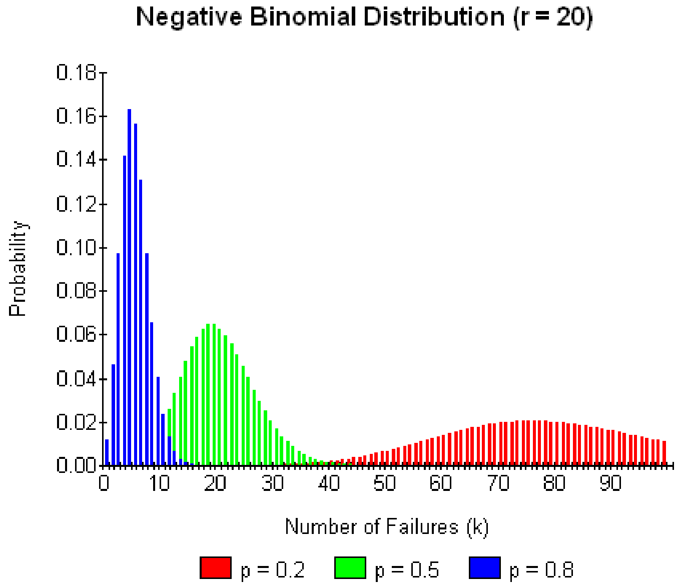
Note

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.

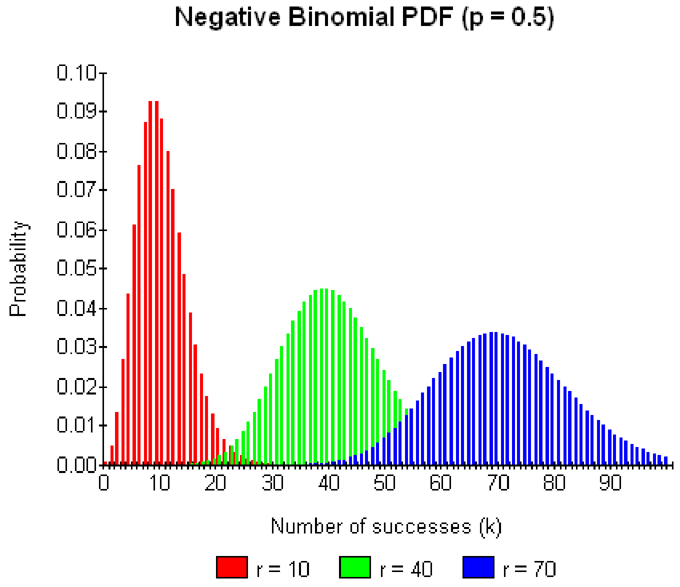
It has the PDF:

$$f(k, r, p) = \frac{\Gamma(r+k)}{k!\Gamma(r)} p^r (1-p)^k$$

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

poisson(λ) = lim $_{r \to \infty}$ negative_binomial(r, r / (λ + 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.

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_trials(0, 1.0-1.0/1000000, 0
```

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 3. 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_binomial(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)

Function	Implementation Notes
	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.

Normal (Gaussian) Distribution

```
#include <boost/math/distributions/normal.hpp>
```

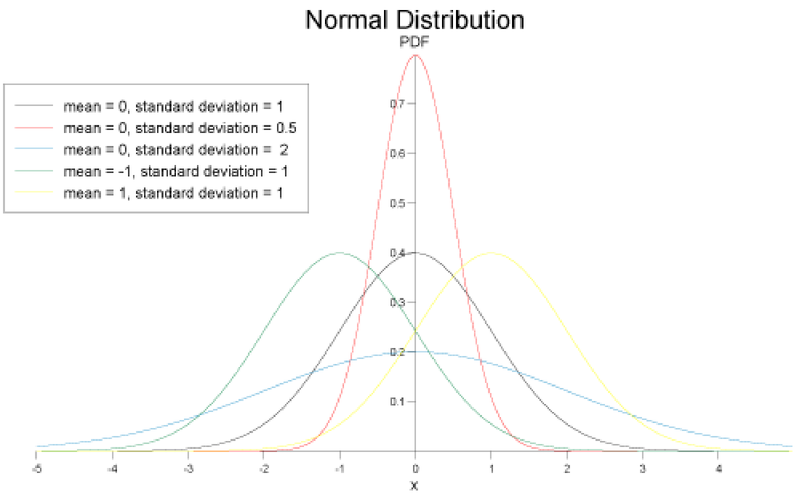
```
namespace boost{ namespace math{

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:

$$f(x;\,\mu,\,\sigma) \quad = \quad \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The variation the PDF with its parameters is illustrated in the following graph:



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.
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 location = 1, RealType shape = 1)
   // Accessors:
   RealType location()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 location parameter $\beta > 0$, and $\alpha > 0$.

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 shape parameter α:

Pareto pdf

Related distributions

Member Functions

```
pareto_distribution(RealType location = 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 location()const;
```

Returns the *location* 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 [location, ∞].

Accuracy

The pareto distribution is implemented in terms of the standard library `exp` functions plus expm1 and as such should have very low error rates except when probability is very close to unity.

Implementation

In the following table α is the shape parameter of the distribution, and β is its location 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: $x = \alpha / (1 - p)^{1/\beta}$
quantile from the complement	Using the relation: $x = \alpha / (q)^{1/\beta}$
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.

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.

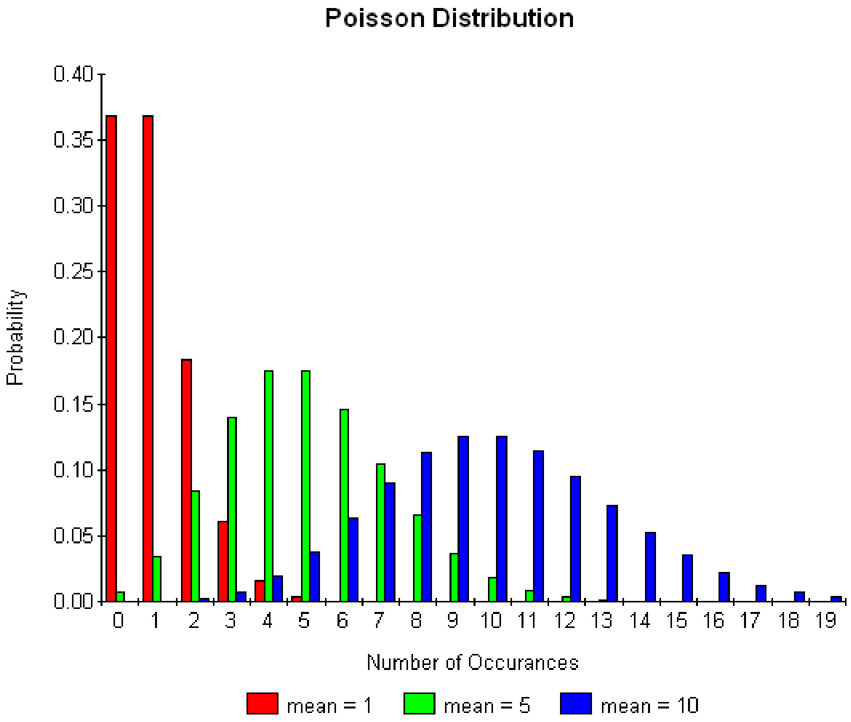
The distribution was discovered by Simé on-Denis Poisson (1781 to 1840).

It has the Probability Mass Function:

$$f(k; \lambda) = \frac{e^{-\lambda} \lambda^{k}}{k!}$$

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
```

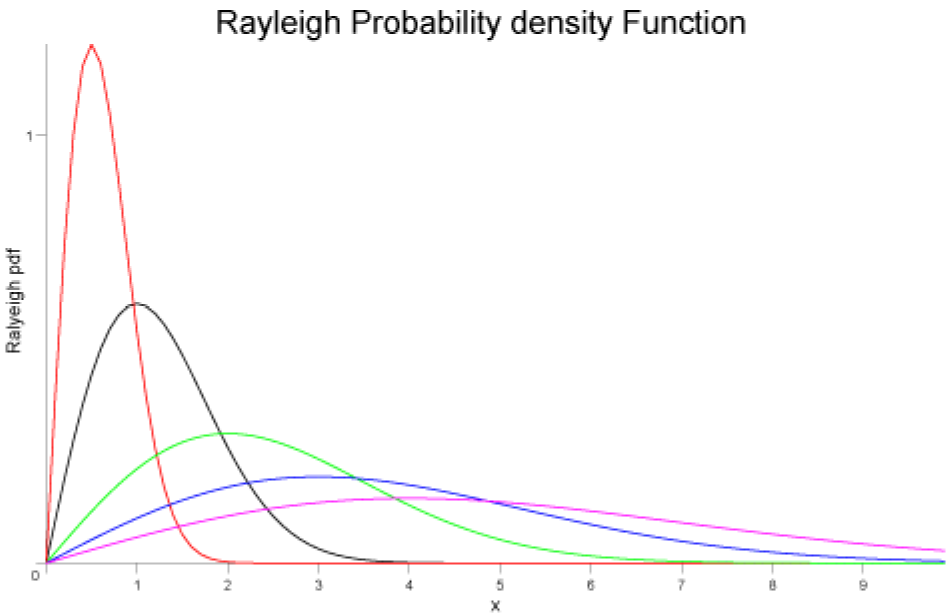
The Rayleigh distribution is a continuous distribution with the probability density function:

f(x; sigma) = x * exp(-x^2/2 σ^2) / σ^2

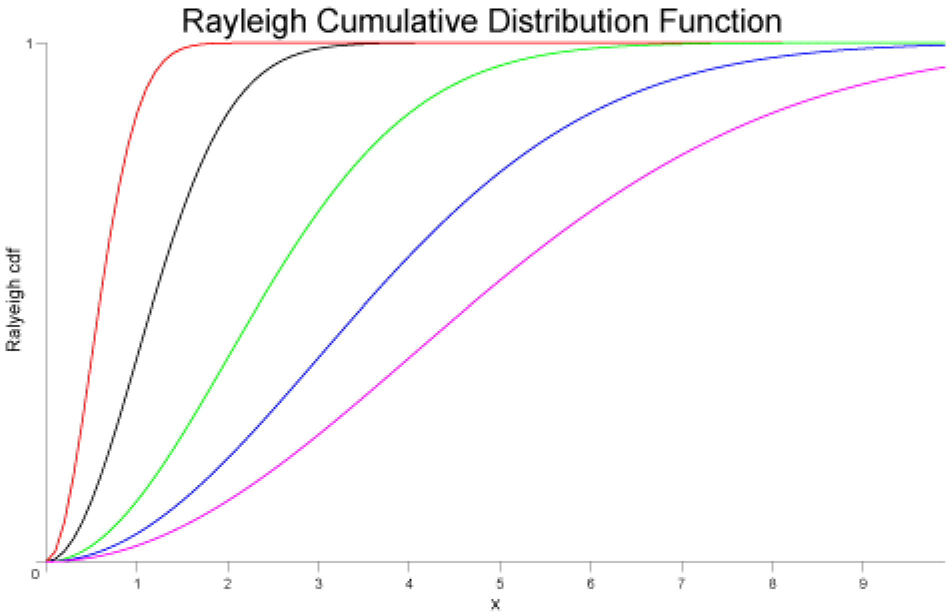
For sigma parameter $\sigma > 0$, and $x > 0$.

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 σ^2
cdf	Using the relation: p = 1 - exp($-x^2$/2) σ^2 = -expm1($-x^2$/2) σ^2
cdf complement	Using the relation: q = exp($-x^2$/ 2) * σ^2
quantile	Using the relation: x = sqrt(-2 * σ^2) * log(1 - p)) = sqrt(-2 * σ^2) * log1p(-p))
quantile from the complement	Using the relation: x = sqrt(-2 * σ^2) * log(q))
mean	σ * sqrt(π/2)
variance	σ^2 * (4 - π/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.

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;

   // Construct:
   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
```

A statistical distribution published by William Gosset in 1908. His employer, Guinness Breweries, required him to publish under a pseudonym, so he chose "Student". Given N independent measurements, let

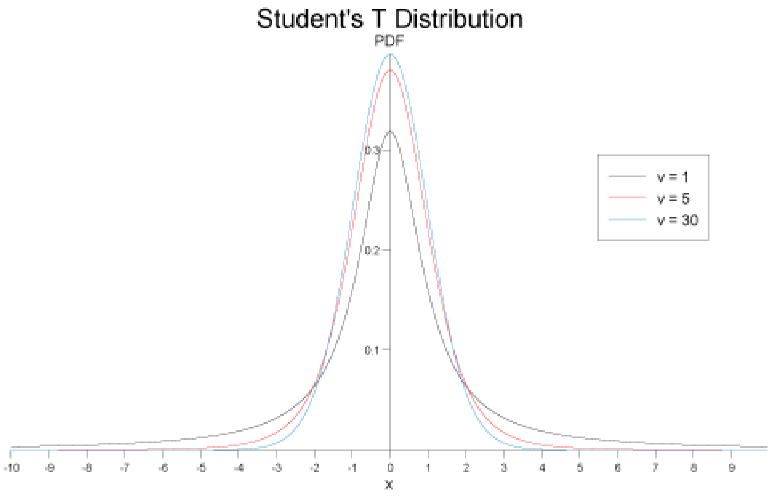
$$t = \frac{\mu - M}{\frac{s}{\sqrt{N}}}$$

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, otherwise calls domain_error. Note that non-integral degrees of freedom are supported, and 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 it's 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))

Function	Implementation Notes
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 < $2t^2$* 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)
mean	0
variance	v / (v - 2)
mode	0
skewness	0
kurtosis	3 * (v - 2) / (v - 4)
kurtosis excess	6 / (df - 4)

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); // Co
       : m_lower(lower), m_mode(mode), m_upper(upper) // Default is -1, 0, +1 triangular dist
    // 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

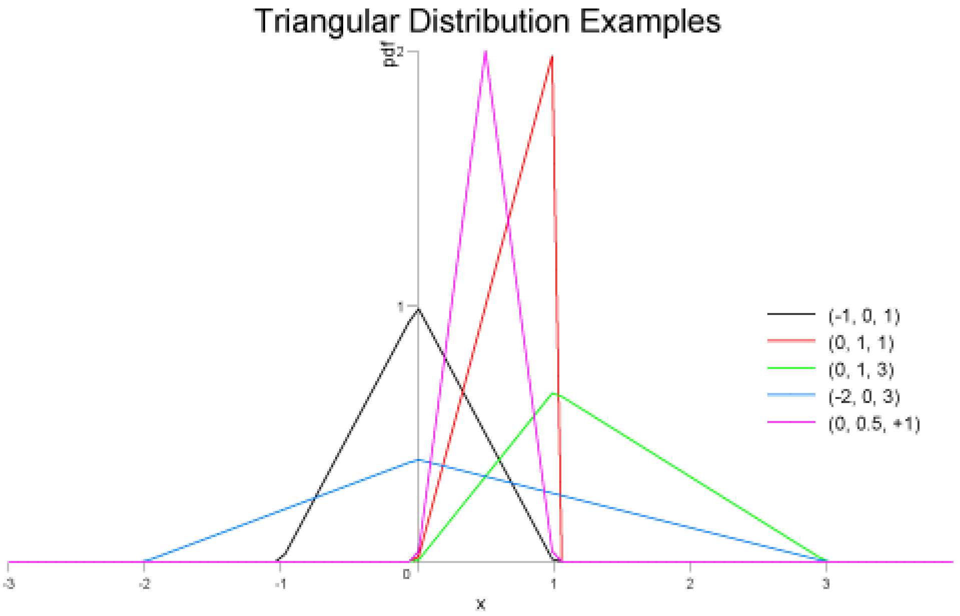
Parameter a (lower) can be any finite value. Parameter b (upper) can be any finite value > a (lower). Parameter c (mode) a <= c <= b. This is the most probable value.

The random variate x must also be finite, and is supported lower <= x <= upper.

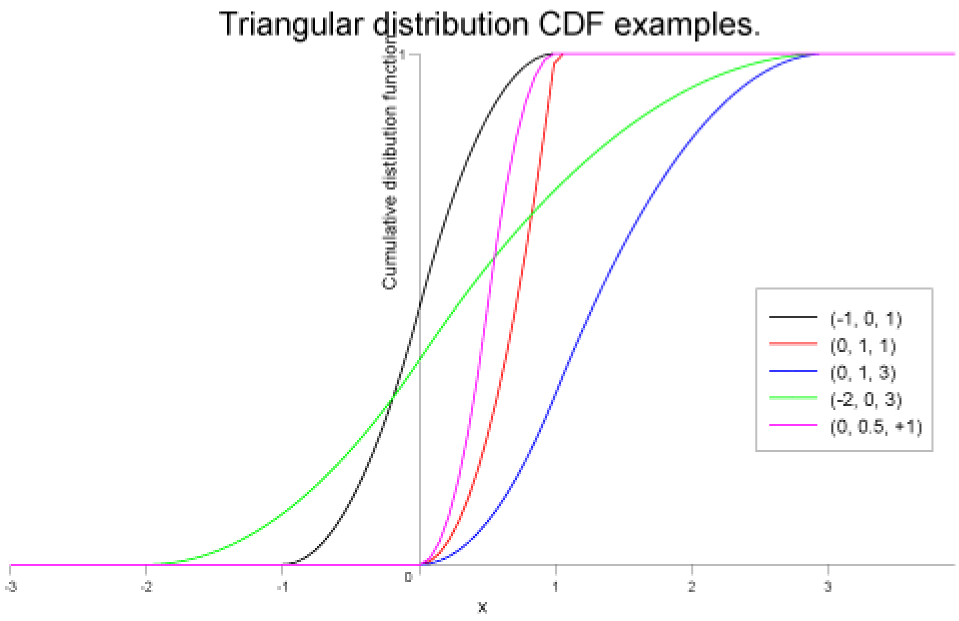
The triangular distribution may be appropriate when an assumption of a normal distribution is unjustified because uncertainty is caused by rounding and quantization from analog to digital conversion. Upper and lower limits are known, and the most probable value lies midway.

The distribution simplifies when the 'best guess' is either the lower or upper limit - a 90 degree angle triangle. The default chosen is 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.

```
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 from Statlet: Calculate and plot probability distributions

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]
- Brighton Webs Ltd. BW D-Calc 1.0 Distribution Calculator
- The Triangular Distribution including its history.
- 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.

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:

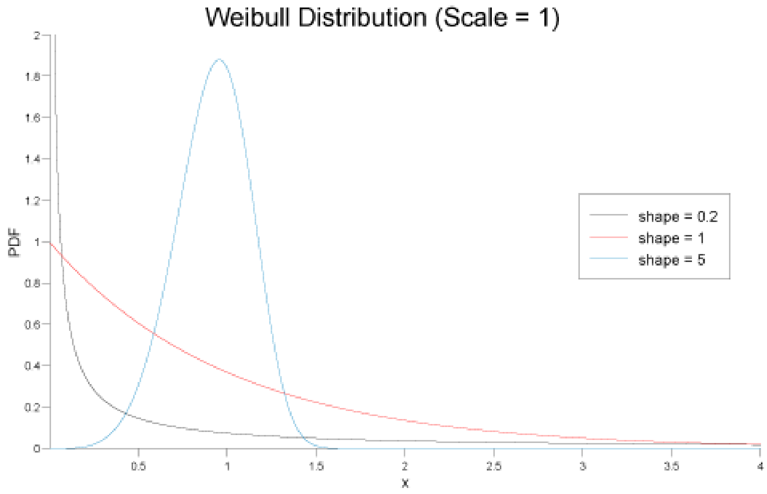
$f(x; \alpha, \beta) = (\alpha/\beta) * (x / \beta)^{\alpha - 1} * e^{-(x/\beta)^{\alpha}}$

For shape parameter $\alpha > 0$, and scale parameter $\beta > 0$, and $x > 0$.

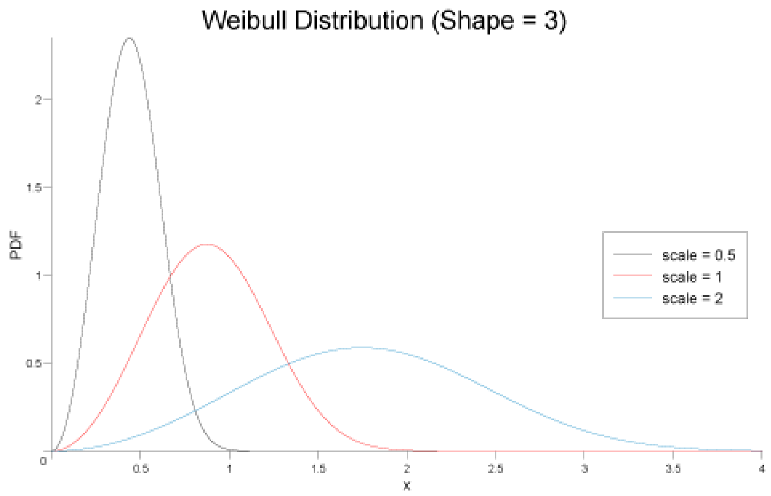
The Weibull distribution is often used in the field of failure analysis; in particular it can mimic distributions where the failure rate varies over time. If the failure rate is:

- 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.

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 it's 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	β * Γ(1 + 1/α)
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

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.

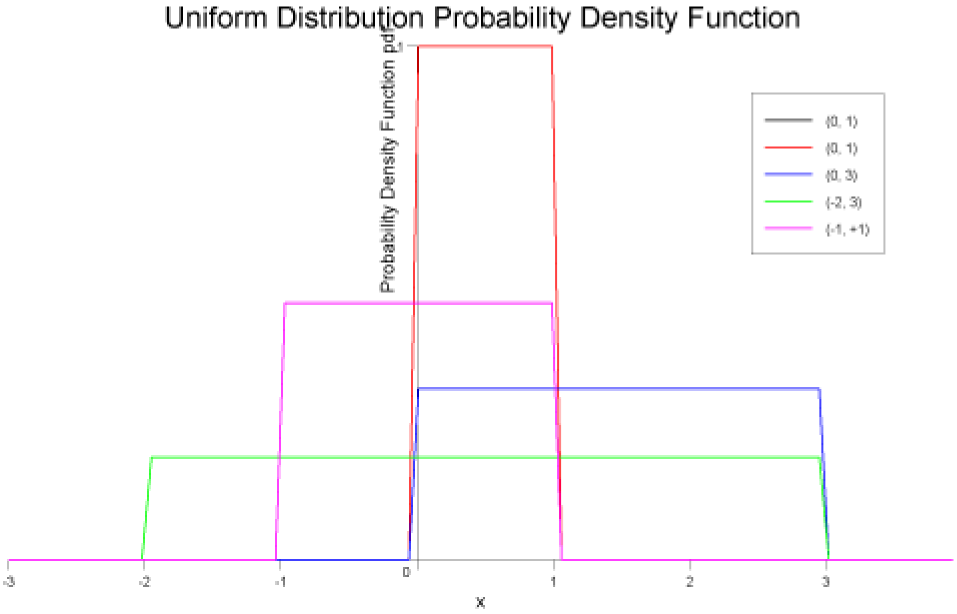
There is also a **discrete** uniform distribution.

Parameters lower and upper can be any finite value.

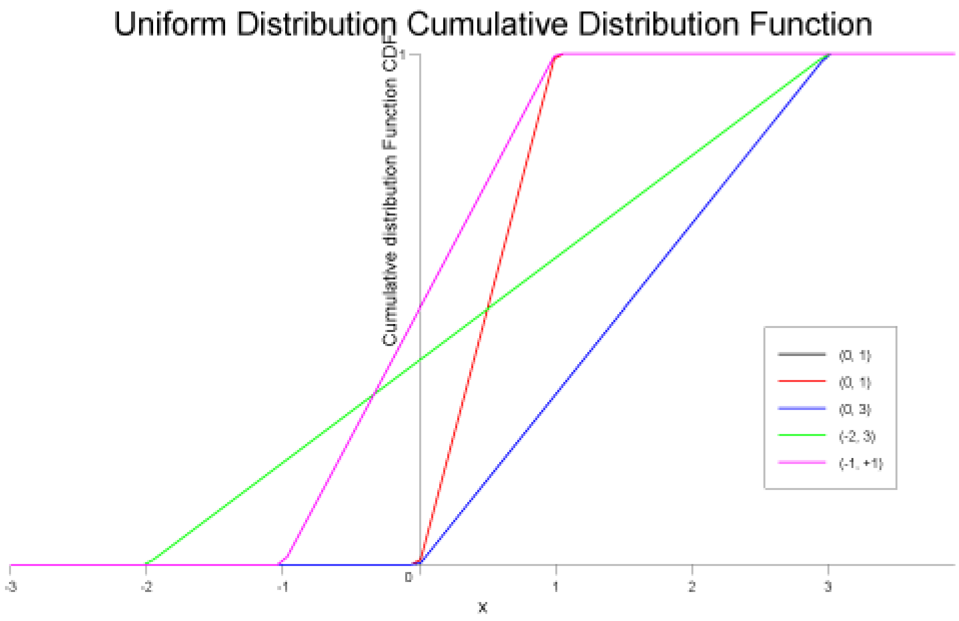
The random variate x must also be finite, and is supported lower <= x <= upper.

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

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)
  // 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)
  // 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 deviatio

  }} // 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)
  // 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
  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
   // 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 distributio
}} // 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 would be fairly trivial 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. 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);
```

Special Functions

Gamma Functions

Gamma

Synopsis

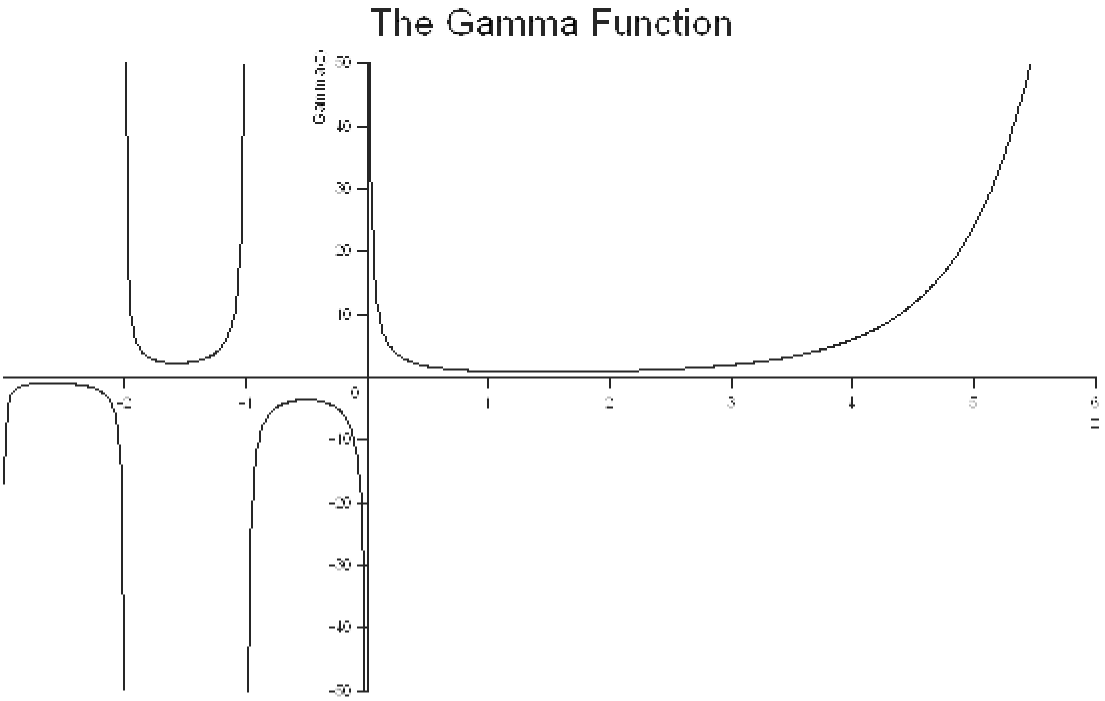
```
#include <boost/math/special_functions/gamma.hpp>


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 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.

Significand Size	Platform and Compiler	Factorials and Half factorials	Values Near Zero	Values Near 1 or 2	Values Near a Negative Pole
53	Win32 Visual C++ 8	Peak=1.9 Mean=0.7 (GSL=3.9) (Cephes=3.0)	Peak=2.0 Mean=1.1 (GSL=4.5) (Cephes=1)	Peak=2.0 Mean=1.1 (GSL=7.9) (Cephes=1.0)	Peak=2.6 Mean=1.3 (GSL=2.5) (Cephes=2.7)
64	Linux IA32 / GCC	Peak=300 Mean=49.5 (GNU C Lib Peak=395 Mean=89)	Peak=3.0 Mean=1.4 (GNU C Lib Peak=11 Mean=3.3)	Peak=5.0 Mean=1.8 (GNU C Lib Peak=0.92 Mean=0.2)	Peak=157 Mean=65 (GNU C Lib Peak=205 Mean=108)
64	Linux IA64 / GCC	GNU C Lib Peak 2.8 Mean=0.9 (GNU C Lib Peak 0.7)	Peak=4.8 Mean=1.5 (GNU C Lib Peak 0)	Peak=4.8 Mean=1.5 (GNU C Lib Peak 0)	Peak=5.0 Mean=1.7 (GNU C Lib Peak 0)
113	HPUX IA64, aCC A.06.06	Peak=2.5 Mean=1.1 (HP-UX C Library Peak 0)	Peak=3.5 Mean=1.7 (HP-UX C Library Peak 0)	Peak=3.5 Mean=1.6 (HP-UX C Library Peak 0)	Peak=5.2 Mean=1.92 (HP-UX C Library Peak 0)

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 by combining the series and continued fraction representations for the incomplete gamma function:

$$\Gamma(z) = \left(l^{a} e^{-l}\right)\left(\sum_{0}^{\infty} \frac{l^{n}}{a^{\overline{n}}} + \cfrac{1}{a - \cfrac{c_1}{d_1 + \cfrac{c_2}{d_2 + \cfrac{c_3}{d_3 + \dots}}}}\right); c_k = \ -(a+k-1) \wedge d_k = a+k+l$$

where *l* is an arbitrary integration limit: choosing `l = max(10, a)` seems to work fairly well.

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([pi][space] * 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
```

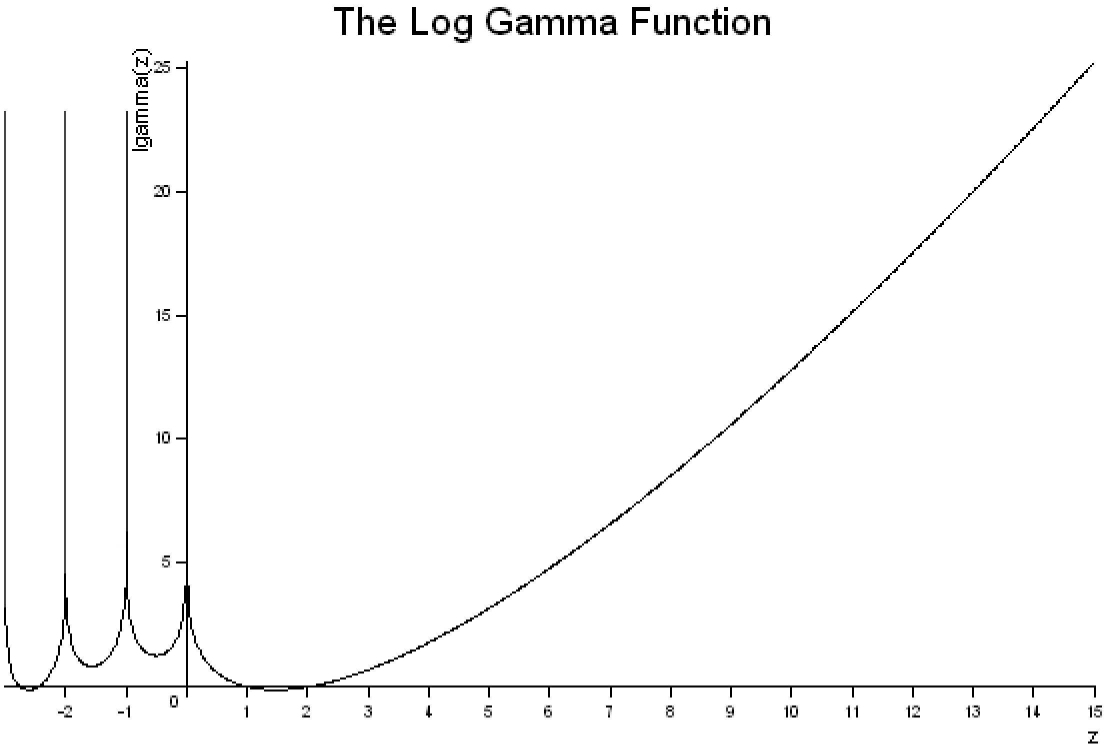
Description

The lgamma function is defined by:

$$\text{lgamma}(z) = \ln\left|\Gamma(z)\right|$$

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 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.

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.

Significand Size	Platform and Compiler	Factorials and Half factorials	Values Near Zero	Values Near 1 or 2	Values Near a Negative Pole
53	Win32 Visual C++ 8	Peak=0.88 Mean=0.14 (GSL=33) (Cephes=1.5)	Peak=0.96 Mean=0.46 (GSL=5.2) (Cephes=1.1)	Peak=0.86 Mean=0.46 (GSL=1168) (Cephes~500000)	Peak=4.2 Mean=1.3 (GSL=25) (Cephes=1.6)
64	Linux IA32 / GCC	Peak=1.9 Mean=0.43 (GNU C Lib Peak=1.7 Mean=0.49)	Peak=1.4 Mean=0.57 (GNU C Lib Peak= 0.96 Mean=0.54)	Peak=0.86 Mean=0.35 (GNU C Lib Peak=0.74 Mean=0.26)	Peak=6.0 Mean=1.8 (GNU C Lib Peak=3.0 Mean=0.86)

Significand Size	Platform and Compiler	Factorials and Half factorials	Values Near Zero	Values Near 1 or 2	Values Near a Negative Pole
64	Linux IA64 / GCC	Peak=0.99 Mean=0.12 (GNU C Lib Peak 0)	Pek=1.2 Mean=0.6 (GNU C Lib Peak 0)	Peak=0.86 Mean=0.16 (GNU C Lib Peak 0)	Peak=2.3 Mean=0.69 (GNU C Lib Peak 0)
113	HPUX IA64, aCC A.06.06	Peak=0.96 Mean=0.13 (HP-UX C Library Peak 0)	Peak=0.99 Mean=0.53 (HP-UX C Library Peak 0)	Peak=0.9 Mean=0.4 (HP-UX C Library Peak 0)	Peak=3.0 Mean=0.9 (HP-UX C Library Peak 0)

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 by combining the series and continued fraction representations for the incomplete gamma function:

$$\ln\left(\left|\Gamma(z)\right|\right) = z\ln(l) - l + \ln\left(\sum_{0}^{\infty}\frac{l^n}{z^{\overline{n}}} + \cfrac{1}{z - \cfrac{c_1}{d_1 + \cfrac{c_2}{d_2 + \cfrac{c_3}{d_3 + \ldots}}}}\right) \quad ; \quad c_k = -(z+k-1) \quad \wedge \quad d_k = z+k+l$$

where *l* is an arbitrary integration limit: choosing `l = max(10, a)` seems to work fairly well. For negative *z* the logarithm version of the reflection formula is used:

$$\ln\left(\left|\Gamma(-z)\right|\right) = \ln(\pi) - \ln(\Gamma(z)) - \ln(z\sin(\pi z))$$

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:

$$\ln\left(\left|\Gamma(z)\right|\right) = (z - 0.5)\ln\left(\frac{z + g - 0.5}{e}\right) + \ln\left(L_{e,g}(z)\right)$$

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*:

$$\ln\left(\left|\Gamma(z)\right|\right) = \Delta z \ln\left(\frac{\Delta z + g + 0.5}{e}\right) + \frac{1}{2}\ln\left(1 + \frac{\Delta z}{g + 0.5}\right) + \ln\left(1 + \sum_{k=0}^{N-1}\frac{\Delta z d_k}{k(\Delta z + k)}\right) \quad ; \quad \Delta z = 1 - z \quad \wedge \quad d_k = \sqrt{}$$

The C_k terms in the above are the same as in the Lanczos approximation.

A similar rearrangement can be performed at *z = 2*:

$$\ln\left(\left|\Gamma(z)\right|\right) = \frac{3}{2}\ln\left(1 + \frac{\Delta z}{g + 1.5}\right) + \Delta z \ln\left(\frac{z + g - 0.5}{e}\right) + \ln\left(1 + \sum_{k=1}^{N-1}\frac{-d_k \Delta z}{z + kz + k^2 - 1}\right) \quad ; \quad \Delta z = z - 2 \quad \wedge \quad d_k$$

Digamma

Synopsis

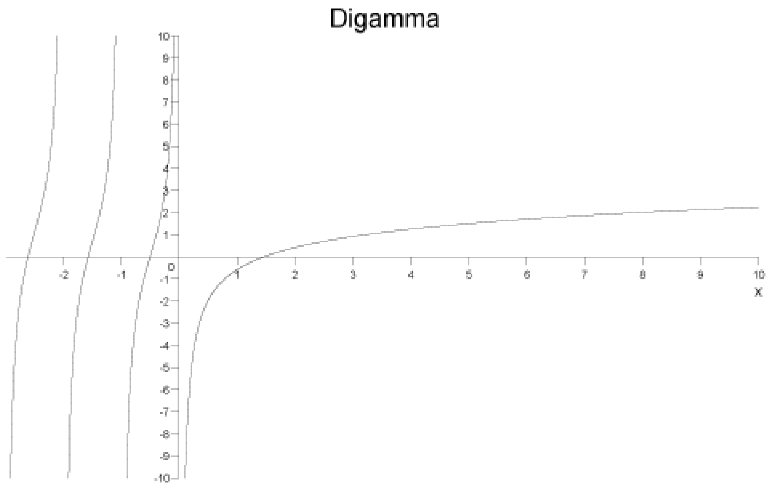
```
#include <boost/math/special_functions/digamma.hpp>
```

```
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:

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



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.

There is no fully generic version of this function: all the implementations are tuned to specific accuracy levels, the most precise of which delivers 34-digits of precision.

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.

Significand Size	Platform and Compiler	Random Positive Values	Values Near The Positive Root	Values Near Zero	Negative Values
53	Win32 Visual C++ 8	Peak=0.98 Mean=0.36	Peak=0.99 Mean=0.5	Peak=0.95 Mean=0.5	Peak=214 Mean=16
64	Linux IA32 / GCC	Peak=1.4 Mean=0.4	Peak=1.3 Mean=0.45	Peak=0.98 Mean=0.35	Peak=180 Mean=13
64	Linux IA64 / GCC	Peak=0.92 Mean=0.4	Peak=1.3 Mean=0.45	Peak=0.98 Mean=0.4	Peak=180 Mean=13
113	HPUX IA64, aCC A.06.06	Peak=0.9 Mean=0.4	Peak=1.1 Mean=0.5	Peak=0.99 Mean=0.4	Peak=64 Mean=6

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.

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:

$$\psi(x) = \ln(x) + \frac{1}{2x} - \sum_{n=1}^{\infty} \frac{B_{2n}}{2n x^{2n}}$$

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 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:

$$\psi(x) = \frac{z-\frac{1}{2}}{x+g-\frac{1}{2}} + \ln\left(x+g-\frac{1}{2}\right) + \frac{P'(x)}{P(x)} - \frac{Q'(x)}{Q(x)} - 1$$

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***.

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:

$$\text{tgamma\_ratio}(a,b) = \frac{\Gamma(a)}{\Gamma(b)}$$

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:

$$\text{tgamma\_delta\_ratio}(a, delta) = \frac{\Gamma(a)}{\Gamma(a + delta)}$$

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 4. Errors In the Function tgamma_delta_ratio(a, delta)

Significand Size	Platform and Compiler	20 < a < 80 and delta < 1
53	Win32, Visual C++ 8	Peak=16.9 Mean=1.7
64	Redhat Linux IA32, gcc-3.4.4	Peak=24 Mean=2.7
64	Redhat Linux IA64, gcc-3.4.4	Peak=12.8 Mean=1.8
113	HPUX IA64, aCC A.06.06	Peak=21.4 Mean=2.3

Table 5. Errors In the Function tgamma_ratio(a, b)

Significand Size	Platform and Compiler	6 < a,b < 50
53	Win32, Visual C++ 8	Peak=34 Mean=9
64	Redhat Linux IA32, gcc-3.4.4	Peak=91 Mean=23
64	Redhat Linux IA64, gcc-3.4.4	Peak=35.6 Mean=9.3
113	HPUX IA64, aCC A.06.06	Peak=43.9 Mean=13.2

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, $\Gamma(a)$]. 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.

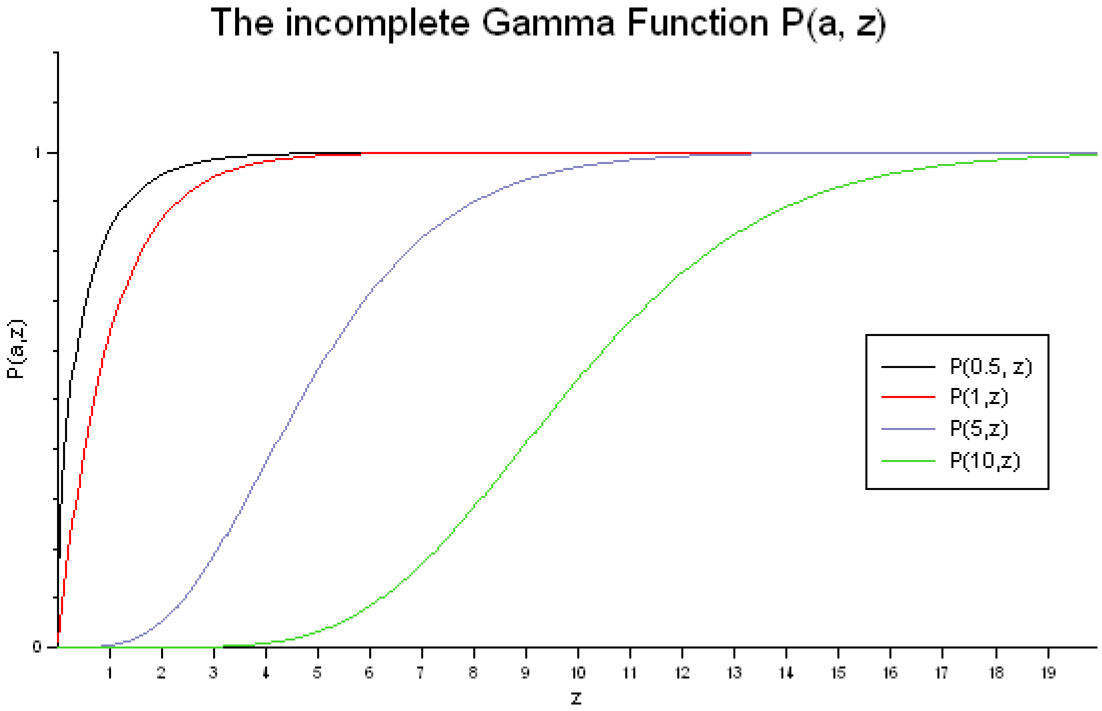
The return type of these functions is computed using the *result type calculation rules* when T1 and T2 are different types, otherwise the return type is simply T1.

```
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:

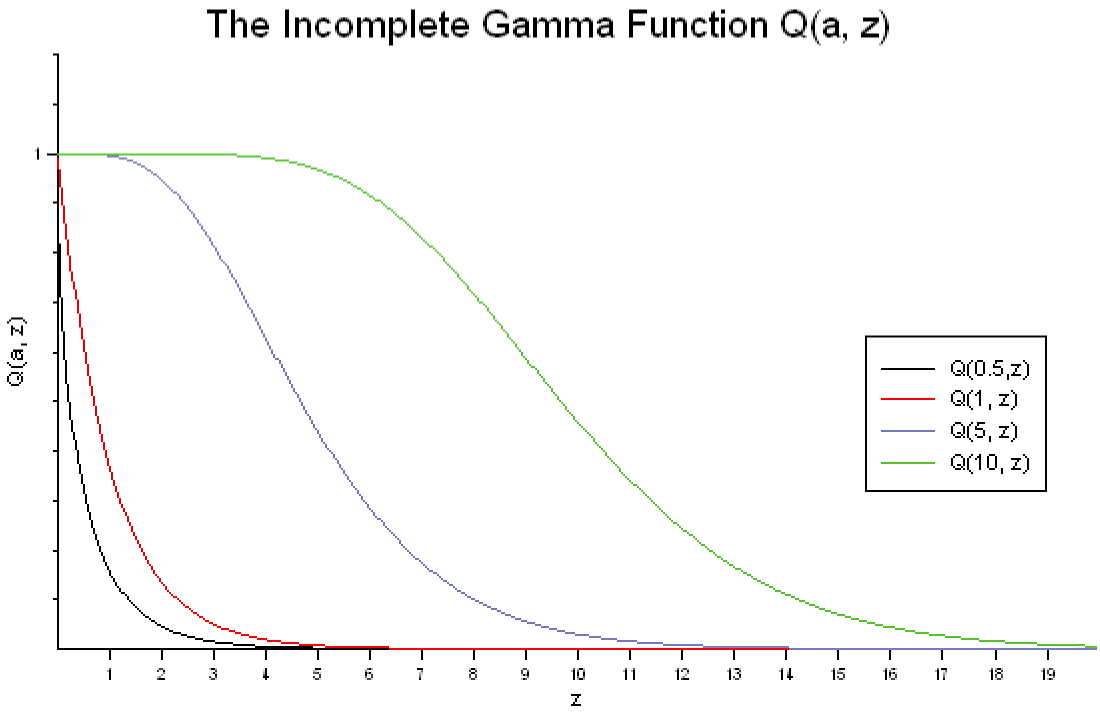


```
template <class T1, class T2>
calculated-result-type gamma_q(T1 a, T2 z);

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:

$$\text{tgamma\_lower}(a, z) = \gamma(a, z) = \int_0^z t^{a-1} e^{-t} dt$$

```
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:

$$\text{tgamma}(a, z) = \Gamma(a, z) = \int_z^\infty t^{a-1} e^{-t} dt$$

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 6. Errors In the Function gamma_p(a,z)

Significand Size	**Platform and Compiler**	**0.5 < a < 100** **and** **0.01*a < z < 100*a**	**$1x10^{-12}$ < a < $5x10^{-2}$** **and** **0.01*a < z < 100*a**	**1e-6 < a < $1.7x10^{6}$** **and** **1 < z < 100*a**
53	Win32, Visual C++ 8	Peak=36 Mean=9.1 (GSL Peak=342 Mean=46) (Cephes Peak=491 Mean=102)	Peak=4.5 Mean=1.4 (GSL Peak=4.8 Mean=0.76) (Cephes Peak=21 Mean=5.6)	Peak=244 Mean=21 (GSL Peak=1022 Mean=1054) (Cephes Peak~$8x10^{6}$ Mean~$7x10^{4}$)
64	RedHat Linux IA32, gcc-3.3	Peak=241 Mean=36	Peak=4.7 Mean=1.5	Peak~30,220 Mean=1929
64	Redhat Linux IA64, gcc-3.4	Peak=41 Mean=10	Peak=4.7 Mean=1.4	Peak~30,790 Mean=1864
113	HPUX IA64, aCC A.06.06	Peak=40.2 Mean=10.2	Peak=5 Mean=1.6	Peak=5,476 Mean=440

Table 7. Errors In the Function gamma_q(a,z)

Significand Size	**Platform and Compiler**	**0.5 < a < 100** **and** **0.01*a < z < 100*a**	**$1x10^{-12}$ < a < $5x10^{-2}$** **and** **0.01*a < z < 100*a**	**$1x10^{-6}$ < a < $1.7x10^{6}$** **and** **1 < z < 100*a**
53	Win32, Visual C++ 8	Peak=28.3 Mean=7.2 (GSL Peak=201 Mean=13) (Cephes Peak=556 Mean=97)	Peak=4.8 Mean=1.6 (GSL Peak~$1.3x10^{10}$ Mean=$1x10^{+9}$) (Cephes Peak~$3x10^{11}$ Mean=$4x10^{10}$)	Peak=469 Mean=33 (GSL Peak=27,050 Mean=2159) (Cephes Peak~$8x10^{6}$ Mean~$7x10^{5}$)
64	RedHat Linux IA32, gcc-3.3	Peak=280 Mean=33	Peak=4.1 Mean=1.6	Peak=11,490 Mean=732
64	Redhat Linux IA64, gcc-3.4	Peak=32 Mean=9.4	Peak=4.7 Mean=1.5	Peak=6815 Mean=414
113	HPUX IA64, aCC A.06.06	Peak=37 Mean=10	Peak=11.2 Mean=2.0	Peak=4,999 Mean=298

Table 8. Errors In the Function tgamma_lower(a,z)

Significand Size	Platform and Compiler	0.5 < a < 100 and 0.01*a < z < 100*a	$1x10^{-12}$ < a < $5x10^{-2}$ and 0.01*a < z < 100*a
53	Win32, Visual C++ 8	Peak=5.5 Mean=1.4	Peak=3.6 Mean=0.78
64	RedHat Linux IA32, gcc-3.3	Peak=402 Mean=79	Peak=3.4 Mean=0.8
64	Redhat Linux IA64, gcc-3.4	Peak=6.8 Mean=1.4	Peak=3.4 Mean=0.78
113	HPUX IA64, aCC A.06.06	Peak=6.1 Mean=1.8	Peak=3.7 Mean=0.89

Table 9. Errors In the Function tgamma(a,z)

Significand Size	Platform and Compiler	0.5 < a < 100 and 0.01*a < z < 100*a	$1x10^{-12}$ < a < $5x10^{-2}$ and 0.01*a < z < 100*a
53	Win32, Visual C++ 8	Peak=5.9 Mean=1.5	Peak=1.8 Mean=0.6
64	RedHat Linux IA32, gcc-3.3	Peak=596 Mean=116	Peak=3.2 Mean=0.84
64	Redhat Linux IA64, gcc-3.4.4	Peak=40.2 Mean=2.5	Peak=3.2 Mean=0.8
113	HPUX IA64, aCC A.06.06	Peak=364 Mean=17.6	Peak=12.7 Mean=1.8

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) $$\gamma(a,x) = x^a e^{-x} \sum_{k=0}^{\infty} \frac{\Gamma(a)}{\Gamma(a+k+1)} x^n = x^a e^{-x} \sum_{k=0}^{\infty} \frac{x^n}{a^{\overline{k+1}}}$$

Or by subtraction of the upper integral from either Γ(a) or 1 when *x > a and x > 1.1*.

The upper integral is computed from Legendre's continued fraction representation:

5) $$\Gamma(a, z) = \frac{x^a e^{-x}}{x - a + 1 + \frac{a_k}{b_k + \frac{a_{k+1}}{b_{k+1} + \cdots}}} \quad ; \quad a_k = k(a-k) \quad ; \quad b_k = x - a + 2k + 1$$

When *x > 1.1* or by subtraction of the lower integral from either Γ(a) or 1 when *x < 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) $$\Gamma(a, x) = \frac{\mathrm{tgamma1pm1}(a) - \mathrm{powm1}(x, a)}{a} + x^a \sum_{k=1}^{\infty} \frac{(-1)^k x^k}{(a+k)k!}$$
$$\mathrm{tgamma1pm1}(a) = \Gamma(a+1) - 1$$
$$\mathrm{powm1}(x, a) = x^a - 1$$

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 * 1.1 < 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 0.6. 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) $$Q(a, x) = \mathrm{erfc}\left(\sqrt{x}\right) + \frac{e^{-x}}{\sqrt{\pi x}} \sum_{n=1}^{i} \frac{x^n}{\left(1 - \frac{1}{2}\right) \ldots \left(n - \frac{1}{2}\right)} \quad ; \quad a = i + \frac{1}{2} \quad ; \quad i \in \mathbb{N}^+$$

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) $$P(a, x) = \frac{1}{2} \mathrm{erfc}\left(\sqrt{y}\right) - \frac{e^{-y}}{\sqrt{2 \pi a}} T(a, \lambda) \quad ; \quad \lambda \leq 1$$

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)
$$e^{x-a}\left(\frac{x}{a+g-0.5}\right)^a \quad = \quad e^{\left(a\,\mathrm{log1pmx}\left(\frac{x-a-g+0.5}{a+g-0.5}\right) + \frac{x(0.5-g)}{a+g-0.5}\right)} \quad ; \quad \mathrm{log1pmx}(z) = \ln(1+z) - z$$

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.

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.

$$\text{gamma\_p\_derivative}(a, x) = \frac{\partial}{\partial x} P(a, x) = \frac{e^{-x} x^{a-1}}{\Gamma(a)}$$

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>
T unchecked_factorial(unsigned i);

template <class T>
struct max_factorial;

}} // namespaces
```

Description

```
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>
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.

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.

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

$(2n-1)!! = \Gamma((2n+1)/2) * 2^{n} / \mathrm{sqrt}(pi)$

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)

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.

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 * beta(k, n-k+1))

and

${}_nC_k$ = 1 / ((n-k) * beta(k+1, n-k))

Beta Functions

Beta

Synopsis

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

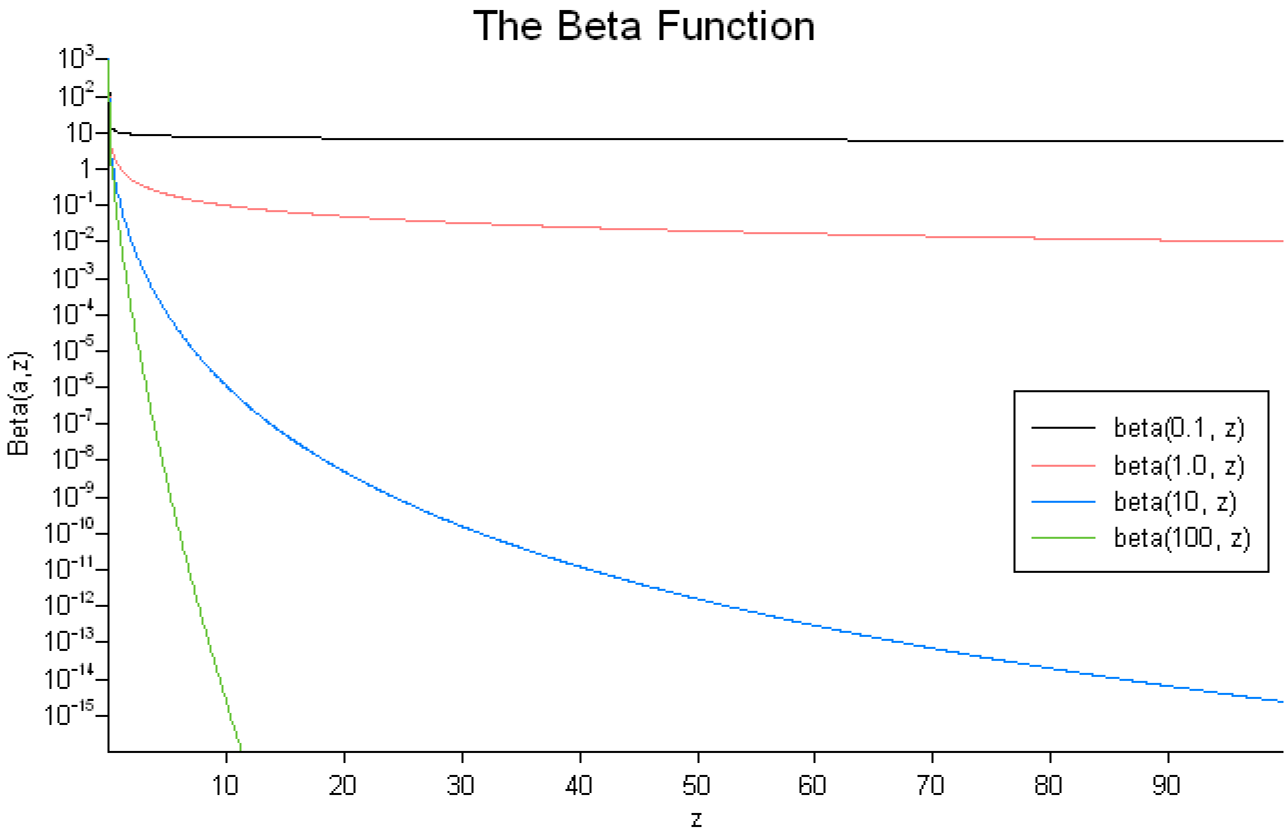
```
namespace boost{ namespace math{
```

```
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:

$$\mathrm{beta}(a, b) = B(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$



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 10. Peak Errors In the Beta Function

Significand Size	**Platform and Compiler**	**Errors in range** **0.4 < a,b < 100**	**Errors in range** **1e-6 < a,b < 36**
53	Win32, Visual C++ 8	Peak=99 Mean=22 (GSL Peak=1178 Mean=238) (Cephes=1612)	Peak=10.7 Mean=2.6 (GSL Peak=12 Mean=2.0) (Cephes=174)
64	Red Hat Linux IA32, g++ 3.4.4	Peak=112.1 Mean=26.9	Peak=15.8 Mean=3.6
64	Red Hat Linux IA64, g++ 3.4.4	Peak=61.4 Mean=19.5	Peak=12.2 Mean=3.6
113	HPUX IA64, aCC A.06.06	Peak=42.03 Mean=13.94	Peak=9.8 Mean=3.1

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:

$$\mathrm{beta}(a, b) = \left(\frac{a+g-0.5}{a+b+g-0.5}\right)^{a-b-0.5}\left(\frac{(a+g-0.5)(b+g-0.5)}{(a+b+g-0.5)^2}\right)^{b}\sqrt{\frac{e}{b+g-0.5}}\,\frac{L(a)L(b)}{L(c)}$$

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:

$$\text{beta}(a, b) = \frac{(a+b)}{b}\text{beta}(a, b+1)$$

$$\text{beta}(a, b) = \frac{(a+b)(a+b+1)}{a\,b}\text{beta}(a+1, b+1)$$

to move to a more favorable region where they are both greater than 1.

In addition:

$$\text{if} \quad a = 1 \qquad \text{then:} \qquad \text{beta}(a, b) = \frac{1}{b}$$

Incomplete Beta Functions

Synopsis

```
#include <boost/math/special_functions/gamma.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 *a > 0*, *b > 0* and *0 <= x <= 1*.

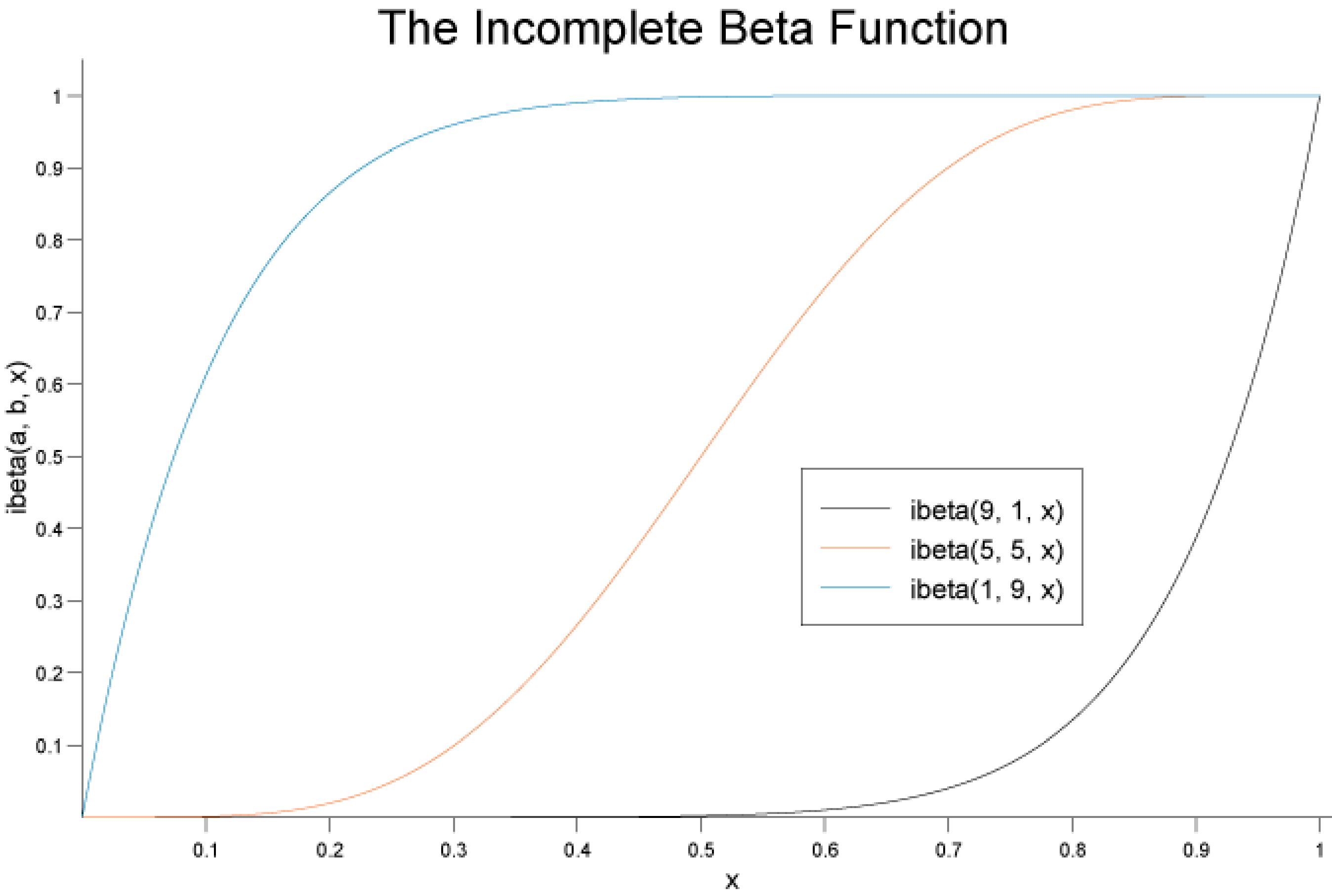
The return type of these functions is computed using the *result type calculation rules* when T1, T2 and T3 are different types.

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:

$$\text{ibeta}(a, b, x) \quad = \quad I_x(a, b) \quad = \quad \frac{1}{\text{beta}(a, b)} \int_0^x t^{a-1}(1-t)^{b-1}\,dt$$



```
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 11. Errors In the Function ibeta(a,b,x)

Significand Size	**Platform and Compiler**	**0 < a,b < 10** **and** **0 < x < 1**	**0 < a,b < 100** **and** **0 < x < 1**	**$1x10^{-5}$ < a,b < $1x10^{5}$** **and** **0 < x < 1**
53	Win32, Visual C++ 8	Peak=42.3 Mean=2.9 (GSL Peak=682 Mean=32.5) (Cephes Peak=42.7 Mean=7.0)	Peak=108 Mean=16.6 (GSL Peak=690 Mean=151) (Cephes Peak=1545 Mean=218)	Peak=$4x10^3$ Mean=203 (GSL Peak~$3x10^5$ Mean~$2x10^4$) (Cephes Peak~$5x10^5$ Mean~$2x10^4$)
64	Redhat Linux IA32, gcc-3.4.4	Peak=21.9 Mean=3.1	Peak=270.7 Mean=26.8	Peak~$5x10^4$ Mean=$3x10^3$
64	Redhat Linux IA64, gcc-3.4.4	Peak=15.4 Mean=3.0	Peak=112.9 Mean=14.3	Peak~$5x10^4$ Mean=$3x10^3$
113	HPUX IA64, aCC A.06.06	Peak=20.9 Mean=2.6	Peak=88.1 Mean=14.3	Peak~$2x10^4$ Mean=$1x10^3$

Table 12. Errors In the Function ibetac(a,b,x)

Significand Size	**Platform and Compiler**	**0 < a,b < 10** **and** **0 < x < 1**	**0 < a,b < 100** **and** **0 < x < 1**	**$1x10^{-5}$ < a,b < $1x10^{5}$** **and** **0 < x < 1**
53	Win32, Visual C++ 8	Peak=13.9 Mean=2.0	Peak=56.2 Mean=14	Peak=$3x10^3$ Mean=159
64	Redhat Linux IA32, gcc-3.4.4	Peak=21.1 Mean=3.6	Peak=221.7 Mean=25.8	Peak~$9x10^4$ Mean=$3x10^3$
64	Redhat Linux IA64, gcc-3.4.4	Peak=10.6 Mean=2.2	Peak=73.9 Mean=11.9	Peak~$9x10^4$ Mean=$3x10^3$
113	HPUX IA64, aCC A.06.06	Peak=9.9 Mean=2.6	Peak=117.7 Mean=15.1	Peak~$3x10^4$ Mean=$1x10^3$

Table 13. Errors In the Function beta(a, b, x)

Significand Size	**Platform and Compiler**	**0 < a,b < 10** **and** **0 < x < 1**	**0 < a,b < 100** **and** **0 < x < 1**	**$1x10^{-5}$ < a,b < $1x10^{5}$** **and** **0 < x < 1**
53	Win32, Visual C++ 8	Peak=39 Mean=2.9	Peak=91 Mean=12.7	Peak=635 Mean=25
64	Redhat Linux IA32, gcc-3.4.4	Peak=26 Mean=3.6	Peak=180.7 Mean=30.1	Peak~$7x10^4$ Mean=$3x10^3$
64	Redhat Linux IA64, gcc-3.4.4	Peak=13 Mean=2.4	Peak=67.1 Mean=13.4	Peak~$7x10^4$ Mean=$3x10^3$
113	HPUX IA64, aCC A.06.06	Peak=27.3 Mean=3.6	Peak=49.8 Mean=9.1	Peak~$6x10^4$ Mean=$3x10^3$

Table 14. Errors In the Function betac(a,b,x)

Significand Size	**Platform and Compiler**	**0 < a,b < 10** **and** **0 < x < 1**	**0 < a,b < 100** **and** **0 < x < 1**	**$1x10^{-5}$ < a,b < $1x10^{5}$** **and** **0 < x < 1**
53	Win32, Visual C++ 8	Peak=12.0 Mean=2.4	Peak=91 Mean=15	Peak=$4x10^{3}$ Mean=113
64	Redhat Linux IA32, gcc-3.4.4	Peak=19.8 Mean=3.8	Peak=295.1 Mean=33.9	Peak~$1x10^{5}$ Mean=$5x10^{3}$
64	Redhat Linux IA64, gcc-3.4.4	Peak=11.2 Mean=2.4	Peak=63.5 Mean=13.6	Peak~$1x10^{5}$ Mean=$5x10^{3}$
113	HPUX IA64, aCC A.06.06	Peak=15.6 Mean=3.5	Peak=39.8 Mean=8.9	Peak~$9x10^{4}$ Mean=$5x10^{3}$

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:

$$I_x(a, b) \quad = \quad \frac{x^a y^b}{a B(a, b)} \left(\frac{1}{1 + \frac{d_1}{1 + \frac{d_2}{1 + \dots}}} \right)$$

$$d_{2m+1} = \frac{-(a+m)(a+b+m)x}{(a+2m)(a+2m+1)}$$

$$d_{2m} = \frac{m(b-m)x}{(a+2m-1)(a+2m)}$$

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:

$$I_x(a, b) = \frac{x^a y^b}{B(a, b)} \left(\cfrac{\alpha_1}{\beta_1 + \cfrac{\alpha_2}{\beta_2 + \cfrac{\alpha_3}{\beta_3 + \dots}}} \right)$$

$$\alpha_1 = 1 \quad , \quad \alpha_{m+1} = \frac{(a + m - 1)(a + b + m - 1)\, m\, (b - m)\, x^2}{(a + 2m - 1)^2}$$

$$\beta_{m+1} = m + \frac{m\,(b - m)\, x}{a + 2m - 1} + \frac{(a + m)(a - (a + b)x + 1 + m\,(2 - x))}{a + 2m + 1}$$

For smallish b and x then a series representation can be used:

$$I_x(a, b) = \frac{x^a}{B(a, b)} \sum_{n=1}^{\infty} \frac{(1 - b)^{\bar{n}}\, x^n}{(a + n)\, n!}$$

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:

$$I_x(a, b) \approx M \sum_{n=0}^{\infty} p_n J_n(b, u) \quad ; \quad a > b$$

$$M = \frac{H(b, u)\, \Gamma(a + b)}{\Gamma(a)\, T^b}$$

$$H(c, u) = \frac{e^{-u} u^c}{\Gamma(a)}$$

$$T = a + \frac{b - 1}{2} \qquad u = -T \ln(x)$$

$$p_0 = 1 \qquad p_n = \frac{(b - 1)}{(2n - 1)!} + \frac{1}{n} \sum_{m=1}^{n-1} \frac{(m\,b - n)}{(2m + 1)!}\, p_{n-m}$$

$$J_n(b, u) = \left(\frac{u}{2T}\right)^{2n} \frac{Q(b + 2n, u)}{H(b + 2n, u)} = \left(\frac{-\ln x}{2}\right)^{2n} \frac{Q(b + 2n, u)}{H(b + 2n, u)}$$

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:

$$\begin{aligned} I_x(a,b) &= I_x(a+n,b) + x^a(1-x)^b \sum_{j=1}^{n} \frac{\Gamma(a+b+j-1)}{\Gamma(b)\Gamma(a+j)} x^{j-1} \\ &= I_x(a+n,b) + \frac{x^a(1-x)^b}{a} \sum_{j=0}^{n-1} \frac{(a+b)^{\bar{j}}}{(a+1)^{\bar{j}}} x^j \end{aligned}$$

$$B_x(a,b) = \frac{(a+b)^{\bar{n}}}{a^{\bar{n}}} B_x(a+n,b) + \frac{x^a(1-x)^b}{a} \sum_{j=0}^{n-1} \frac{(a+b)^{\bar{j}}}{(a+1)^{\bar{j}}} x^j$$

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 ibeta_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`.

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 12 *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

$$I_x(a, a+\beta) \quad ; \quad \beta < \sqrt{a}$$

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

$$I_x(a, b) \quad ; \quad 0.2 \le \frac{a}{a+b} \le 0.8$$

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:

$$I_x^{-1}(a, b) \quad \approx \quad (a\,p\,B(a, b))^{\frac{1}{a}}$$

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:

$$y \quad = \quad 1 - I_x^{-1}(a, b) \quad \approx \quad (b\,q\,B(a, b))^{\frac{1}{b}}$$

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:

$$x_0 \quad = \quad \frac{x_g}{1+x_g} \quad ; \quad x_g \quad = \quad (a\,p\,B(a, b))^{\frac{1}{a}}$$

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:

$$I_x^{-1}(a, b) \quad \approx \quad (a\,p\,B(a, b))^{\frac{1}{a}}$$

which can be improved somewhat to:

$$I_x^{-1}(a,\ b) \quad \approx \quad (a\,p\,B(a,\ b))^{\frac{1}{a}} + \frac{b-1}{a+1}(a\,p\,B(a,\ b))^{\frac{2}{a}} + \frac{(b-1)\left(a^2+3\,b\,a-a+5b-4\right)}{2(a+1)^2(a+2)}(a\,p\,B(a,\ b))^{\frac{3}{a}}$$

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

$$I_x^{-1}(a,\ b) \quad \approx \quad (a\,p\,B(a,\ b))^{\frac{1}{a}}$$

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:

$$y \quad = \quad \left(1 - p^{b\,B(a,\ b)}\right)^{\frac{1}{b}}$$

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.

$$\text{ibeta\_derivative}(a, b, x) \quad = \quad \frac{\partial}{\partial x} I_x(a, b) \quad = \quad \frac{(1-x)^{b-1} x^{a-1}}{B(a, b)}$$

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.

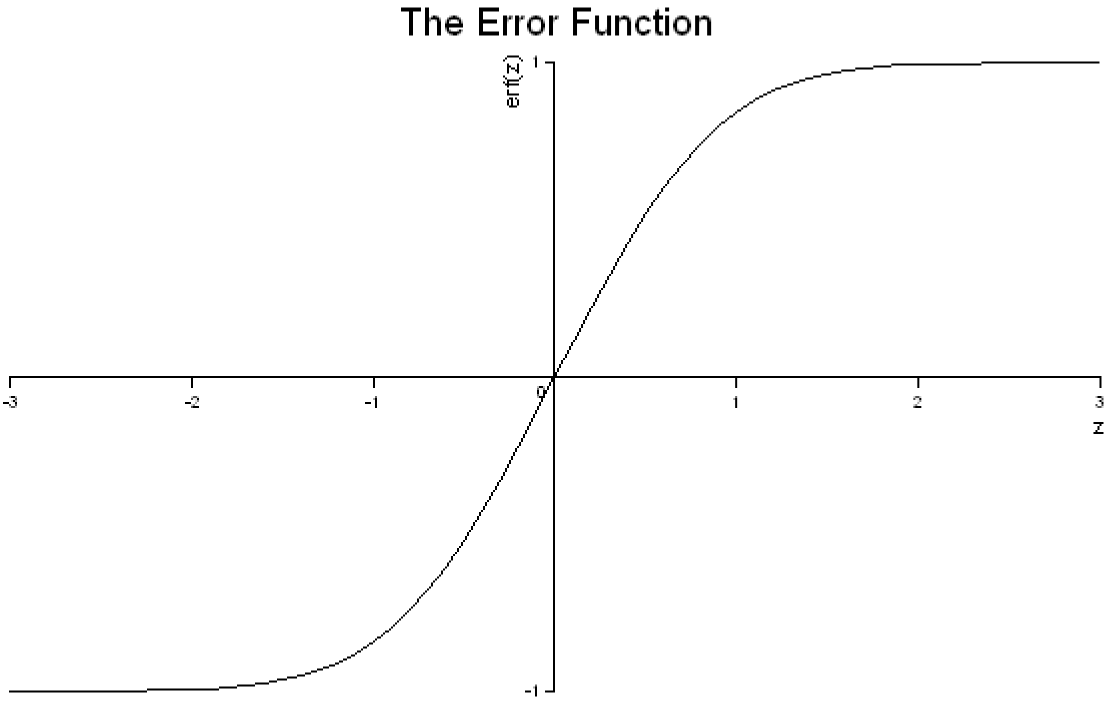
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

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:

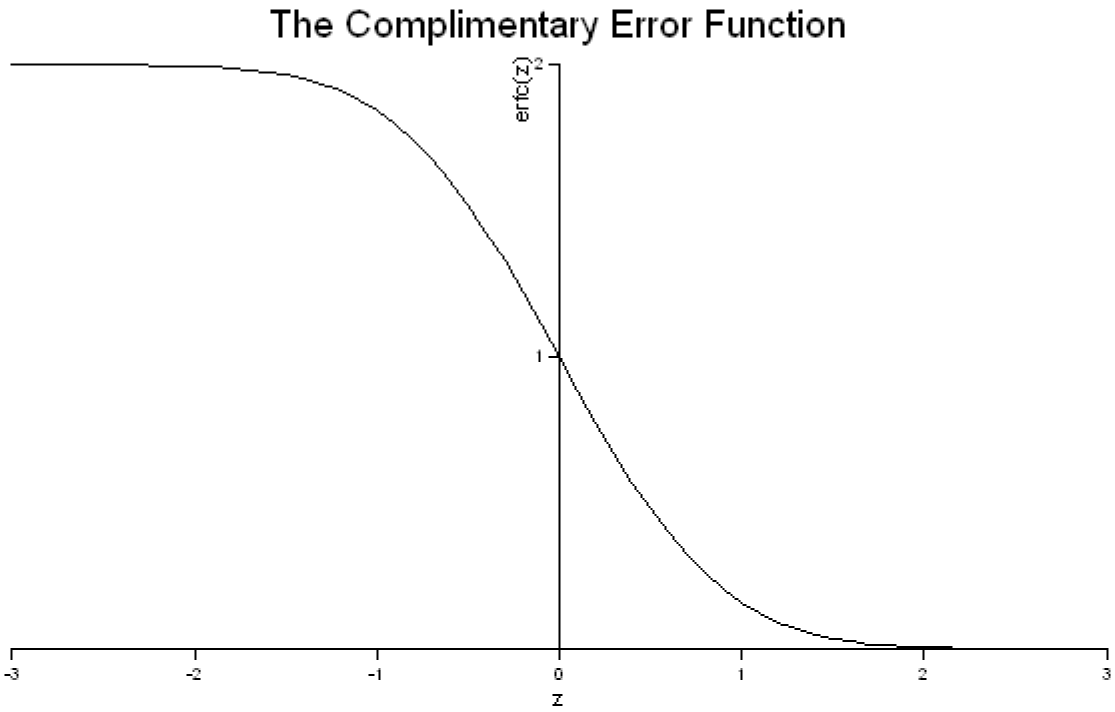
$$\operatorname{erf}(z) \quad = \quad \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$



```
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:

$$\operatorname{erfc}(z) \quad = \quad 1 - \operatorname{erf}(z)$$



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 15. Errors In the Function erf(z)

Significand Size	Platform and Compiler	z < 0.5	0.5 < z < 8	z > 8
53	Win32, Visual C++ 8	Peak=0 Mean=0 GSL Peak=2.0 Mean=0.3 Cephes Peak=1.1 Mean=0.7	Peak=0.9 Mean=0.09 GSL Peak=2.3 Mean=0.3 Cephes Peak=1.3 Mean=0.2	Peak=0 Mean=0 GSL Peak=0 Mean=0 Cephes Peak=0 Mean=0
64	RedHat Linux IA32, gcc-3.3	Peak=0.7 Mean=0.07 GNU C Lib Peak=0.9 Mean=0.2	Peak=0.9 Mean=0.2 GNU C Lib Peak=0.9 Mean=0.07	Peak=0 Mean=0 GNU C Lib Peak=0 Mean=0
64	Redhat Linux IA64, gcc-3.4.4	Peak=0.7 Mean=0.07 GNU C Lib Peak=0 Mean=0	Peak=0.9 Mean=0.1 GNU C Lib Peak=0.5 Mean=0.03	Peak=0 Mean=0 GNU C Lib Peak=0 Mean=0
113	HPUX IA64, aCC A.06.06	Peak=0.8 Mean=0.1 HP-UX C Library Lib Peak=0.9 Mean=0.2	Peak=0.9 Mean=0.1 HP-UX C Library Lib Peak=0.5 Mean=0.02	Peak=0 Mean=0 HP-UX C Library Lib Peak=0 Mean=0

Table 16. Errors In the Function erfc(z)

Significand Size	Platform and Compiler	z < 0.5	0.5 < z < 8	z > 8
53	Win32, Visual C++ 8	Peak=0.7 Mean=0.06 GSL Peak=1.0 Mean=0.4 Cephes Peak=0.7 Mean=0.06	Peak=0.99 Mean=0.3 GSL Peak=2.6 Mean=0.6 Cephes Peak=3.6 Mean=0.7	Peak=1.0 Mean=0.2 GSL Peak=3.9 Mean=0.4 Cephes Peak=2.7 Mean=0.4
64	RedHat Linux IA32, gcc-3.3	Peak=0 Mean=0 GNU C Lib Peak=0 Mean=0	Peak=1.4 Mean=0.3 GNU C Lib Peak=1.3 Mean=0.3	Peak=1.6 Mean=0.4 GNU C Lib Peak=1.3 Mean=0.4
64	Redhat Linux IA64, gcc-3.4.4	Peak=0 Mean=0 GNU C Lib Peak=0 Mean=0	Peak=1.4 Mean=0.3 GNU C Lib Peak=0 Mean=0	Peak=1.5 Mean=0.4 GNU C Lib Peak=0 Mean=0
113	HPUX IA64, aCC A.06.06	Peak=0 Mean=0 HP-UX C Library Peak=0 Mean=0	Peak=1.5 Mean=0.3 HP-UX C Library Peak=0.9 Mean=0.08	Peak=1.6 Mean=0.4 HP-UX C Library Peak=0.9 Mean=0.1

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(z)/z ~ 1.12....
```

Therefore erf is calculated using:

```
erf(z) = z * (1.125F + R(z));
```

where the rational approximation R(z) is optimised for absolute error: as long as its absolute error is small enough compared to 1.125, then any round-off error incurred during the computation of R(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)) / z;
```

Again R(z) is optimised for absolute error, and the constant `c` is the average of `erfc(z) * exp(z*z) * z` taken at the endpoints of the range. Once again, as long as the absolute error in R(z) is small compared to `c` then `c + R(z)` 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.

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.

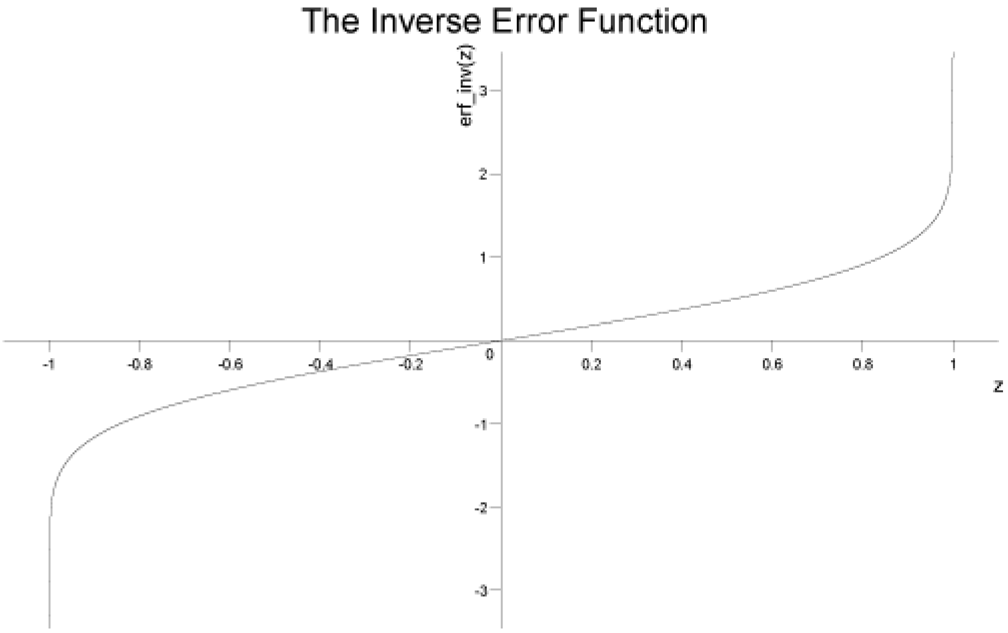
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

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:

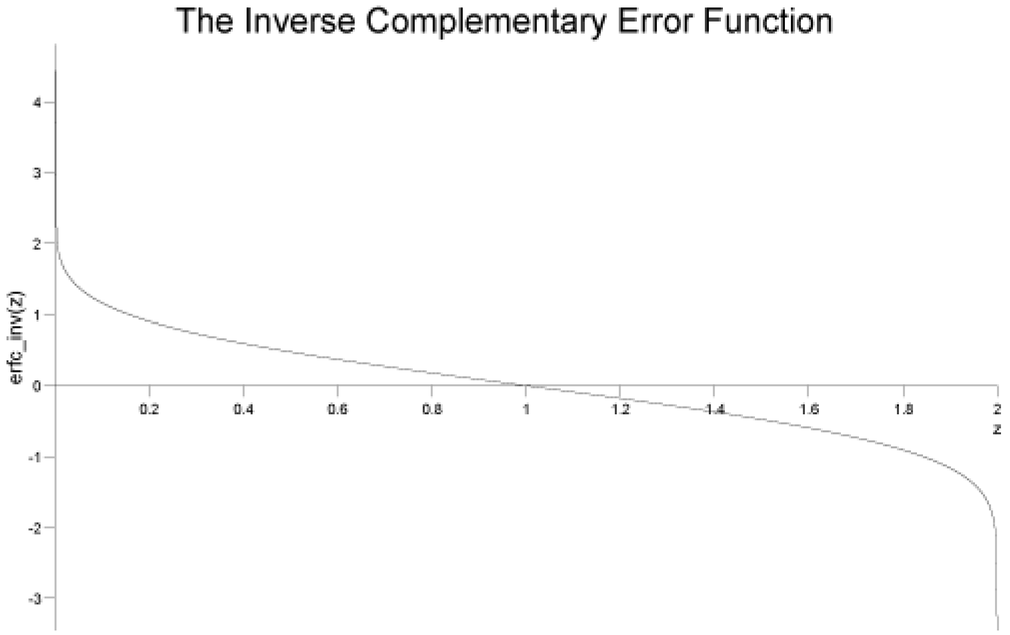
```
p = erf(x);
```



```
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.

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 $\sim 10^{-19}$, 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(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:

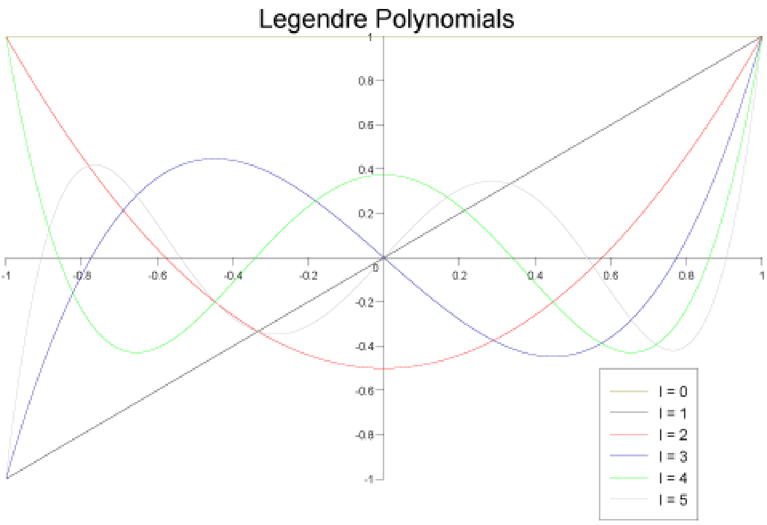
$$\text{legendre\_p}(l, x) \quad = \quad P_l(x) \quad = \quad \frac{1}{2^l\, l!} \frac{d^l}{d\,x^l} \left(x^2 - 1\right)^l \quad ; \quad \left|x\right| \le 1$$

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(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:

$$\text{legendre\_p}(l, m, x) = P_l^m(x) = (-1)^m \left(1 - x^2\right)^{\frac{m}{2}} \frac{d^m P_l(x)}{d x^m}$$

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.

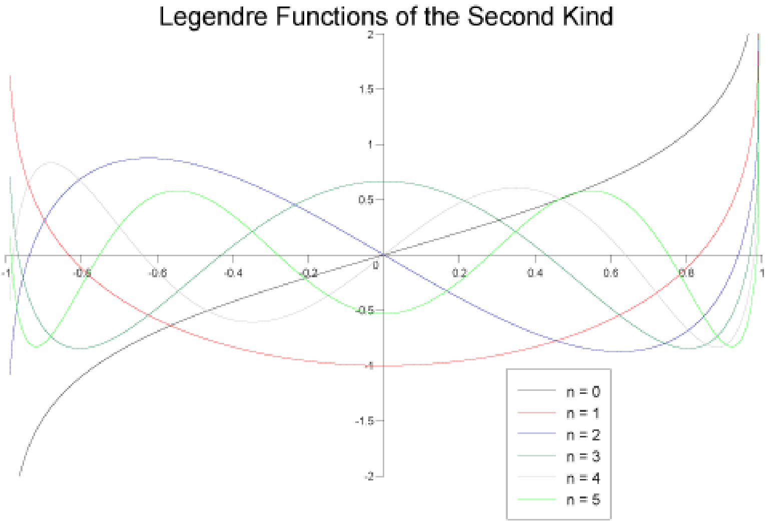
```
template <class T>
calculated-result-type legendre_q(unsigned n, T x);

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:

$$\begin{aligned} \text{legendre\_q}(0, x) &= Q_0(x) = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right) \\ \text{legendre\_q}(1, x) &= Q_1(x) = \frac{x}{2} \ln\left(\frac{1+x}{1-x}\right) - 1 \end{aligned}$$

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.

$$P_{l+1}(x) = \frac{(2l+1)x P_l(x) - l P_{l-1}(x)}{(l+1)}$$

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)).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]));
```

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*.

$$P_{l+1}^{m}(x) \quad = \quad \frac{(2l+1)\,x\,P_{l}^{m}(x) - (l+m+1)\,P_{l-1}^{m}(x)}{(l-m+1)}$$

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)).push_back(legendre_p(1 + m, m, x));
for(unsigned l = 1 + m; l < m + 10; ++l)
   v.push_back(legendre_next(l, m, x, v[l], v[l-1]));
```

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 17. Peak Errors In the Legendre P Function

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**	**Errors in range** **20 < l < 120**
53	Win32, Visual C++ 8	Peak=211 Mean=20	Peak=300 Mean=33
64	SUSE Linux IA32, g++ 4.1	Peak=70 Mean=10	Peak=700 Mean=60
64	Red Hat Linux IA64, g++ 3.4.4	Peak=70 Mean=10	Peak=700 Mean=60
113	HPUX IA64, aCC A.06.06	Peak=35 Mean=6	Peak=292 Mean=41

Table 18. Peak Errors In the Associated Legendre P Function

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**
53	Win32, Visual C++ 8	Peak=1200 Mean=7
64	SUSE Linux IA32, g++ 4.1	Peak=80 Mean=5
64	Red Hat Linux IA64, g++ 3.4.4	Peak=80 Mean=5
113	HPUX IA64, aCC A.06.06	Peak=42 Mean=4

Table 19. Peak Errors In the Legendre Q Function

Significand Size	Platform and Compiler	Errors in range 0 < l < 20	Errors in range 20 < l < 120
53	Win32, Visual C++ 8	Peak=50 Mean=7	Peak=4600 Mean=370
64	SUSE Linux IA32, g++ 4.1	Peak=51 Mean=8	Peak=6000 Mean=480
64	Red Hat Linux IA64, g++ 3.4.4	Peak=51 Mean=8	Peak=6000 Mean=480
113	HPUX IA64, aCC A.06.06	Peak=90 Mean=10	Peak=1700 Mean=140

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.

No comparisons to other libraries are shown here: there appears to be only one viable implementation method for these functions, the comparisons to other libraries that have been run show identical error rates to those given here.

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 guarentee low absolute error but cannot guarentee low relative error near one of the roots of the polynomials.

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.

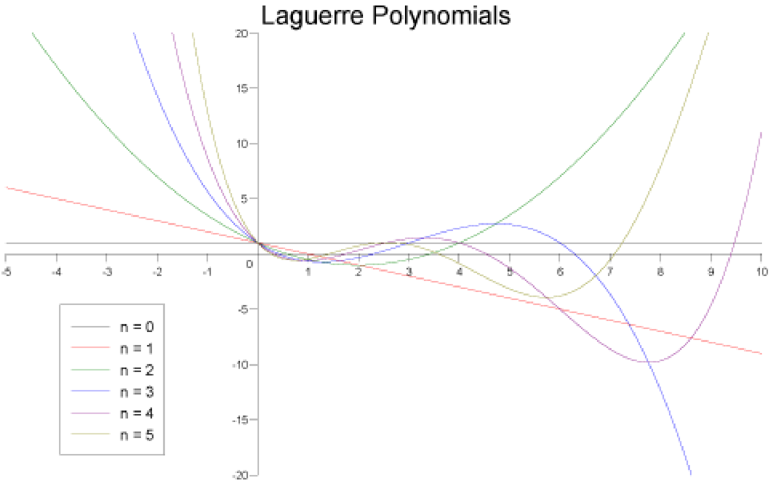
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

```
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*:

$$\text{laguerre}(n, m, x) = L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x)$$

```
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*.

$$L_{n+1}(x) \quad = \quad \frac{(2n+1-x)L_n(x) - n\,L_{n-1}(x)}{(n+1)}$$

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 20. Peak Errors In the Laguerre Polynomial

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**
53	Win32, Visual C++ 8	Peak=3000 Mean=185
64	SUSE Linux IA32, g++ 4.1	Peak=1x10^4 Mean=828
64	Red Hat Linux IA64, g++ 3.4.4	Peak=1x10^4 Mean=828
113	HPUX IA64, aCC A.06.06	Peak=680 Mean=40

Table 21. Peak Errors In the Associated Laguerre Polynomial

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**
53	Win32, Visual C++ 8	Peak=433 Mean=11
64	SUSE Linux IA32, g++ 4.1	Peak=61.4 Mean=19.5
64	Red Hat Linux IA64, g++ 3.4.4	Peak=61.4 Mean=19.5
113	HPUX IA64, aCC A.06.06	Peak=540 Mean=13.94

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 guarentee low absolute error but cannot guarentee 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.

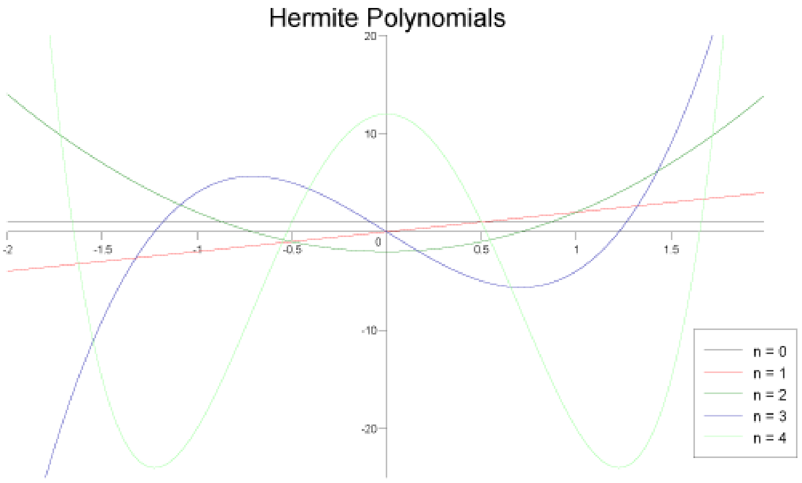
```
template <class T>
calculated-result-type hermite(unsigned n, T x);

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*:

$$\mathrm{hermite}(x) = H_n(x) = (-1)^n e^{x^2} \frac{d^2}{dx^2} e^{-x^2}$$

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 22. Peak Errors In the Hermite Polynomial

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**
53	Win32, Visual C++ 8	Peak=4.5 Mean=1.5
64	Red Hat Linux IA32, g++ 4.1	Peak=6 Mean=2
64	Red Hat Linux IA64, g++ 3.4.4	Peak=6 Mean=2
113	HPUX IA64, aCC A.06.06	Peak=6 Mean=4

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 guarentee low absolute error but cannot guarentee low relative error near one of the roots of the polynomials.

Spherical Harmonics

Synopsis

```
#include <boost/math/special_functions/spheric_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(unsigned n, int m, T1 theta, T2 phi, 

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(unsigned n, int m, T1 theta, T2 phi, 
```

Returns the value of the Spherical Harmonic Y_n^m(theta, phi):

$$Y_n^m(\theta, \phi) = \sqrt{\frac{2n+1}{4\pi} \frac{(n-m)!}{(n+m)!}} P_n^m(\cos\theta) e^{im\phi}$$

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):

$$\operatorname{Re}\left(Y_n^m(\theta, \phi)\right) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}}P_n^m(\cos\theta)\cos(m\phi)$$

```
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):

$$\operatorname{Im}\left(Y_n^m(\theta, \phi)\right) = \sqrt{\frac{2n+1}{4\pi}\frac{(n-m)!}{(n+m)!}}P_n^m(\cos\theta)\sin(m\phi)$$

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 23. Peak Errors In the Sperical Harmonic Functions

Significand Size	**Platform and Compiler**	**Errors in range** **0 < l < 20**
53	Win32, Visual C++ 8	Peak=2x10^4 Mean=700
64	SUSE Linux IA32, g++ 4.1	Peak=2900 Mean=100
64	Red Hat Linux IA64, g++ 3.4.4	Peak=2900 Mean=100
113	HPUX IA64, aCC A.06.06	Peak=6700 Mean=230

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.

Bessel Functions

Bessel Function Overview

Ordinary Bessel Functions

Bessel Functions are solutions to Bessel's ordinary differential equation:

$$z^2\frac{d^2u}{dz^2} + z\frac{du}{dz} + (z^2 - \nu^2)u = 0$$

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_v and known as a Bessel function of the first kind:

$$J_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}z^2)^k}{k!\Gamma(\nu + k + 1)}$$

This function is implemented in this library as cyl_bessel_j.

The second solution is denoted either Y_v or N_v 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:

$$J_{-\nu}(z) = \cos(\nu\pi)J_\nu(z) - \sin(\nu\pi)Y_\nu(z)$$

$$Y_{-\nu}(z) = \sin(\nu\pi)J_\nu(z) + \cos(\nu\pi)Y_\nu(z)$$

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:

$$z^2\frac{d^2u}{dz^2} + z\frac{du}{dz} - (z^2 + \nu^2)u = 0$$

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:

$$I_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{(\frac{1}{4}z^2)^k}{k!\Gamma(\nu + k + 1)}$$

$$K_\nu(z) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

These functions are implemented in this library as cyl_bessel_i and cyl_bessel_k respectively.

The modified Bessel functions satisfy the recurrence relations:

$$I_{\nu+1}(z) = -\frac{2\nu}{z}I_\nu(z) + I_{\nu-1}(z)$$

$$K_{\nu+1}(z) = \frac{2\nu}{z}K_\nu(z) + K_{\nu-1}(z)$$

Have the derivatives:

$$I'_\nu(z) = \frac{\nu}{z}I_\nu(z) + I_{\nu+1}(z)$$

$$K'_\nu(z) = \frac{\nu}{z}K_\nu(z) - K_{\nu+1}(z)$$

Have the Wronskian relation:

$$W = I_\nu(z)K'_\nu(z) - K_\nu(z)I'_\nu(z) = -\left[I_\nu(z)K_{\nu+1}(z) + K_\nu(z)I_{\nu+1}(z)\right] = -\frac{1}{z}$$

and the reflection formulae:

$$I_{-\nu}(z) = I_\nu(z) + \frac{2}{\pi}\sin(\nu\pi)K_\nu(z)$$

$$K_{-\nu}(z) = K_\nu(z)$$

Spherical Bessel Functions

When solving the Helmholtz equation in spherical coordinates by separation of variables, the radial equation has the form:

$$z^2\frac{d^2u}{dz^2} + 2z\frac{du}{dz} + [z^2 - n(n+1)]u = 0$$

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:

$$j_n(z) = \sqrt{\frac{\pi}{2z}}J_{n+\frac{1}{2}}(z)$$

$$y_n(z) = \sqrt{\frac{\pi}{2z}}Y_{n+\frac{1}{2}}(z)$$

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

```
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:

$$J_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}z^2)^k}{k!\Gamma(\nu+k+1)}$$

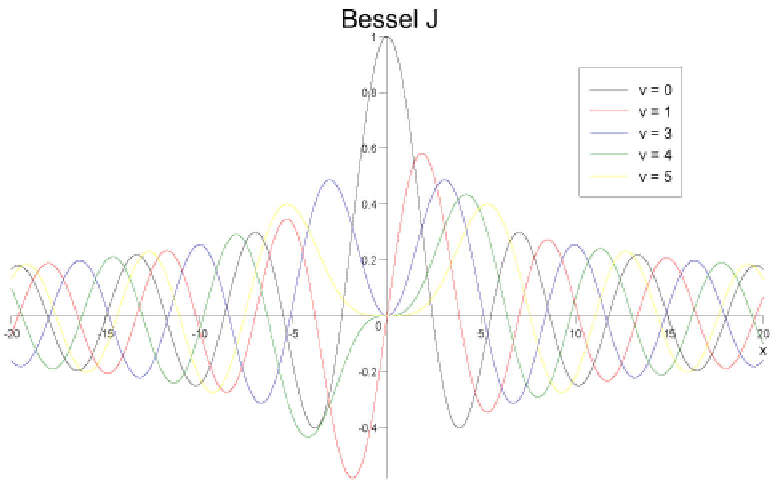
$$Y_\nu(z) = \frac{J_\nu(z)\cos(\nu\pi) - J_{-\nu}(z)}{\sin(\nu\pi)}$$

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.

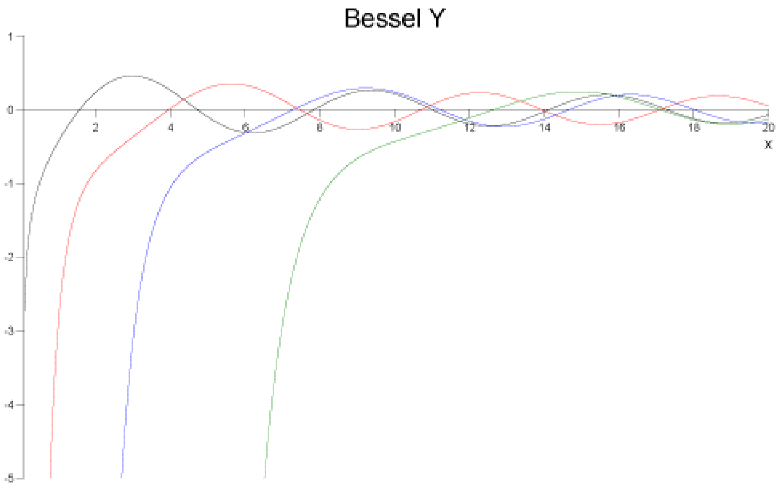
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

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 cyclic nature of J_v:



The following graph shows the behaviour of Y_v: this 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

The following tables show how the accuracy of these functions varies on various platforms, along with comparisons to the GSL-1.9 and Cephes 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.

Table 24. Errors Rates in cyl_bessel_j

Significand Size	Platform and Compiler	J_0 and J_1	J_v	J_v (large values of x > 1000)
53	Win32 / Visual C++ 8.0	Peak=2.5 Mean=1.1 GSL Peak=6.6 Cephes Peak=2.5 Mean=1.1	Peak=11 Mean=2.2 GSL Peak=11 Cephes Peak=17 Mean=2.5	Peak=413 Mean=110 GSL Peak=6x10^{11} Cephes Peak=2x10^5
64	Red Hat Linux IA64 / G++ 3.4	Peak=7 Mean=3	Peak=117 Mean=10	Peak=2x10^4 Mean=6x10^3
64	SUSE Linux AMD64 / G++ 4.1	Peak=7 Mean=3	Peak=400 Mean=40	Peak=2x10^4 Mean=1x10^4
113	HP-UX / HP aCC 6	Peak=14 Mean=6	Peak=29 Mean=3	Peak=2700 Mean=450

Table 25. Errors Rates in cyl_neumann

Significand Size	Platform and Compiler	J_0 and J_1	J_n (integer orders)	J_v (fractional orders)
53	Win32 / Visual C++ 8.0	Peak=330 Mean=54 GSL Peak=34 Mean=9 Cephes Peak=330 Mean=54	Peak=923 Mean=83 GSL Peak=500 Mean=54 Cephes Peak=923 Mean=83	Peak=561 Mean=36 GSL Peak=1.4x10^6 Mean=7x10^4 Cephes Peak=+INF
64	Red Hat Linux IA64 / G++ 3.4	Peak=470 Mean=56	Peak=843 Mean=51	Peak=741 Mean=51
64	SUSE Linux AMD64 / G++ 4.1	Peak=1300 Mean=424	Peak=2x10^4 Mean=8x10^3	Peak=1x10^5 Mean=6x10^3
113	HP-UX / HP aCC 6	Peak=180 Mean=63	Peak=340 Mean=150	Peak=2x10^4 Mean=1200

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.

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*:

$$J_{-\nu}(z) = \cos(\nu\pi)J_\nu(z) - \sin(\nu\pi)Y_\nu(z)$$

$$Y_{-\nu}(z) = \sin(\nu\pi)J_\nu(z) + \cos(\nu\pi)Y_\nu(z)$$

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 small, J_x is best computed directly from the series:

$$J_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{(-\frac{1}{4}z^2)^k}{k!\Gamma(\nu+k+1)}$$

In the general case we compute J_v and Y_v simultaneously.

To get the initial values, 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 $J_\mu(x)$, $J_{\mu+1}(x)$, $Y_\mu(x)$, $Y_{\mu+1}(x)$ 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:

$$W = J_\nu(z)Y'_\nu(z) - Y_\nu(z)J'_\nu(z) = Y_\nu(z)J_{\nu+1}(z) - J_\nu(z)Y_{\nu+1}(z) = \frac{2}{\pi z}$$

$$\mathrm{CF1}: \quad f_\nu = \frac{J_{\nu+1}}{J_\nu} = \frac{1}{\frac{2(\nu+1)}{x} -} \; \frac{1}{\frac{2(\nu+2)}{x} -} \cdots$$

$$\mathrm{CF2}: \quad p + iq = \frac{J'_\nu + iY'_\nu}{J_\nu + iY_\nu} = \left(i - \frac{1}{2x}\right) + \frac{i}{x}\left[\frac{\left(\frac{1}{2}\right)^2 - \nu^2}{2(x+i)+} \; \frac{\left(\frac{3}{2}\right)^2 - \nu^2}{2(x+2i)+} \cdots\right]$$

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_v 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_v is calculated from the forward recurrence, so is Y_{v+1}. With these two values and f_v, the Wronskian yields $J_v(x)$ directly without backward recurrence.

Modified Bessel Functions of the First and Second Kinds

Synopsis

```
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:

$$I_\nu(z) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

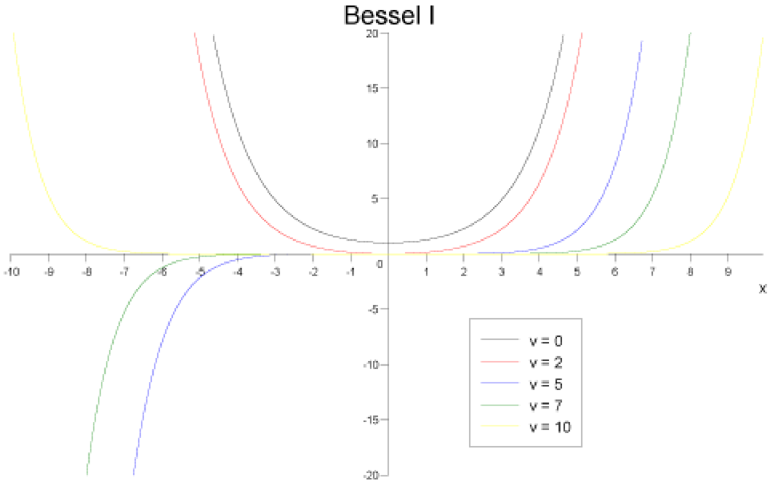
$$K_\nu(z) = \frac{\pi}{2} \cdot \frac{I_{-\nu}(z) - I_\nu(z)}{\sin(\nu\pi)}$$

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.

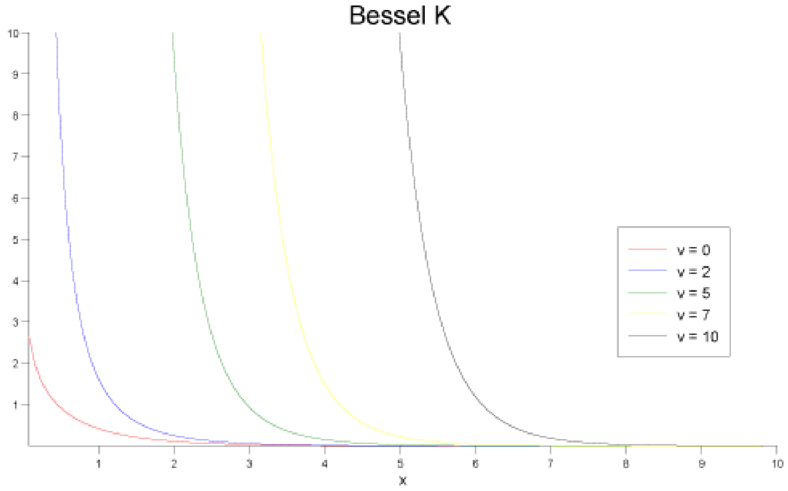
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

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 a comparison to the GSL-1.9 library. 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 26. Errors Rates in cyl_bessel_i

Significand Size	Platform and Compiler	I_v
53	Win32 / Visual C++ 8.0	Peak=10 Mean=3.4 GSL Peak=6000
64	Red Hat Linux IA64 / G++ 3.4	Peak=11 Mean=3
64	SUSE Linux AMD64 / G++ 4.1	Peak=11 Mean=4
113	HP-UX / HP aCC 6	Peak=15 Mean=4

Table 27. Errors Rates in cyl_bessel_k

Significand Size	Platform and Compiler	K_v
53	Win32 / Visual C++ 8.0	Peak=9 Mean=2 GSL Peak=9
64	Red Hat Linux IA64 / G++ 3.4	Peak=10 Mean=2
64	SUSE Linux AMD64 / G++ 4.1	Peak=10 Mean=2
113	HP-UX / HP aCC 6	Peak=12 Mean=5

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 minimax rational approximations on finite and infinite intervals. The coefficients are from:

- J.M. Blair and C.A. Edwards, *Stable rational minimax approximations to the modified Bessel functions I_0(x) and I_1(x)*, Atomic Energy of Canada Limited Report 4928, Chalk River, 1974.
- S. Moshier, *Methods and Programs for Mathematical Functions*, Ellis Horwood Ltd, Chichester, 1989.

While the 0.5 case is a simple trigonometric function:

$I_{0.5}(x)$ = sqrt(2 / πx) * sinh(x)

For K_v with *v* an integer, the result is calculated using the recurrence relation:

$$K_{\nu+1}(z) = \frac{2\nu}{z} K_\nu(z) + K_{\nu-1}(z)$$

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.

In the general case, we first normalize ν to [0, [inf]) with the help of the reflection formulae:

$$I_{-\nu}(z) = I_\nu(z) + \frac{2}{\pi}\sin(\nu\pi) K_\nu(z)$$

$$K_{-\nu}(z) = K_\nu(z)$$

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:

$$\mathrm{CF1}: \quad f_\nu = \frac{I_{\nu+1}}{I_\nu} = \frac{1}{\frac{2(\nu+1)}{x}+} \; \frac{1}{\frac{2(\nu+2)}{x}+} \cdots$$

$$\mathrm{CF2}: \quad \frac{z_1}{z_0} = \frac{1}{2(x+1)+} \; \frac{\nu^2 - \left(\frac{3}{2}\right)^2}{2(x+2)+} \; \frac{\nu^2 - \left(\frac{5}{2}\right)^2}{2(x+3)+} \cdots$$

$$W = I_\nu(z)K'_\nu(z) - K_\nu(z)I'_\nu(z) = -\left[I_\nu(z)K_{\nu+1}(z) + K_\nu(z)I_{\nu+1}(z)\right] = -\frac{1}{z}$$

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

$$K_\mu = \sqrt{\pi}\,(2x)^\mu\, e^{-x} z_0$$

$$K_{\mu+1} = \frac{K_\mu}{x}\left[\frac{1}{2} + \mu + x + \left(\mu^2 - \frac{1}{4}\right)\frac{z_1}{z_0}\right]$$

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:

$$K_\mu = \sum_{k=0}^{\infty} c_k f_k$$

$$K_{\mu+1} = \frac{2}{x}\sum_{k=0}^{\infty} c_k h_k$$

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 |μ| <= 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

```
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:

$$j_n(z) = \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z)$$

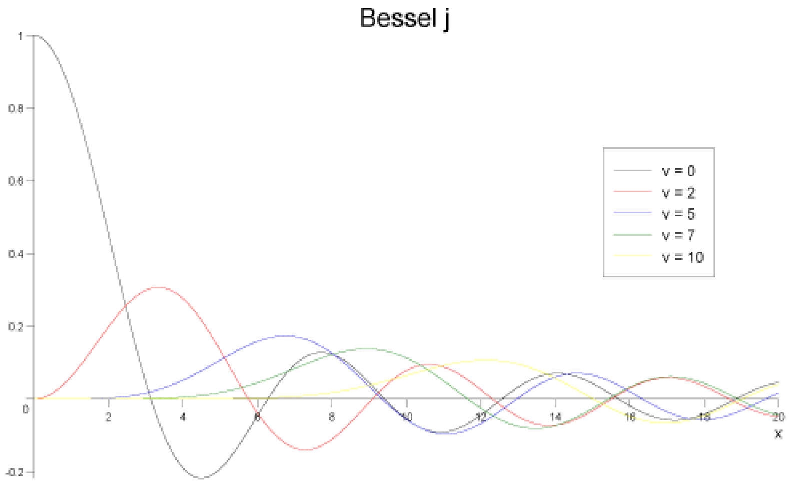
$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z)$$

The return type of these functions is computed using the *result type calculation rules* for the single argument type T.

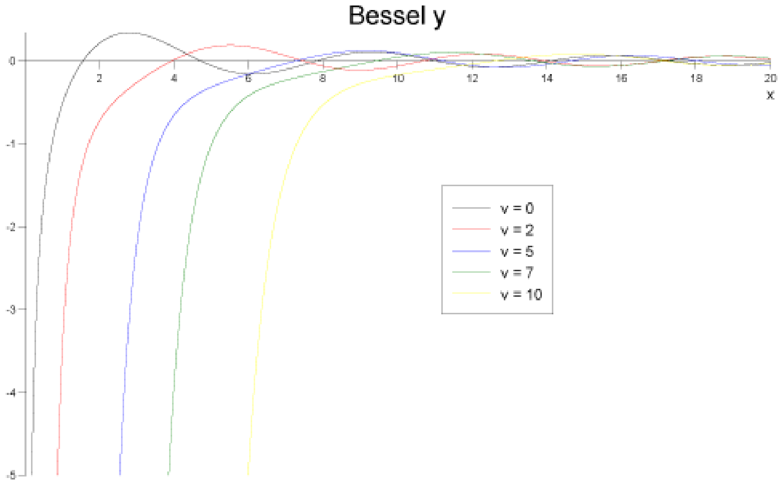
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

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

Other than for some special cases, these functions are computed in terms of cyl_bessel_j and cyl_neumann: refer to these functions for accuracy data.

Implementation

Other than error handling and a couple of special cases these functions are implemented directly in terms of their definitions:

$$j_n(z) = \sqrt{\frac{\pi}{2z}} J_{n+\frac{1}{2}}(z)$$

$$y_n(z) = \sqrt{\frac{\pi}{2z}} Y_{n+\frac{1}{2}}(z)$$

The special cases occur for:

j_0= sinc_pi(x) = sin(x) / x

and for small *x < 1*, we can use the series:

$$j_\nu(z) = \sqrt{\frac{\pi}{4}} \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z}{2}\right)^{2k+n}}{k!\Gamma(n+k+1+\frac{1}{2})}$$

which neatly avoids the problem of calculating 0/0 that can occur with the main definition as x → 0.

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.

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 can not 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)

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

Elliptic Integral of the Second Kind (Legendre form)

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

Elliptic Integral of the Third Kind (Legendre form)

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

where

$$k = \sin \alpha, \quad |k| \leq 1$$

Note

ϕ is called the amplitude.

k is called the modulus.

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 = sinα

m = k^2 = $\sin^2$α

So that the integral of the third kind (for example) may be expressed as either:

Π(n, ϕ, k)

Π(n, ϕ \ α)

Π(n, ϕ| m)

To further complicate matters, some texts refer to the *complement of the parameter m*, or 1 - m, where:

1 - m = 1 - k^2 = $\cos^2$α

This implementation uses *k* throughout: this matches the requirements of the Technical Report on C++ Library Extensions. However, you should be extra careful when using these functions!

When $\phi = \pi / 2$, the elliptic integrals are called *complete*.

Complete Elliptic Integral of the First Kind (Legendre form)

$$K(k) = F(\frac{\pi}{2}, k) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}$$

Complete Elliptic Integral of the Second Kind (Legendre form)

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

Complete Elliptic Integral of the Third Kind (Legendre form)

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

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

$R_C(x, y) = R_F(x, y, y)$

$R_D(x, y, z) = R_J(x, y, z, z)$

Duplication Theorem

Carlson proved in [Carlson78] that

$$\begin{aligned} R_F(x,y,z) &= 2R_F(x+\lambda, y+\lambda, z+\lambda) \\ &= R_F\left(\frac{x+\lambda}{4}, \frac{y+\lambda}{4}, \frac{z+\lambda}{4}\right) \\ \lambda &= \sqrt{xy} + \sqrt{yz} + \sqrt{zx} \end{aligned}$$

Carlson's Formulas

The Legendre form and Carlson form of elliptic integrals are related by equations:

$$\begin{aligned} F(\phi,k) &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ E(\phi,k) &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) - \frac{1}{3}k^2\sin^3\phi R_D(\cos^2\phi, 1-k^2\sin^2\phi, 1) \\ \Pi(n,\phi,k) &= \sin\phi R_F(\cos^2\phi, 1-k^2\sin^2\phi, 1) + \frac{1}{3}n\sin^3\phi R_J(\cos^2\phi, 1-k^2\sin^2\phi, 1, 1-n\sin^2\phi) \end{aligned}$$

In particular,

$$\begin{aligned} K(k) &= R_F(0, 1-k^2, 1) \\ E(k) &= R_F(0, 1-k^2, 1) - \frac{1}{3}k^2 R_D(0, 1-k^2, 1) \\ \Pi(n,k) &= R_F(0, 1-k^2, 1) + \frac{1}{3}nR_J(0, 1-k^2, 1, 1-n) \end{aligned}$$

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, *Traitd 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. B.C. Carlson, *Computing elliptic integrals by duplication*, Numerische Mathematik, vol 33, 1 (1979).

3. B.C. Carlson, *Elliptic Integrals of the First Kind*, SIAM Journal on Mathematical Analysis, vol 8, 231 (1977).

4. B.C. Carlson, *Short Proofs of Three Theorems on Elliptic Integrals*, SIAM Journal on Mathematical Analysis, vol 9, 524 (1978).

5. B.C. Carlson and E.M. Notis, *ALGORITHM 577: Algorithms for Incomplete Elliptic Integrals*, ACM Transactions on Mathematmal Software, vol 7, 398 (1981).

6. B. C. Carlson, *On computing elliptic integrals and functions*. J. Math. and Phys., 44 (1965), pp. 36-51.

7. B. C. Carlson, *A table of elliptic integrals of the second kind*. Math. Comp., 49 (1987), pp. 595-606. (Supplement, ibid., pp. S13-S17.)

8. B. C. Carlson, *A table of elliptic integrals of the third kind*. Math. Comp., 51 (1988), pp. 267-280. (Supplement, ibid., pp. S1-S5.)

9. B. C. Carlson, *A table of elliptic integrals: cubic cases*. Math. Comp., 53 (1989), pp. 327-333.

10. B. C. Carlson, *A table of elliptic integrals: one quadratic factor*. Math. Comp., 56 (1991), pp. 267-280.

11. B. C. Carlson, *A table of elliptic integrals: two quadratic factors*. Math. Comp., 59 (1992), pp. 165-180.

12. B. C. Carlson, *Numerical computation of real or complex elliptic integrals*. Numerical Algorithms, Volume 10, Number 1 / March, 1995, p13-26.

13. 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.

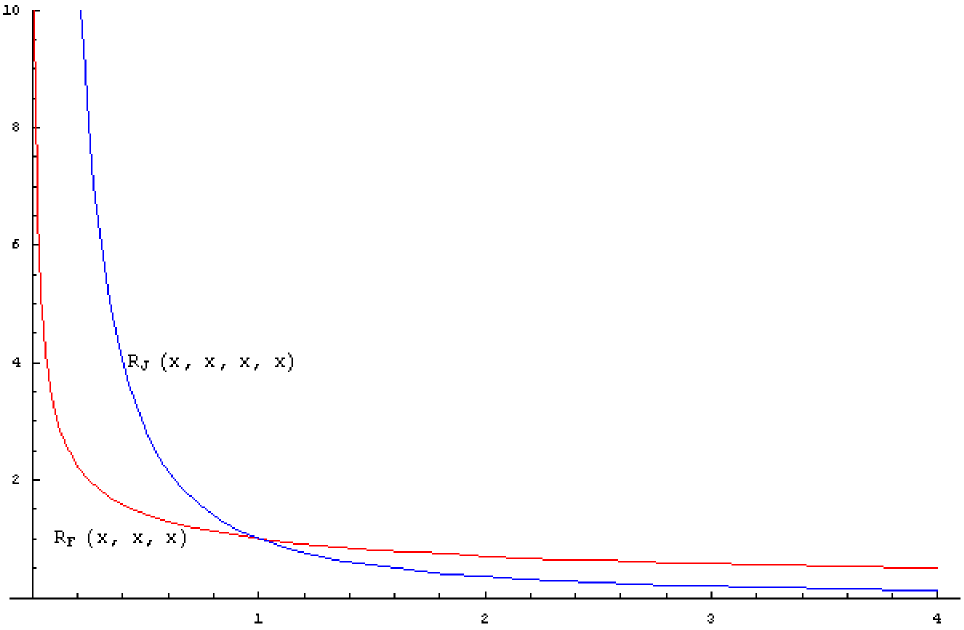
Elliptic Integrals - Carlson Form

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
```

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:

$$\begin{aligned} (y+q)R_J(x,\ y,\ z,\ -q) \quad &= \quad (p-y)R_J(x,\ y,\ z,\ p) - 3R_F(x,\ y,\ z) \\ &\quad +3\left(\frac{x\,y\,z}{x\,z+p\,q}\right)^{\frac{1}{2}} R_C(x\,z+p\,q,\ p\,q) \\ \text{with: } q > 0 \qquad &\qquad \text{and: } p = y + \frac{(z-y)(y-x)}{(y+q)} \end{aligned}$$

```
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:

$$R_C(x,\ -y) \quad = \quad \left(\frac{x}{x+y}\right)^{\frac{1}{2}} R_C(x+y,\ y)$$

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 28. Errors Rates in the Carlson Elliptic Integrals

Significand Size	Platform and Compiler	R_F	R_D	R_J	R_C
53	Win32 / Visual C++ 8.0	Peak=2.9 Mean=0.75	Peak=2.6 Mean=0.9	Peak=108 Mean=6.9	Peak=2.4 Mean=0.6
64	Red Hat Linux / G++ 3.4	Peak=2.5 Mean=0.75	Peak=2.7 Mean=0.9	Peak=105 Mean=8	Peak=1.9 Mean=0.7
113	HP-UX / HP aCC 6	Peak=5.3 Mean=1.6	Peak=2.9 Mean=0.99	Peak=180 Mean=12	Peak=1.8 Mean=0.7

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.

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:

$$\begin{aligned} (y+q)R_J(x,\ y,\ z,\ -q) \quad &= \quad (p-y)R_J(x,\ y,\ z,\ p) - 3R_F(x,\ y,\ z) \\ &\quad +3\left(\frac{x\,y\,z}{x\,z+p\,q}\right)^{\frac{1}{2}} R_C(x\,z+p\,q,\ p\,q) \\ \text{with: } q > 0 \quad & \quad \text{and: } p = y + \frac{(z-y)(y-x)}{(y+q)} \end{aligned}$$

$$R_C(x, \; -y) \quad = \quad \left(\frac{x}{x+y}\right)^{\frac{1}{2}} R_C(x+y, \; y)$$

Elliptic Integrals of the First Kind - Legendre Form

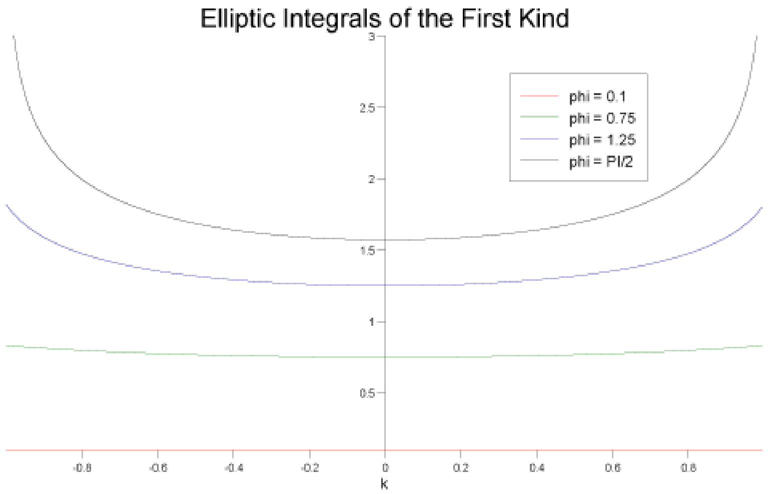
Synopsis

```
#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)*:

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

Requires -1 <= 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.

```
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)*:

$$K(k) = F(\frac{\pi}{2}, k) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}$$

Requires -1 <= 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 29. Errors Rates in the Elliptic Integrals of the First Kind

Significand Size	Platform and Compiler	F(φ, k)	K(k)
53	Win32 / Visual C++ 8.0	Peak=3 Mean=0.8	Peak=1.8 Mean=0.7
64	Red Hat Linux / G++ 3.4	Peak=2.6 Mean=1.7	Peak=2.2 Mean=1.8
113	HP-UX / HP aCC 6	Peak=4.6 Mean=1.5	Peak=3.7 Mean=1.5

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:

$$\begin{aligned} F(-\phi, k) &= -F(\phi, k) \\ F(\phi + m\pi, k) &= F(\phi, k) + 2m K(k) \\ F(\phi, k) &= \sin\phi \, R_F\left(\cos^2\phi, 1 - k^2 \sin^2\phi, 1\right) \end{aligned}$$

and

$$K(k) = R_F\left(0, 1 - k^2, 1\right)$$

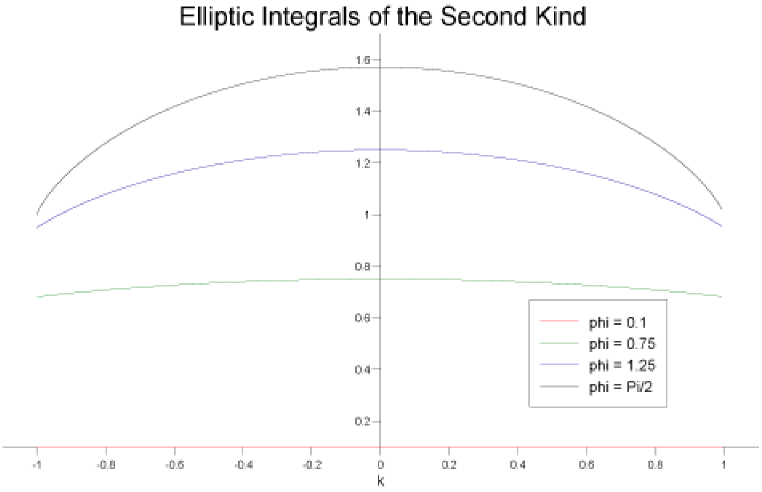
Elliptic Integrals of the Second Kind - Legendre Form

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)*:

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

Requires -1 <= 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.

```
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 first kind *E(k)*:

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

Requires -1 <= 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 30. Errors Rates in the Elliptic Integrals of the Second Kind

Significand Size	Platform and Compiler	F(ϕ, k)	K(k)
53	Win32 / Visual C++ 8.0	Peak=4.6 Mean=1.2	Peak=3.5 Mean=1.0
64	Red Hat Linux / G++ 3.4	Peak=4.3 Mean=1.1	Peak=4.6 Mean=1.2
113	HP-UX / HP aCC 6	Peak=5.8 Mean=2.2	Peak=10.8 Mean=2.3

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:

$$\begin{aligned} E(-\phi, k) &= -E(\phi, k) \\ E(\phi + m\pi, k) &= E(\phi, k) + 2mE(k) \quad ; \quad \phi \notin [0, \frac{\pi}{2}] \\ E(\phi, k) &= \sin\phi\, R_F\left(\cos^2\phi,\ 1 - k^2\sin^2\phi,\ 1\right) - \frac{1}{3}k^2\sin^3\phi\, R_D\left(\cos^2\phi,\ 1 - k^2\sin^2\phi,\ 1\right) \end{aligned}$$

and

$$E(k) = R_F\left(0,1 - k^2,\ 1\right) - \frac{1}{3}k^2 R_D\left(0,1 - k^2,\ 1\right)$$

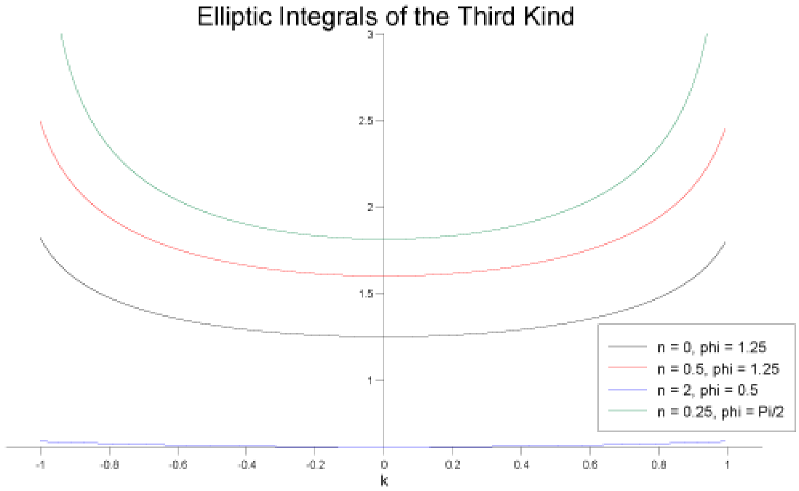
Elliptic Integrals of the Third Kind - Legendre Form

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 $\Pi(n, \phi, k)$ and its complete counterpart $\Pi(n, k) = E(n, \pi/2, k)$.



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, 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 $\Pi(n, \phi, k)$:

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

Requires $-1 <= k <= 1$ 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.

Caution

In addition, the region where $n > 1$ and ϕ *is not in the range* $[0, \pi/2]$ is currently unsupported and returns the result of domain_error. For this reason it is recomended that you keep ϕ inside its "natural" range of $[0, \pi/2]$.

```
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 *Π(n, k)*:

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

Requires *-1* <= *k* <= *1* and *n* < *1*, otherwise returns the result of domain_error (outside this range the result would be complex).

[opitonal_policy]

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 31. Errors Rates in the Elliptic Integrals of the Third Kind

Significand Size	Platform and Compiler	Π(n, ϕ, k)	Π(n, k)
53	Win32 / Visual C++ 8.0	Peak=29 Mean=2.2	Peak=3 Mean=0.8
64	Red Hat Linux / G++ 3.4	Peak=14 Mean=1.3	Peak=2.3 Mean=0.8
113	HP-UX / HP aCC 6	Peak=10 Mean=1.4	Peak=4.2 Mean=1.1

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:

Π(0, ϕ, k) = F(ϕ, k)

Π(n, π/2, k) = Π(n, k)

and

$$\begin{aligned}\prod(n,\ \phi,\ 0) &= \sqrt{\frac{1}{1-n}}\tan^{-1}\left(\sqrt{1-n}\tan\phi\right) & ; n < 1\\ &= \sqrt{\frac{1}{n-1}}\tanh^{-1}\left(\sqrt{n-1}\tan\phi\right) & ; n > 1\\ &= \tan\phi & ; n = 1\end{aligned}$$

Then if n < 0 the relations (A&S 17.7.15/16):

$$\begin{aligned}
\sqrt{(1-n)\left(1-\frac{k^2}{n}\right)}\prod(n,\ \phi,\ k) &= \sqrt{(1-N)\left(1-\frac{k^2}{N}\right)}\prod(N,\ \phi,\ k) \\
&+ \frac{k^2}{p_2}F(\phi,\ k) \\
&+ \tan^{-1}\left(\frac{p_2}{2}\frac{\sin 2\,\phi}{\Delta(\phi)}\right) \\
N &= \frac{\left(k^2-n\right)}{(1-n)} \\
p_2 &= \sqrt{-\frac{n}{1-n}\left(k^2-n\right)}
\end{aligned}$$

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)

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:

$$\prod(n,\ \phi,\ k) = \sin\phi\,R_F\left(\cos^2\phi,\ 1-k^2\sin^2\phi,\ 1\right)+\frac{n}{3}\sin^3\phi\,R_J\left(\cos^2\phi,\ 1-k^2\sin^2\phi,\ 1,1-n\sin^2\phi\right)$$

and

$$\prod(n,\ k) = R_F\left(0,1-k^2,\ 1\right)+\frac{n}{3}R_J\left(0,1-k^2,\ 1,1-n\right)$$

The remaining problem area occurs when n > 1 and ϕ is outside the range [0, π/2]. In this range the reduction formula for large ϕ can no longer be applied. Likewise the identities 17.7.7/8 in A&S for reducing n to the range [0,1] appear to be no longer applicable.

Logs, Powers, Roots and Exponentials

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.

Accuracy

For built in floating point types `log1p` should have approximately 1 epsilon accuracy.

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.

Accuracy

For built in floating point types `expm1` should have approximately 1 epsilon accuracy.

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.

Accuracy

For built in floating-point types `cbrt` should have approximately 2 epsilon accuracy.

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`.

Accuracy

For built in floating-point types `sqrt1pm1` should have approximately 3 epsilon accuracy.

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`.

Accuracy

Should have approximately 2-3 epsilon accuracy.

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 $\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$ 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 $\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$ 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 * ε >= y* we can simply return *x*.

Otherwise the result is given by:

$$\mathrm{hypot}(x, y) = x\sqrt{1 + \left(\frac{y}{x}\right)^2}$$

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions

Sinus Cardinal and Hyperbolic Sinus Cardinal Functions Overview

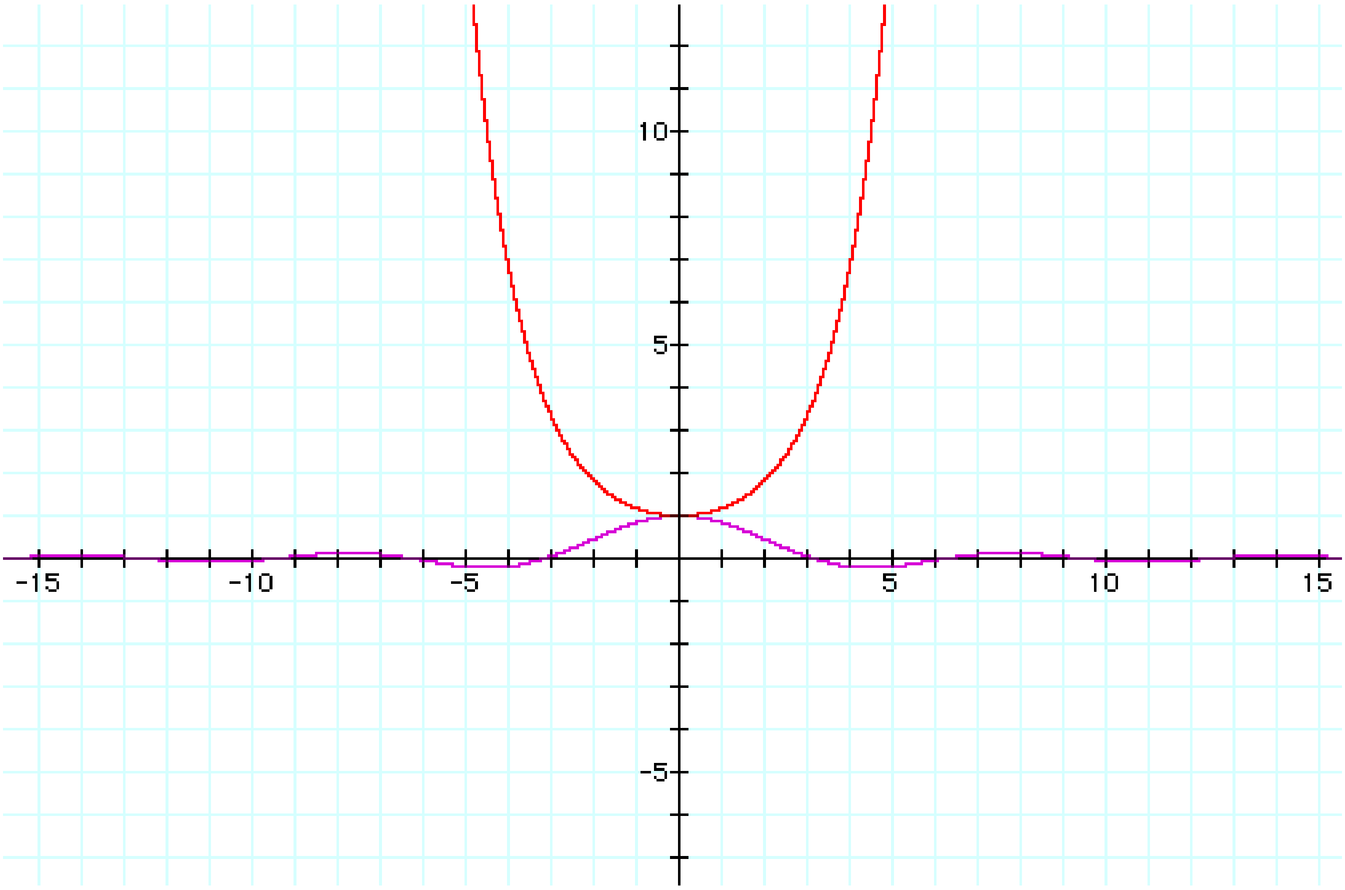
The Sinus Cardinal family of functions (indexed by the family of indices `a > 0`) is defined by $\mathrm{sinc}_a(x) = \frac{\sin\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$; it sees heavy use in signal processing tasks.

By analogy, the Hyperbolic Sinus Cardinal family of functions (also indexed by the family of indices `a > 0`) is defined by $\mathrm{sinhc}_a(x) = \frac{\sinh\left(\frac{\pi x}{a}\right)}{\frac{\pi x}{a}}$.

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.

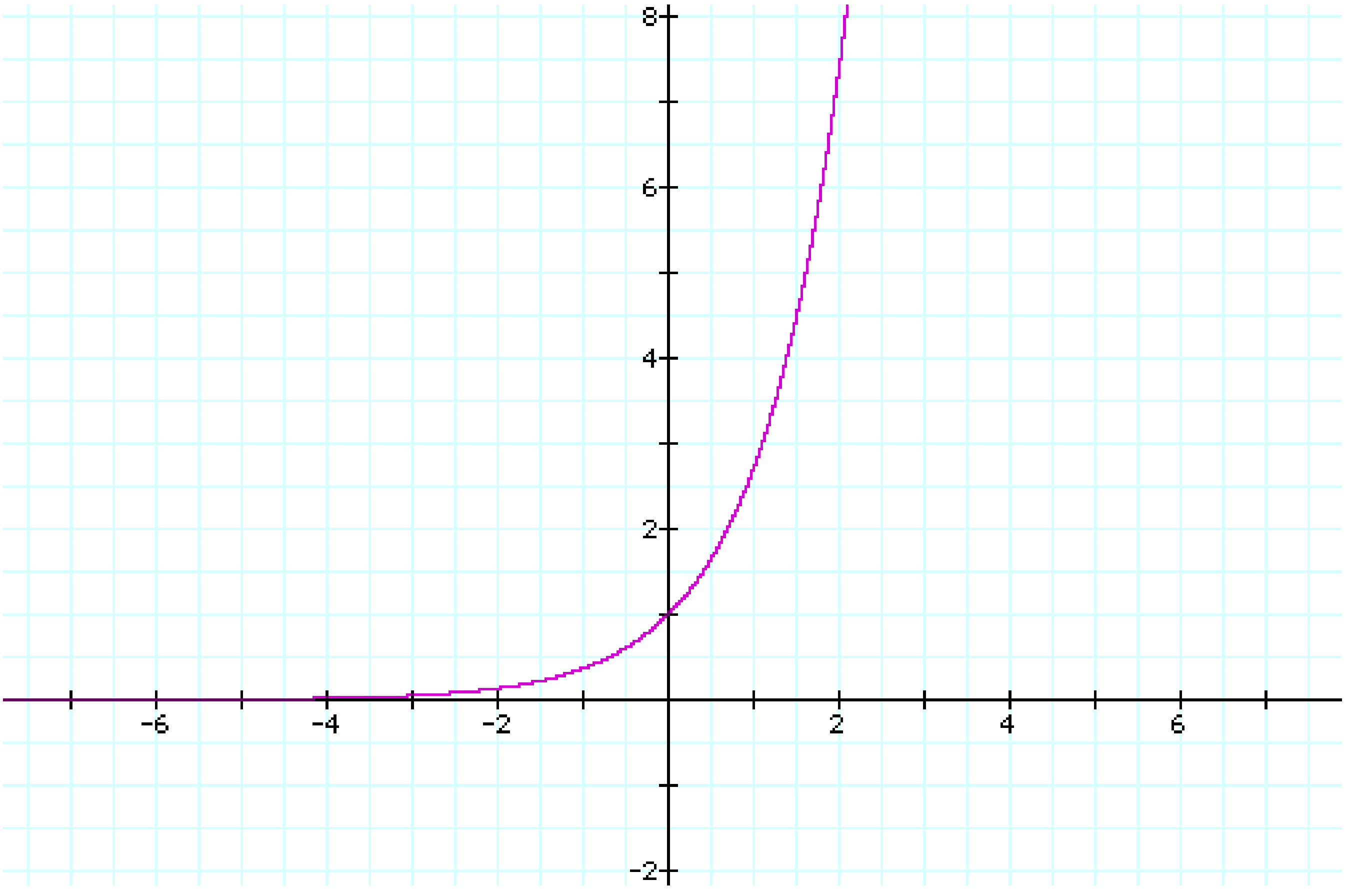
The final Policy argument is optional and can be used to control the behaviour of the function: how it handles errors, what level of precision to use etc. Refer to the policy documentation for more details.

Inverse Hyperbolic Functions

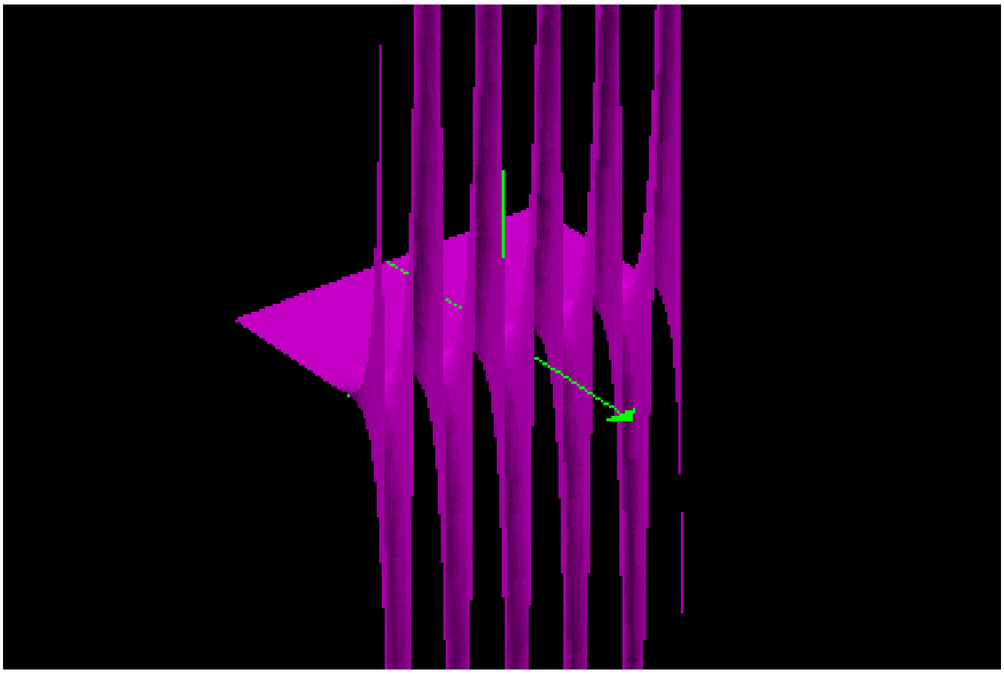
Inverse Hyperbolic Functions Overview

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



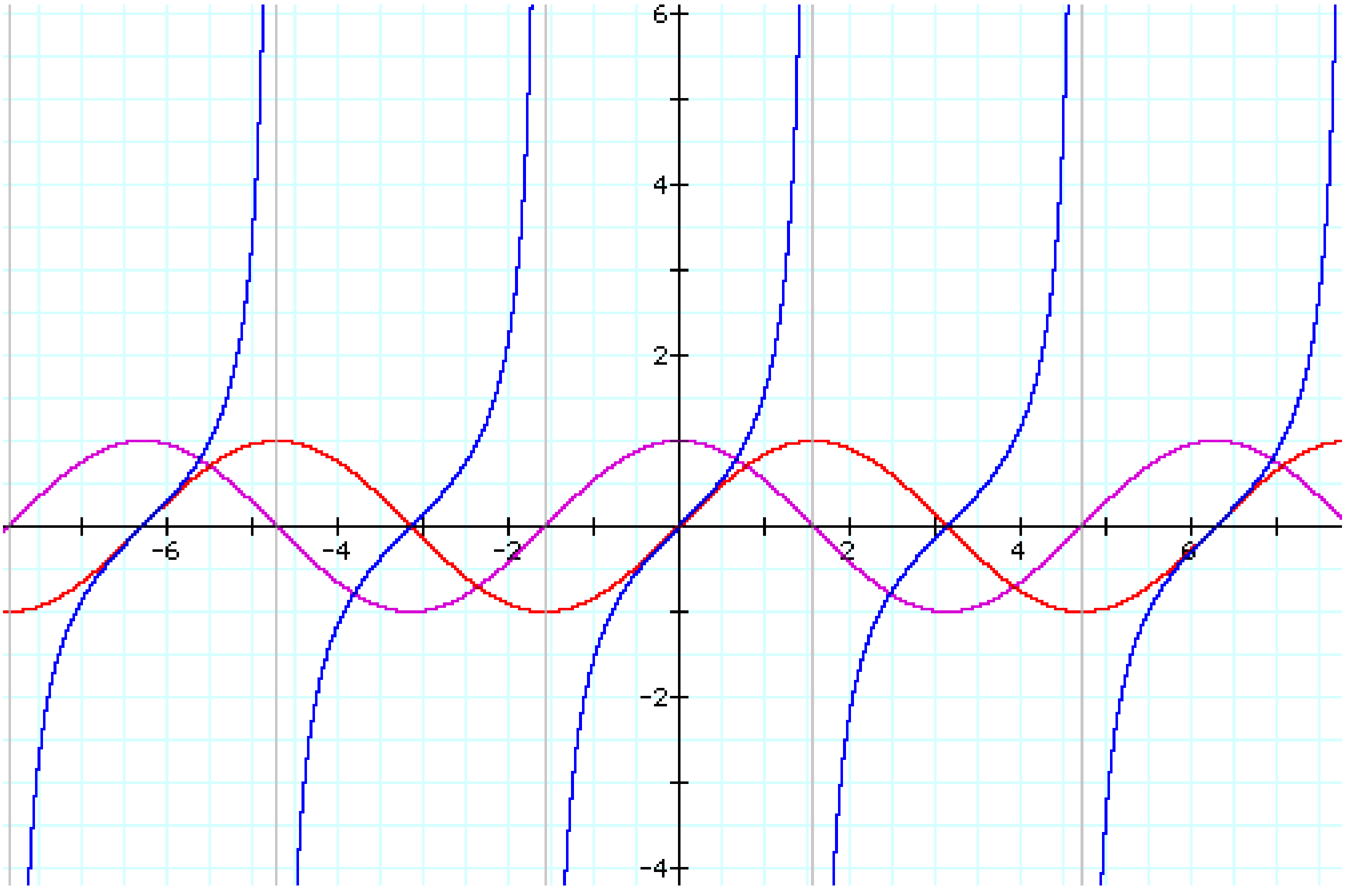
The hyperbolic functions are defined as power series which can be computed (for reals, complex, quaternions and octonions) as:

Hyperbolic cosine: $\cosh(x) = \frac{\exp(+x) + \exp(-x)}{2}$

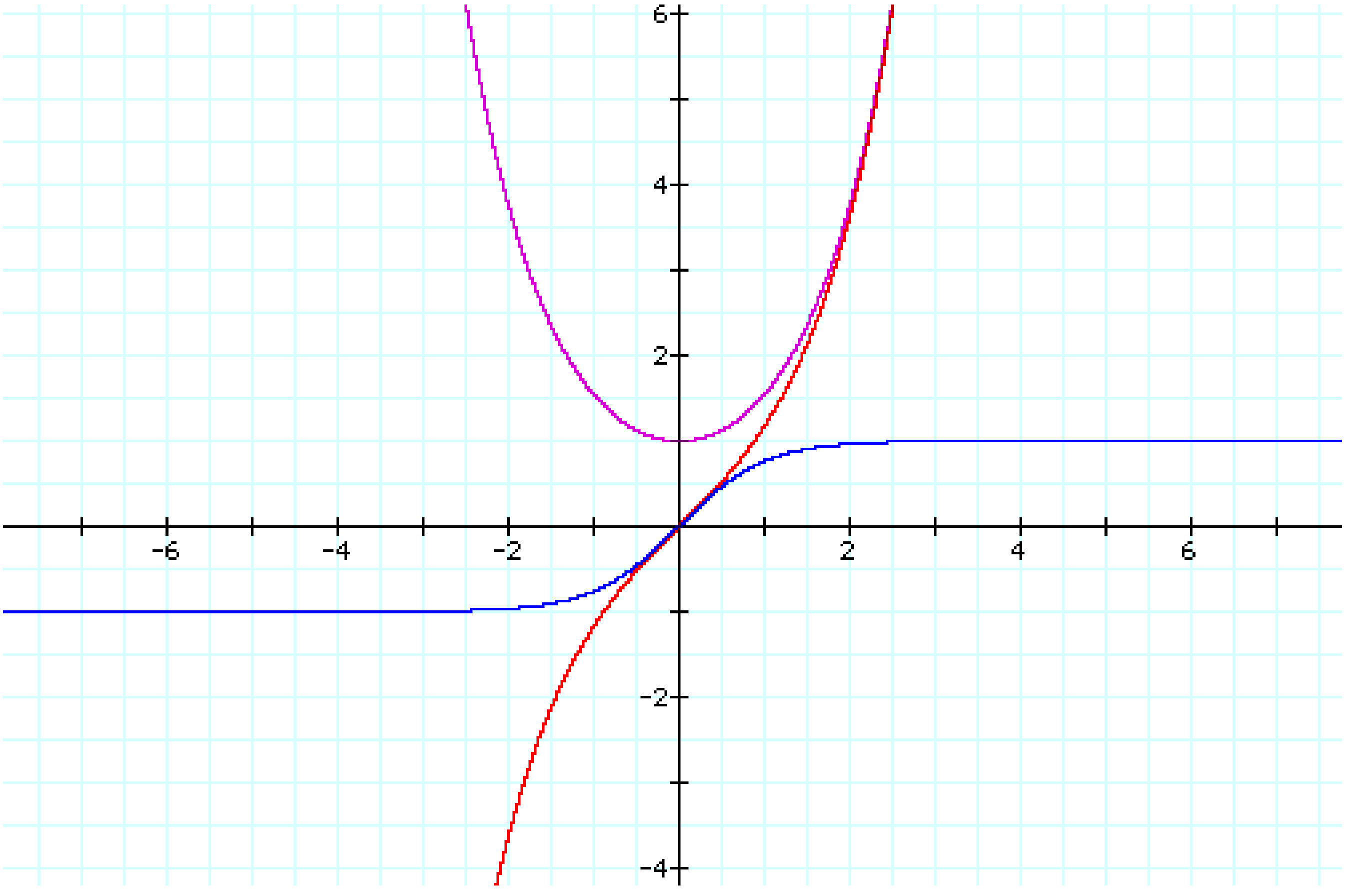
Hyperbolic sine: $\sinh(x) = \dfrac{\exp(+x) - \exp(-x)}{2}$

Hyperbolic tangent: $\tanh(x) = \dfrac{\sinh(x)}{\cosh(x)}$

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 $\operatorname{atanh}(x) = \frac{\log\left(\frac{1+x}{1-x}\right)}{2}$.

The inverse of the hyperbolic sine is called the Argument hyperbolic sine, and can be computed (for `[-1;-1+ [`) as $\operatorname{asinh}(x) = \log\left(x + \sqrt{x^2 + 1}\right)$.

The inverse of the hyperbolic cosine is called the Argument hyperbolic cosine, and can be computed as $\operatorname{asinh}(x) = \log\left(x + \sqrt{x^2 - 1}\right)$.

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. Generalised Taylor series are used near 1 and Laurent series are used near the infinity to ensure accuracy.

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.

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. Taylor series are used at the origin and Laurent series are used near the infinity to ensure accuracy.

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.

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. 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.

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+ [`, then the result of -overflow_error is returned, with ε denoting numeric_limits<T>::epsilon().

If x is in the range `]+1- ;+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.

Floating Point Classification: Infinities and NaN's

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);

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

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

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

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 namespace qualification.
(boost::math::isnan)(z);
//
// This will cause a compiler error is isnan is a native macro:
boost::math::isnan(z);
// So always use (boost::math::isnan)(z); instead.
```

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).

Internal Details and Tools (Experimental)

Overview

This section contains internal utilities used by the library's implementation along with tools used in development and testing. These tools have only minimal documentation, and crucially *do not have stable interfaces*.

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.

Reused Utilities

Series Evaluation

Synopsis

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


namespace boost{ namespace math{ namespace tools{

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(Functor& func, int bits, boost::uintmax_t& max_terms

template <class Functor>
typename Functor::result_type kahan_sum_series(Functor& func, int bits);

template <class Functor>
typename Functor::result_type kahan_sum_series(Functor& func, int bits, boost::uintmax_t& max_

}}} // 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 number of binary bits of precision required, summation will stop when the next term is too small to have any effect on the first *bits* bits of the result.

The optional 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.

Example

Let's suppose we want to implement *log(1+x)* via its infinite series,

$$\log(1+x) \quad = \quad \sum_{k=1}^{\infty} \frac{(-1)^{k-1} z^k}{k}$$

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);
   // and add it up, with enough digits for full machine precision
   // plus a couple more for luck.... !
   return tools::sum_series(s, tools::digits(x) + 2);
}
```

Continued Fraction Evaluation

Synopsis

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


namespace boost{ namespace math{ namespace tools{

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:

$$\cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4} + \ldots}}}$$

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 type, or a std::pair<> of arithmetic 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 *bits* parameter is the number of bits precision desired in the result, evaluation of the fraction will continue until the last term evaluated leaves the first *bits* bits in the result unchanged.

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

The golden ratio phi = 1.618033989... can be computed from the simplest continued fraction of all:

$$\text{Golden Ratio} \quad = \quad \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\dots}}}}$$

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:

```
continued_fraction_a(
   golden_ratio_fraction<double>(),
   std::numeric_limits<double>::digits);
```

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:

$$\tan(z) = \cfrac{z}{1 - \cfrac{z^2}{3 - \cfrac{z^2}{5 - \cfrac{z^2}{7 - \cfrac{z^2}{9 - \dots}}}}}$$

So it's 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>::digits);
}
```

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.

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.

Root Finding With Derivatives

Synopsis

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


namespace boost{ namespace math{ namespace tools{

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);

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);

template <class F, class T>
T schroeder_iterate(F f, T guess, T min, T max, int digits);

template <class F, class T>
T schroeder_iterate(F f, T guess, T min, T max, int digits, boost::uintmax_t& max_iter);

}}} // namespaces
```

Description

These functions all perform iterative root finding: `newton_raphson_iterate` performs second order Newton Raphson iteration, while `halley_iterate` and `schroeder_iterate` perform third order Halley and Schroeder iteration respectively.

The functions all take the same parameters:

Parameters of the root finding functions

F f

Type F must be a callable function object that accepts one parameter and returns a tuple:

For the second order iterative methods (Newton Raphson) the tuple should have two elements containing the evaluation of the function and it's first derivative.

For the third order methods (Halley and Schroeder) the tuple should have three elements containing the evaluation of the function and its first and second derivatives.

T guess

The initial starting value.

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.

uintmax_t max_iter An optional maximum number of iterations to perform.

When using these functions you should note that:

- 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 bisection, or in some rare cases even worse than bisection 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 bisection in this case. However, it is helpful if 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.
- These functions fall back to bisection 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.
- 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:

$$x_{N+1} = x_N - \frac{2 f(x) f'(x)}{2 (f'(x))^2 - f(x) f''(x)}$$

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.

Schroeder'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.

Example

Lets suppose we want to find the cube root of a number, the equation we want to solve along with its derivatives are:

$$\begin{aligned} f(x) &= x^3 - a \\ f'(x) &= 3x^2 \\ f''(x) &= 6x \end{aligned}$$

To begin with lets solve the problem using Newton Raphson iterations, we'll begin be defining a function object that returns the evaluation of the function to solve, along with its first derivative:

```
template <class T>
struct cbrt_functor
{
   cbrt_functor(T const& target) : a(target){}
   std::tr1::tuple<T, T> operator()(T const& z)
   {
      T sqr = z * z;
      return std::tr1::make_tuple(sqr * z - a, 3 * sqr);
   }
private:
   T a;
};
```

Implementing the cube root 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:

```
template <class T>
T cbrt(T z)
{
   using namespace std;
   int exp;
   frexp(z, &exp);
   T min = ldexp(0.5, exp/3);
   T max = ldexp(2.0, exp/3);
   T guess = ldexp(1.0, exp/3);
   int digits = std::numeric_limits<T>::digits;
   return tools::newton_raphson_iterate(detail::cbrt_functor<T>(z), guess, min, max, digits);
}
```

Using the test data in libs/math/test/cbrt_test.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 its 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 error handling, and does not handle negative values of z correctly. That will be left as an exercise for the reader!

Now lets adapt the functor slightly to return the second derivative as well:

```
template <class T>
struct cbrt_functor
{
   cbrt_functor(T const& target) : a(target){}
   std::tr1::tuple<T, T, T> operator()(T const& z)
   {
      T sqr = z * z;
      return std::tr1::make_tuple(sqr * z - a, 3 * sqr, 6 * z);
   }
private:
   T a;
};
```

And then adapt the `cbrt` function to use Halley iterations:

```
template <class T>
T cbrt(T z)
{
   using namespace std;
   int exp;
   frexp(z, &exp);
   T min = ldexp(0.5, exp/3);
   T max = ldexp(2.0, exp/3);
   T guess = ldexp(1.0, exp/3);
   int digits = std::numeric_limits<T>::digits / 2;
   return tools::halley_iterate(detail::cbrt_functor<T>(z), guess, min, max, digits);
}
```

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 `schroeder_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, 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!*

Root Finding Without Derivatives

Synopsis

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


namespace boost{ namespace math{ namespace tools{

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&);

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&);
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;

}}} // namespaces
```

Description

These functions solve the root of some function *f(x)* without the need for the derivatives of *f(x)*. The functions here that use TOMS Algorithm 748 are asymptotically the most efficient known, and have been shown to be optimal for a certain classes of smooth functions.

Alternatively, there 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 the methods of Brent or Dekker) which only reduce 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, while *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.

```
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&);
```

These functions locate the root using bisection, function arguments are:

f A unary functor which is the function 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 signals evaluation error if these preconditions are violated. The action taken is controlled by the evaluation error policy. A best guess may be returned, perhaps significantly wrong.

tol A binary functor that specifies the termination condition: the function will return the current brackets enclosing the root when *tol(min,max)* becomes true.

max_iter The maximum number of invocations of *f(x)* to make while searching for the root.

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.

```
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&);
```

This is a convenience function that calls *toms748_solve* internally to find the root of *f(x)*. It's usable only when *f(x)* is a monotonic function, and the location of the root is known approximately, and in particular it is known whether the root is occurs for positive or negative *x*. The parameters are:

f

A unary functor 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*.

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 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.

max_iter

The maximum number of function invocations to perform in the search for the root.

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.

```
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 two 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 functions differ only by whether values for *f(a)* and *f(b)* are already available. The parameters are:

f A unary functor 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.

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 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.

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.

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*), rather than because the termination condition *tol* was satisfied.

```
template <class T>
struct eps_tolerance
{
   eps_tolerance(int bits);
   bool operator()(const T& a, const T& b)const;
};
```

This 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 twice the machine epsilon for T, or $2^{1\text{-bits}}$, 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 twice the machine epsilon of T is required to ensure that we get back a bracketing interval: since this must clearly be at least 1 epsilon in size.

```
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 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++.

Locating Function Minima

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 algorithm. Parameters are:

f
The function to minimise. The function 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, therefore if *bits* is set to a value greater than one half of the bits in type T, then the value will be ignored.

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 pair containing the value of the abscissa at the minima and the value of f(x) at the minima.

Implementation

This is a reasonably faithful implementation of Brent's algorithm, refer to:

Brent, R.P. 1973, Algorithms for Minimization without Derivatives (Englewood Cliffs, NJ: Prentice-Hall), Chapter 5.

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.

An algorithm with guaranteed convergence for finding a zero of a function, R. P. Brent, The Computer Journal, Vol 44, 1971.

Testing and Development

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 U>
   polynomial(const U& point);

   // access:
   size_type size()const;
   size_type degree()const;
   value_type& operator[](size_type i);
   const value_type& operator[](size_type i)const;

   // 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 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, 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 charT, class traits, class T>
std::basic_ostream<charT, traits>& operator <<
   (std::basic_ostream<charT, traits>& os, const polynomial<T>& poly);

}}} // namespaces
```

Description

This is a fairly trivial class for polynomial manipulation.

Implementation is currently of the "naive" variety, with O(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.

Advanced manipulations: the FFT, division, GCD, factorisation etc are not currently provided. Submissions for these are of course welcome :-)

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 principals 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(x + X) + Y` where *X* is the x-offset 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 val

Sets the y-offset to *val*: 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. 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>
```

```
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

```
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:

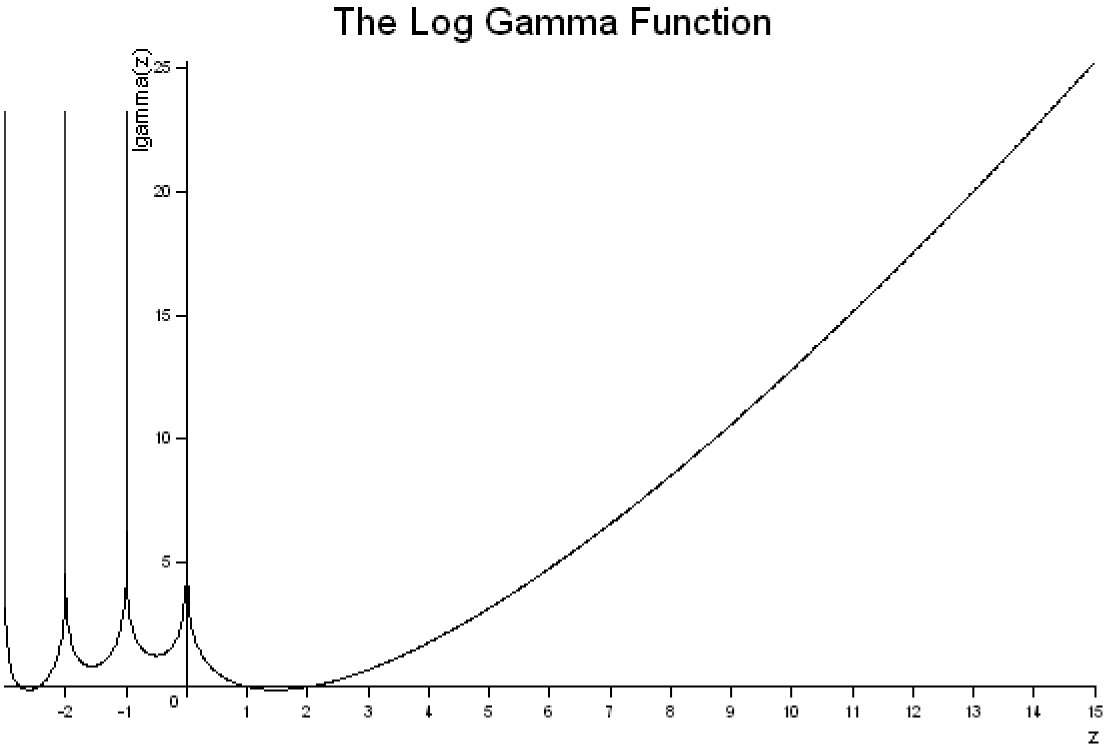
- A functor that implements the function for which data is being generated: this is the bit you have to write.
- One of more parameters that are to be passed to the functor, these are described in fairly abstract terms: give me N points distributed like *this* etc.
- The class test_data, that takes the 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 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 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/tr1/tuple.hpp>

template <class T>
struct profile_gamma_fraction
{
   typedef std::tr1::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_limits
      //
      // now just return the results as a tuple:
      return std::tr1::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

Using With NTL - a High-Precision Floating-Point Library

The special functions and tools in this library can be used with NTL::RR (an arbitrary precision number type), via the bindings in boost/math/bindings/rr.hpp. See also NTL: A Library for doing Number Theory by Victor Shoup

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.

Finally there is a high precision Lanczos approximation suitable for use with `boost::math::ntl::RR`, used at 1000-bit precision in libs/math/tools/ntl_rr_lanczos.hpp. The approximation has a theoretical precision of > 90 decimal digits, and an experimental precision of > 100 decimal digits. To use that approximation, just include that header before any of the special function headers (if you don't use it, you'll get a slower, but fully generic implementation for all of the gamma-like functions).

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 will meet these requirements. User defined types that meet the requirements can also be used. For example, with a thin wrapper class one of the types provided with NTL (RR) can be used. Submissions of binding to other extended precision types would also be most 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

Expression	Result Type	Notes
`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
`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/tools/ntl.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/tools/ntl.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 (where as for `numeric_limits` it 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

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`, `modf` and `log10` are not currently used, but may be if further special functions are added.

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 libs/math/tools/ntl_rr_lanczos.hpp: 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 of the distribution.
quantile(complement(cd, cr))	RealType	Returns the quantile 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 and Testing

There are several concept archetypes available:

```
#include <boost/concepts/std_real_concept.hpp>
```

```
namespace boost{
namespace math{
namespace concepts{

class std_real_concept;

}}} // namespaces
```

`std_real_concept` is an archetype for the built-in Real types.

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.

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 it's 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 it's standard library functions in it's 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 this is not a conceptual requirement for RealType's.

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`.

```
#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.

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

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 prefered method for installation wide policies.
- At your chosen namespace scope for distributions and/or functions: this is the prefered 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 runtime 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):

```
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> > mypolicy;
```

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.
Internal Evaluation Error	Throws a `boost::math::evaluation_error` exception.
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 be-

haviour 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> > mypolicy;
```

then `mypolicy` defines a policy where only the overflow error handling and `double`-promotion policies differ from the defaults.

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 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. 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:

```
// 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>
 using std::exception;
```

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;
      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;
         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!
errno is set to: 0
```

And finally let's build with:

```
#define BOOST_MATH_DOMAIN_ERROR_POLICY errno_on_error
```

Which gives the output:

```
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, 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>

//
// Begin by defining a policy type, that gives the
// behaviour we want:
//
using namespace boost::math::policies;
typedef policy<
   promote_float<false>,
   discrete_quantile<integer_round_nearest>
> mypolicy;
//
// Then define a distribution that uses it:
//
typedef boost::math::binomial_distribution<float, mypolicy> mybinom;
//
//  And now use it to get the quantile:
//
int main()
{
   std::cout << "quantile is: " <<
      quantile(mybinom(200, 0.25), 0.05) << std::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>
//
// Define the policy to use:
//
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;
//
// Now use the policy when calling tgamma:
//
int main()
{
   errno = 0;
   std::cout << "Result of tgamma(30000) is: "
      << boost::math::tgamma(30000, c_policy()) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   std::cout << "Result of tgamma(-10) is: "
      << boost::math::tgamma(-10, c_policy()) << std::endl;
   std::cout << "errno = " << errno << std::endl;
}
```

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>

int main()
{
   using namespace boost::math::policies;
   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, 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:

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

Open up the "C" namespace that we'll use for our functions, and define the policy type we want: in this case one that sets ::errno rather than throwing exceptions. Any policies we don't specify here will inherit the defaults:

```
namespace C{

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;
```

All we need do now is invoke the BOOST_MATH_DECLARE_SPECIAL_FUNCTIONS macro passing our policy type 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;
   std::cout << "Result of tgamma(30000) is: "
      << C::tgamma(30000) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   std::cout << "Result of tgamma(-10) is: "
      << C::tgamma(-10) << std::endl;
   std::cout << "errno = " << errno << std::endl;
}
```

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 set - possibly fragile and easy to forget - project wide build macros.

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>

namespace {

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)

} // close unnamed namespace

int main()
{
   errno = 0;
   std::cout << "Result of tgamma(30000) is: "
      << tgamma(30000) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   std::cout << "Result of tgamma(-10) is: "
      << tgamma(-10) << std::endl;
   std::cout << "errno = " << errno << std::endl;
}
```

Handling 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, mypolicy)
```

Results a set of typedefs being defined like this:

```
typedef boost::math::normal_distribution<double, mypolicy> 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:

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

Open up an appropriate namespace 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;

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 and policy types as arguments:

```
BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)

} // close namespace my_namespace
```

We now have a set of typedefs defined in namespace my_namespace 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_namespace::normal we really end up using `boost::math::normal_distribution<double, my_policy>`:

```
int main()
{
   //
   // Start with something we know will overflow:
   //
   my_distributions::normal norm(10, 2);
   errno = 0;
   std::cout << "Result of quantile(norm, 0) is: "
      << quantile(norm, 0) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   errno = 0;
   std::cout << "Result of quantile(norm, 1) is: "
      << quantile(norm, 1) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   //
   // Now try a discrete distribution:
   //
   my_distributions::binomial binom(20, 0.25);
   std::cout << "Result of quantile(binom, 0.05) is: "
      << quantile(binom, 0.05) << std::endl;
   std::cout << "Result of quantile(complement(binom, 0.05)) is: "
      << quantile(complement(binom, 0.05)) << std::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 - possibly fragile and easy to forget - project wide build macros.

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>

namespace {

using namespace boost::math::policies;

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()
{
   //
   // Start with something we know will overflow:
   //
   normal norm(10, 2);
   errno = 0;
   std::cout << "Result of quantile(norm, 0) is: "
      << quantile(norm, 0) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   errno = 0;
   std::cout << "Result of quantile(norm, 1) is: "
      << quantile(norm, 1) << std::endl;
   std::cout << "errno = " << errno << std::endl;
   //
   // Now try a discrete distribution:
   //
   binomial binom(20, 0.25);
   std::cout << "Result of quantile(binom, 0.05) is: "
```

```
      << quantile(binom, 0.05) << std::endl;
   std::cout << "Result of quantile(complement(binom, 0.05)) is: "
      << quantile(complement(binom, 0.05)) << std::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 policy{

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);

}}} // namespaces
```

So out first job is to include the header we want to use, and then provide definitions for the user-defined error handlers we want to use:

```
#include <iostream>
#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)
{
   std::cerr << "Domain Error." << std::endl;
   return std::numeric_limits<T>::quiet_NaN();
}

template <class T>
T user_pole_error(const char* function, const char* message, const T& val)
{
   std::cerr << "Pole Error." << std::endl;
   return std::numeric_limits<T>::quiet_NaN();
}


}}} // namespaces
```

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{

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 an unnamed namespace 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 `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()
{
   std::cout << "Result of erf_inv(-10) is: "
      << erf_inv(-10) << std::endl;
   std::cout << "Result of tgamma(-10) is: "
      << tgamma(-10) << std::endl;
}
```

Which outputs:

```
Domain Error.
Result of erf_inv(-10) is: 1.#QNAN
Pole Error.
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 type T.
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 <iostream>
#include <boost/math/special_functions.hpp>
```

Next we'll implement the 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:

```
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:

```
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 digits of *val*:

```
msg += ": \n";
int prec = 2 + (std::numeric_limits<T>::digits * 30103UL) / 100000UL;
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 occured 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;
   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, just return a NaN for the purposes
   // of the example:
   return std::numeric_limits<T>::quiet_NaN();
}

}}} // namespaces
```

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{
```

```
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)

} // close unnamed namespace
```

We now have a set of forwarding functions defined in an unnamed namespace 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 `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:
   std::cout << "Result of erf_inv(-10) is: "
      << erf_inv(-10) << std::endl << std::endl;
   // Raise a pole error:
   std::cout << "Result of tgamma(-10) is: "
      << tgamma(-10) << std::endl << std::endl;
   // Raise an overflow error:
   std::cout << "Result of tgamma(3000) is: "
      << tgamma(3000) << std::endl << std::endl;
   // Raise an underflow error:
   std::cout << "Result of tgamma(-190.5) is: "
      << tgamma(-190.5) << std::endl << std::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!
}
```

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
```

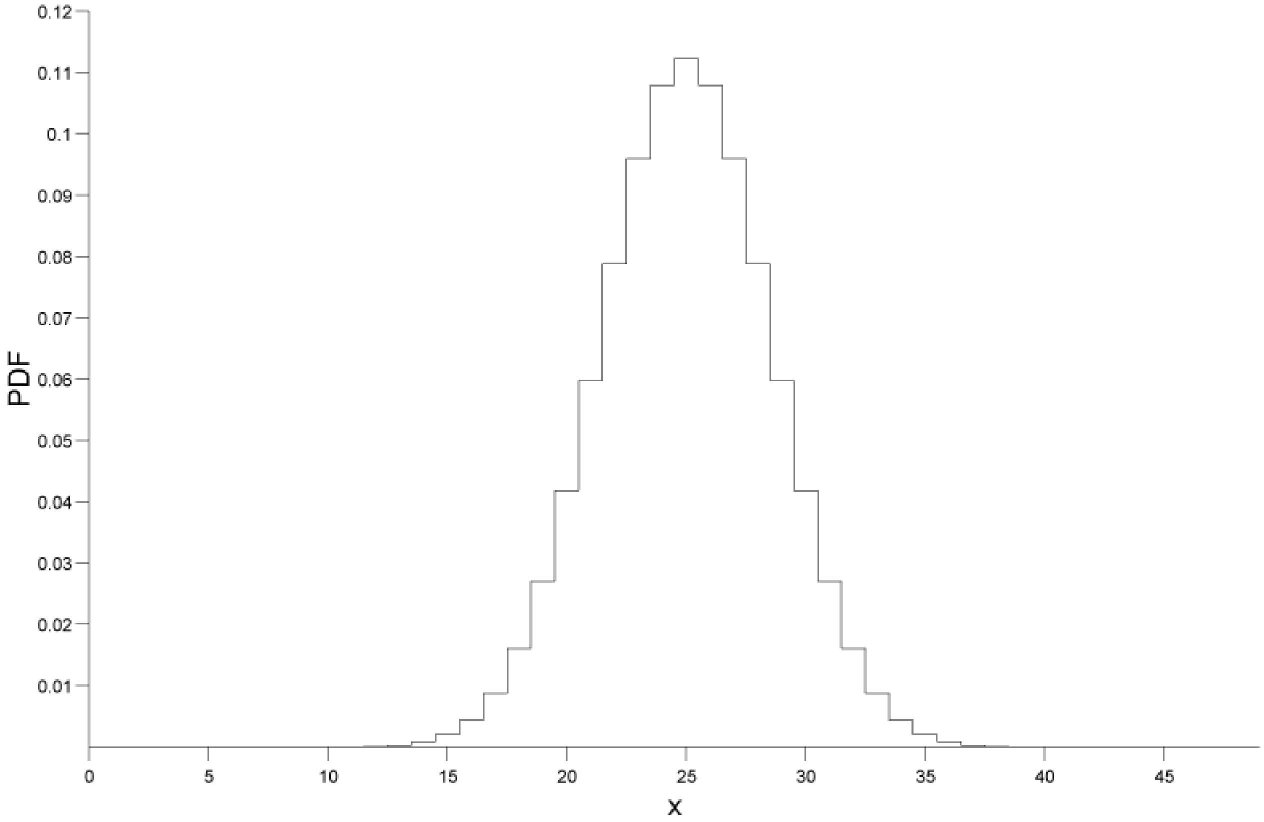
```
Error in function boost::math::tgamma<long double>(long double):
Result of tgamma is too large to represent.
Error in function boost::math::tgamma<double>(double):
Result of function is too large to represent
Result of tgamma(3000) is: 1.#INF

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's dividing by infinity and underflows.

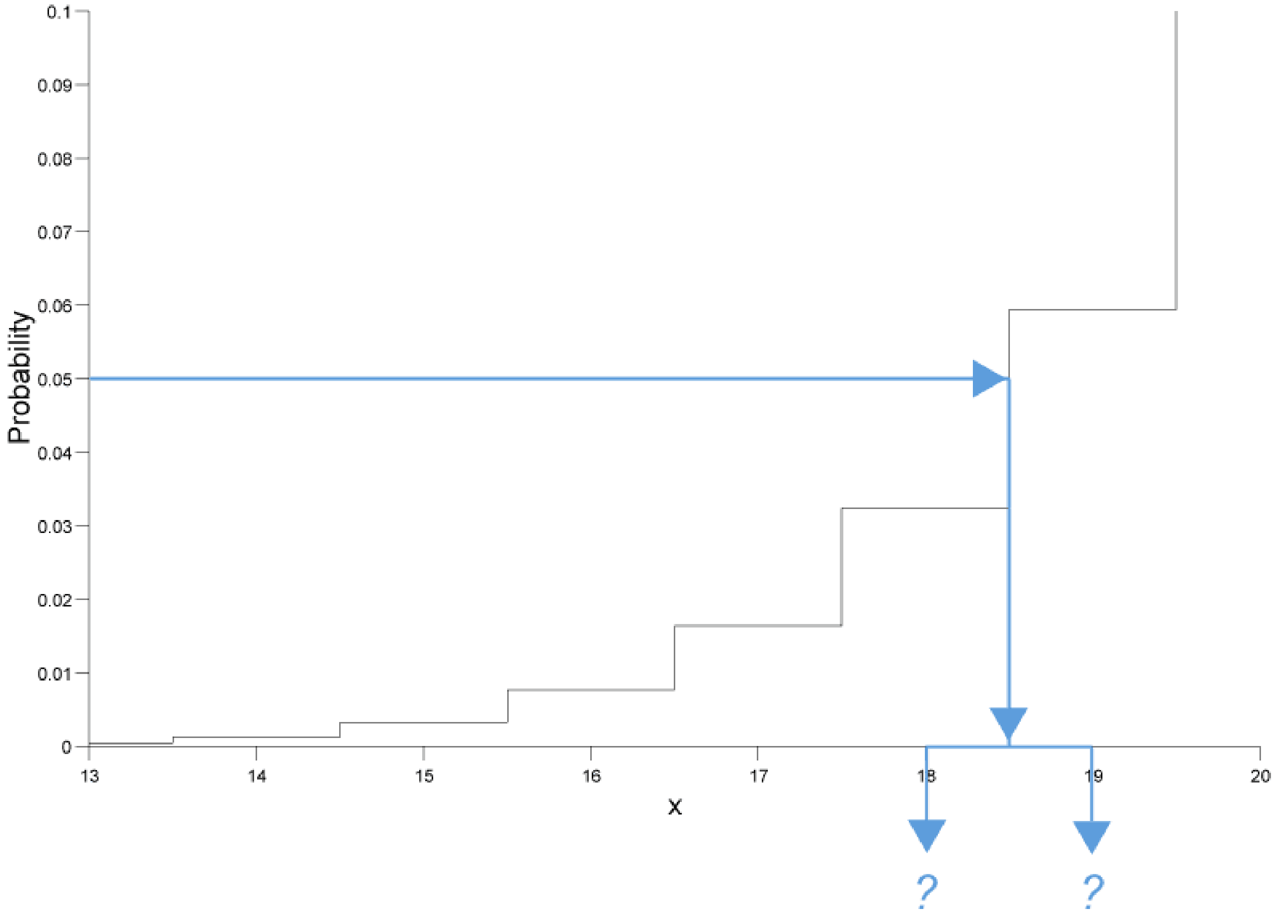
Understanding Quantiles of Discrete Distributions

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."*

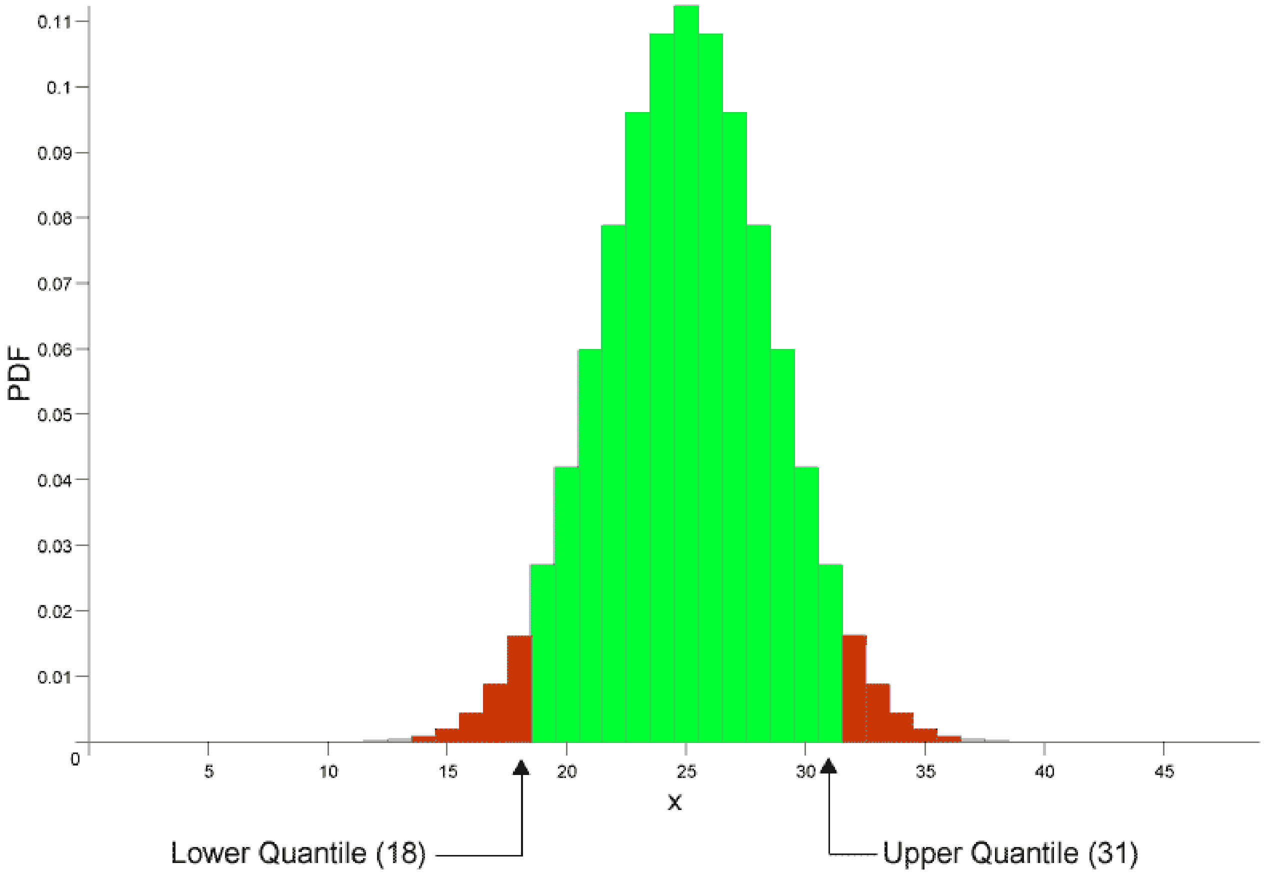
So there is a genuine benefit to calculating an integer result as well as it being "the right thing to do" from a philosophical point of view. What's more if someone asks for a quantile at 0.05, then we can normally assume that they are asking for ***at least** 95% of the probability to the right of 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:

```
#include <iostream>
#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:

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

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()
{
   std::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";

   std::cout << std::setw(25) << std::right
      << "Policy"<< std::setw(18) << std::right
      << "Lower Quantile" << std::setw(18) << std::right
      << "Upper Quantile" << std::endl;

   // Test integer_round_outwards:
   std::cout << std::setw(25) << std::right
      << "integer_round_outwards"
      << std::setw(18) << std::right
      << quantile(binom_round_outwards(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_round_outwards(50, 0.5), 0.95)
      << std::endl;

   // Test integer_round_inwards:
   std::cout << std::setw(25) << std::right
      << "integer_round_inwards"
      << std::setw(18) << std::right
      << quantile(binom_round_inwards(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_round_inwards(50, 0.5), 0.95)
      << std::endl;

   // Test integer_round_down:
   std::cout << std::setw(25) << std::right
      << "integer_round_down"
      << std::setw(18) << std::right
      << quantile(binom_round_down(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_round_down(50, 0.5), 0.95)
      << std::endl;

   // Test integer_round_up:
   std::cout << std::setw(25) << std::right
      << "integer_round_up"
      << std::setw(18) << std::right
      << quantile(binom_round_up(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_round_up(50, 0.5), 0.95)
      << std::endl;

   // Test integer_round_nearest:
   std::cout << std::setw(25) << std::right
      << "integer_round_nearest"
      << std::setw(18) << std::right
      << quantile(binom_round_nearest(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_round_nearest(50, 0.5), 0.95)
      << std::endl;

   // Test real:
   std::cout << std::setw(25) << std::right
      << "real"
      << std::setw(18) << std::right
      << quantile(binom_real_quantile(50, 0.5), 0.05)
      << std::setw(18) << std::right
      << quantile(binom_real_quantile(50, 0.5), 0.95)
      << std::endl;
}
```

Which produces the program output:

```
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.

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

Error Type	Exception
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

errno_on_error

Will set global `::errno` to one of the following values depending upon the error type, 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

ignore_error

Will return a 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 as to the result: which may be significantly in error.

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_evaluation_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 five 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`
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

Error Type	Policy Class	Description
		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`

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 namespace boost::math::policies;
using namespace boost::math;

// Define a policy:
typedef policy<
      domain_error<errno_on_error>,
      pole_error<errno_on_error>,
      overflow_error<errno_on_error>,
      policies::evaluation_error<errno_on_error>
      > my_policy;

// call the function:
double t1 = tgamma(some_value, my_policy());

// Alternatively we could use make_policy and define everything at the call site:
double t2 = tgamma(some_value, make_policy(
         domain_error<errno_on_error>(),
         pole_error<errno_on_error>(),
         overflow_error<errno_on_error>(),
         policies::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 namespace boost::math::policies;
using namespace boost::math;

// Define a policy:
typedef policy<
      overflow_error<ignore_error>
      > my_policy;

// Define the distribution:
typedef normal_distribution<double, my_policy> my_norm;

// Get a quantile:
```

```
double q = quantile(my_norm(), 0.05);
```

Internal 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 effect whether internal promotion 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 namespace boost::math;

// Define a policy:
typedef policy<
      promote_double<false>
      > my_policy;

// Call the function:
double t1 = tgamma(some_value, my_policy());

// Alternatively we could use make_policy and 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 namespace boost::math::policies;
using namespace boost::math;

// Define a policy:
typedef policy<
      promote_float<false>
      > my_policy;

// Define the distribution:
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 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> > >());
```

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 namespace boost::math;
using namespace boost::math::policies;

typedef negative_binomial_distribution<
      double,
      policy<discrete_quantile<integer_round_inwards> >
   > dist_type;

// Lower quantile:
double x = quantile(dist_type(20, 0.3), 0.05);
// Upper 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 namespace boost::math;

// Lower quantile rounded down:
double x = quantile(negative_binomial(20, 0.3), 0.05);
// Upper quantile rounded up:
double y = quantile(complement(negative_binomial(20, 0.3), 0.05));
```

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 namespace boost::math;
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);
// Upper quantile rounded down:
double y = quantile(complement(dist_type(20, 0.3), 0.05));
```

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 namespace boost::math;
using namespace boost::math::policies;

typedef negative_binomial_distribution<
      double,
      policy<discrete_quantile<integer_round_nearest> >
   > dist_type;

// Lower quantile rounded up:
double x = quantile(dist_type(20, 0.3), 0.05);
// Upper quantile rounded down:
double y = quantile(complement(dist_type(20, 0.3), 0.05));
```

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.

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

typedef policy<digits10<5> > pol;

double t = tgamma(12, pol());
```

Or again using *make_policy*:

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

using namespace boost::math;
using namespace boost::math::policies;

double t = tgamma(12, policy<digits10<5> >());
```

And for a quantile of a distribution to approximately 25-bit precision:

```
#include <boost/math/distributions/normal.hpp>

using namespace boost::math;
using namespace boost::math::policies;

double q = quantile(
      normal_distribution<double, policy<digits2<25> > >(),
      0.05);
```

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-hock 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_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_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>

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
```

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;

// Define a policy to use, in this case we want all the distribution
// accessor functions to compile, even if they are mathematically
// undefined:
typedef policy<assert_undefined<false> > my_policy;

BOOST_MATH_DECLARE_DISTRIBUTIONS(double, my_policy)

}

// Now we can use myspace::cauchy etc, which will use policy
// myspace::mypolicy:
//
// This compiles but raises a domain error at runtime:
//
void test_cauchy()
{
   try
   {
      double d = mean(myspace::cauchy());
   }
   catch(const std::domain_error& e)
   {
      std::cout << e.what() << std::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>
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 evaluation_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>
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<...>::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<...>::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>
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, you should note that the library's primary emphasis is on accuracy and numerical stability, and *not* speed.

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 FORTAN 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 two 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.

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.

Interpretting 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 seconds per function call are also shown in parenthesis.

Result were obtained on a system with an Intel 2.8GHz Pentium 4 processor with 2Gb of RAM and running either Windows XP or Mandriva 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

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.

Table 32. Performance Comparison of Release and Debug Settings

Function	**Microsoft Visual C++ 8.0** **Debug Settings: /Od /ZI**	**Microsoft Visual C++ 8.0** **Release settings: /Ox /arch:SSE2**
erf	16.65 (1.028e-006s)	**1.00** (6.173e-008s)
erf_inv	19.28 (1.215e-006s)	**1.00** (6.302e-008s)
ibeta and ibetac	8.32 (1.540e-005s)	**1.00** (1.852e-006s)
ibeta_inv and ibetac_inv	10.25 (7.492e-005s)	**1.00** (7.311e-006s)
ibeta_inva, ibetac_inva, ibeta_invb and ibetac_invb	8.57 (2.441e-004s)	**1.00** (2.847e-005s)
gamma_p and gamma_q	10.98 (1.044e-005s)	**1.00** (9.504e-007s)
gamma_p_inv and gamma_q_inv	10.25 (3.721e-005s)	**1.00** (3.631e-006s)
gamma_p_inva and gamma_q_inva	11.26 (1.124e-004s)	**1.00** (9.982e-006s)

Comparing Compilers

After a good choice of build settings the next most important thing you can do, is choose your compiler - and the standard C library it sits on top of - very carefully. GCC-3.x in particular has been found to be particularly bad at inlining code, and performing the kinds of high level transformations that good C++ performance demands (thankfully GCC-4.x is somewhat better in this respect).

Table 33. Performance Comparison of Various Windows Compilers

Function	**Intel C++ 10.0** **(/Ox /Qipo /QxN)**	**Microsoft Visual C++ 8.0** **(/Ox /arch:SSE2)**	**Cygwin G++ 3.4** **(/O3)**
erf	**1.00** (4.118e-008s)	1.50 (6.173e-008s)	3.24 (1.336e-007s)
erf_inv	**1.00** (4.439e-008s)	1.42 (6.302e-008s)	7.88 (3.500e-007s)
ibeta and ibetac	**1.00** (1.631e-006s)	1.14 (1.852e-006s)	3.05 (4.975e-006s)
ibeta_inv and ibetac_inv	**1.00** (6.133e-006s)	1.19 (7.311e-006s)	2.60 (1.597e-005s)
ibeta_inva, ibetac_inva, ibeta_invb and ibetac_invb	**1.00** (2.453e-005s)	1.16 (2.847e-005s)	2.83 (6.947e-005s)
gamma_p and gamma_q	**1.00** (6.735e-007s)	1.41 (9.504e-007s)	2.78 (1.872e-006s)
gamma_p_inv and gamma_q_inv	**1.00** (2.637e-006s)	1.38 (3.631e-006s)	3.31 (8.736e-006s)
gamma_p_inva and gamma_q_inva	**1.00** (7.716e-006s)	1.29 (9.982e-006s)	2.56 (1.974e-005s)

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: #define BOOST_MATH_INT_TABLE_TYPE(RT, IT) RT to boost/math/tools/user.hpp, otherwise the default of:

Macro	Meaning
	#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`.

To determine which of these options is best for your particular compiler/platform build the performance test application with your usual release settings, and run the program with the --tune command line option.

In practice the difference between methods is rather small at present, as the following table shows. However, parallelisation /vectorisation is likely to become more important in the future: quite likely the methods currently supported will need to be supplemented or replaced by ones more suited to highly vectorisable processors in the future.

Table 34. A Comparison of Polynomial Evaluation Methods

Compiler/platform	Method 0	Method 1	Method 2	Method 3
Microsoft C++ 8.0, Polynomial evaluation	1.34 (1.161e-007s)	1.13 (9.777e-008s)	1.07 (9.289e-008s)	**1.00** (8.678e-008s)
Microsoft C++ 8.0, Rational evaluation	**1.00** (1.443e-007s)	1.03 (1.492e-007s)	1.20 (1.736e-007s)	1.07 (1.540e-007s)
Intel C++ 10.0 (Windows), Polynomial evaluation	1.03 (7.702e-008s)	1.03 (7.702e-008s)	**1.00** (7.446e-008s)	1.03 (7.690e-008s)
Intel C++ 10.0 (Windows), Rational evaluation	**1.00** (1.245e-007s)	**1.00** (1.245e-007s)	1.18 (1.465e-007s)	1.06 (1.318e-007s)
GNU G++ 4.2 (Linux), Polynomial evaluation	1.61 (1.220e-007s)	1.68 (1.269e-007s)	1.23 (9.275e-008s)	**1.00** (7.566e-008s)
GNU G++ 4.2 (Linux), Rational evaluation	1.26 (1.660e-007s)	1.33 (1.758e-007s)	**1.00** (1.318e-007s)	1.15 (1.513e-007s)
Intel C++ 10.0 (Linux), Polynomial evaluation	1.15 (9.154e-008s)	1.15 (9.154e-008s)	**1.00** (7.934e-008s)	**1.00** (7.934e-008s)
Intel C++ 10.0 (Linux), Rational evaluation	**1.00** (1.245e-007s)	**1.00** (1.245e-007s)	1.35 (1.684e-007s)	1.04 (1.294e-007s)

There is one final performance tuning option that is available as a compile time policy. Normally when evaluating functions at `double` precision, these are actually evaluated at `long double` precision internally: this helps to ensure that as close to full `double` precision as possible is achieved, but may slow down execution in some environments. The defaults for this policy can be changed by defining the macro `BOOST_MATH_PROMOTE_DOUBLE_POLICY` to `false`, or by specifying a specific policy when calling the special functions or distributions. See also the policy tutorial.

Table 35. Performance Comparison with and Without Internal Promotion to long double

Function	GCC 4.2 , Linux (with internal promotion of double to long double).	GCC 4.2, Linux (without promotion of double).
erf	1.48 (1.387e-007s)	**1.00** (9.377e-008s)
erf_inv	1.11 (4.009e-007s)	**1.00** (3.598e-007s)
ibeta and ibetac	1.29 (5.354e-006s)	**1.00** (4.137e-006s)
ibeta_inv and ibetac_inv	1.44 (2.220e-005s)	**1.00** (1.538e-005s)
ibeta_inva, ibetac_inva, ibeta_invb and ibetac_invb	1.25 (7.009e-005s)	**1.00** (5.607e-005s)
gamma_p and gamma_q	1.26 (3.116e-006s)	**1.00** (2.464e-006s)
gamma_p_inv and gamma_q_inv	1.27 (1.178e-005s)	**1.00** (9.291e-006s)
gamma_p_inva and gamma_q_inva	1.20 (2.765e-005s)	**1.00** (2.311e-005s)

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.

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.

Comparison to GSL-1.9 and Cephes

All the results were measured on a 2.8GHz Intel Pentium 4, 2Gb RAM, Windows XP machine, with all the libraries compiled with Microsoft Visual C++ 2005 using the `/Ox /arch:SSE2` options.

Function	Boost	GSL-1.9	Cephes
tgamma	1.50 (2.566e-007s)	1.54 (2.627e-007s)	**1.00** (1.709e-007s)
lgamma	1.73 (2.688e-007s)	3.61 (5.621e-007s)	**1.00** (1.556e-007s)
gamma_p and gamma_q	**1.00**	2.15	2.57

Function	Boost	GSL-1.9	Cephes
	(9.504e-007s)	(2.042e-006s)	(2.439e-006s)
gamma_p_inv and gamma_q_inv	**1.00** (3.631e-006s)	N/A	+INF [1]
ibeta and ibetac	**1.00** (1.852e-006s)	1.07 (1.974e-006s)	1.07 (1.974e-006s)
ibeta_inv and ibetac_inv	**1.00** (7.311e-006s)	N/A	2.24 (1.637e-005s)

[1] Cephes gets stuck in an infinite loop while trying to execute our test cases.

Comparison to the R Statistical Library on Windows

All the results were measured on a 2.8GHz Intel Pentium 4, 2Gb RAM, Windows XP machine, with the test program compiled with Microsoft Visual C++ 2005, and R-2.5.0 compiled in "standalone mode" with MinGW-3.4 (R-2.5.0 appears not to be buildable with Visual C++).

Table 36. A Comparison to the R Statistical Library on Windows XP

Statistical Function	**Boost**	**R**
Beta Distribution CDF	1.20 (1.916e-006s)	**1.00** (1.597e-006s)
Beta Distribution Quantile	**1.00** (6.570e-006s)	74.66 [1] (4.905e-004s)
Binomial Distribution CDF	**1.00** (5.276e-007s)	2.45 (1.293e-006s)
Binomial Distribution Quantile	**1.00** (4.013e-006s)	1.32 (5.280e-006s)
Cauchy Distribution CDF	**1.00** (1.231e-007s)	1.28 (1.576e-007s)
Cauchy Distribution Quantile	**1.00** (1.498e-007s)	**1.00** (1.498e-007s)
Chi Squared Distribution CDF	**1.00** (7.889e-007s)	2.48 (1.955e-006s)
Chi Squared Distribution Quantile	**1.00** (4.303e-006s)	1.61 (6.925e-006s)
Exponential Distribution CDF	**1.00** (1.955e-007s)	1.97 (3.844e-007s)
Exponential Distribution Quantile	1.07 (1.206e-007s)	**1.00** (1.126e-007s)
Fisher F Distribution CDF	**1.00** (1.309e-006s)	2.12 (2.780e-006s)
Fisher F Distribution Quantile	**1.00** (7.204e-006s)	1.78 (1.280e-005s)
Gamma Distribution CDF	**1.00** (1.076e-006s)	2.07 (2.227e-006s)
Gamma Distribution Quantile	**1.00** (5.189e-006s)	1.14 (5.937e-006s)
Log-normal Distribution CDF	**1.00** (2.078e-007s)	1.41 (2.930e-007s)
Log-normal Distribution Quantile	**1.00** (6.692e-007s)	1.63 (1.090e-006s)
Negative Binomial Distribution CDF	**1.00** (9.005e-007s)	2.42 (2.178e-006s)

Statistical Function	**Boost**	**R**
Negative Binomial Distribution Quantile	**1.00** (9.601e-006s)	53.59 [2] (5.145e-004s)
Normal Distribution CDF	**1.00** (5.926e-008s)	3.01 (1.785e-007s)
Normal Distribution Quantile	**1.00** (1.248e-007s)	1.05 (1.311e-007s)
Poisson Distribution CDF	**1.00** (8.999e-007s)	2.42 (2.175e-006s)
Poisson Distribution	**1.00** (1.853e-006s)	2.17 (4.014e-006s)
Students t Distribution CDF	**1.00** (1.223e-006s)	1.13 (1.376e-006s)
Students t Distribution Quantile	**1.00** (2.570e-006s)	1.04 (2.668e-006s)
Weibull Distribution CDF	**1.00** (4.741e-007s)	1.46 (6.943e-007s)
Weibull Distribution Quantile	**1.00** (7.926e-007s)	1.08 (8.542e-007s)

[1] There are a small number of our test cases where the R library fails to converge on a result: these tend to dominate the performance result.

[2] The R library appears to use a linear-search strategy, that can perform very badly in a small number of pathological cases, but may or may not be more efficient in "typical" cases

Comparison to the R Statistical Library on Linux

All the results were measured on a 2.8GHz Intel Pentium 4, 2Gb RAM, Mandriva Linux machine, with the test program and R-2.5.0 compiled with GNU G++ 4.2.0.

Table 37. A Comparison to the R Statistical Library on Linux

Statistical Function	**Boost**	**R**
Beta Distribution CDF	1.71 (3.508e-006s)	**1.00** (2.050e-006s)
Beta Distribution Quantile	**1.00** (1.294e-005s)	44.06 [1] (5.701e-004s)
Binomial Distribution CDF	1.22 (1.342e-006s)	**1.00** (1.104e-006s)
Binomial Distribution Quantile	1.36 (7.083e-006s)	**1.00** (5.194e-006s)
Cauchy Distribution CDF	**1.00** (1.372e-007s)	1.47 (2.017e-007s)
Cauchy Distribution Quantile	**1.00** (1.542e-007s)	1.14 (1.752e-007s)
Chi Squared Distribution CDF	1.04 (1.820e-006s)	**1.00** (1.753e-006s)
Chi Squared Distribution Quantile	1.39 (9.345e-006s)	**1.00** (6.728e-006s)
Exponential Distribution CDF	**1.00** (2.195e-007s)	1.17 (2.561e-007s)
Exponential Distribution Quantile	**1.00** (1.123e-007s)	1.03 (1.155e-007s)
Fisher F Distribution CDF	**1.00** (2.744e-006s)	1.08 (2.970e-006s)
Fisher F Distribution Quantile	1.14 (1.550e-005s)	**1.00** (1.359e-005s)
Gamma Distribution CDF	1.29 (2.578e-006s)	**1.00** (1.992e-006s)
Gamma Distribution Quantile	1.77 (1.020e-005s)	**1.00** (5.757e-006s)
Log-normal Distribution CDF	**1.00** (1.782e-007s)	2.00 (3.564e-007s)
Log-normal Distribution Quantile	**1.00** (7.093e-007s)	1.07 (7.607e-007s)
Negative Binomial Distribution CDF	1.03 (2.209e-006s)	**1.00** (2.141e-006s)

Statistical Function	**Boost**	**R**
Negative Binomial Distribution Quantile	**1.00** (1.826e-005s)	30.07 [2] (5.490e-004s)
Normal Distribution CDF	**1.00** (8.542e-008s)	2.09 (1.782e-007s)
Normal Distribution Quantile	**1.00** (1.362e-007s)	1.26 (1.722e-007s)
Poisson Distribution CDF	1.10 (1.953e-006s)	**1.00** (1.775e-006s)
Poisson Distribution	1.12 (4.214e-006s)	**1.00** (3.752e-006s)
Students t Distribution CDF	1.55 (2.441e-006s)	**1.00** (1.576e-006s)
Students t Distribution Quantile	1.33 (3.972e-006s)	**1.00** (2.990e-006s)
Weibull Distribution CDF	**1.00** (6.640e-007s)	1.06 (7.031e-007s)
Weibull Distribution Quantile	**1.00** (7.504e-007s)	1.03 (7.710e-007s)

[1] There are a small number of our test cases where the R library fails to converge on a result: these tend to dominate the performance result.

[2] The R library appears to use a linear-search strategy, that can perform very badly in a small number of pathological cases, but may or may not be more efficient in "typical" cases

The Performance Test Application

Under *boost-path*/libs/math/performance you will find a (fairly rudimentary) performance test application for this library.

To run this application yourself, build the all the .cpp files in *boost-path*/libs/math/performance into an application using your usual release-build settings. Run the application with --help to see a full list of options, or with --all to test everything (which takes quite a while), or with --tune to test the available performance tuning options.

If you want to use this application to test the effect of changing any of the Policies, then you will need to build and run it twice: once with the default Policies, and then a second time with the Policies you want to test set as the default.

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.

Implemention 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 . _math_undefined 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 T is a User-defined type), 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 median approximation for distribution `dist` may be available from

`quantile(dist, 0.5)`.

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.

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 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, for more complex 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 uncomputable 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 coeffiecents 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 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 overlo
double p = boost::math::constants::pi(); // Context does not allow for disambiguation of over
```

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.

Other than that, the code is intended to be thread safe **for built in real-number types** : so float, double and long double are all thread safe.

For non-built-in types - NTL::RR for example - initialisation of the various constants used in the implementation is potentially **not** thread safe. This most undesiable, but it would be a signficant challenge to fix it. Some compilers may offer the option of having static-constants initialised in a thread safe manner (Commeau, and maybe others?), if that's the case then the problem is solved. This is a topic of hot debate for the next C++ std revision, so hopefully all compilers will be required to 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.

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.

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 *the table makers 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:

$$a_k \quad = \quad (-1)^k\sqrt{\frac{2}{\pi}}\,e^g\,k\sum_{j=0}^{k}(-1)^j\frac{(k+j-1)!}{(k-j)!\,j!}\left(\frac{e}{j+g+\frac{1}{2}}\right)^{j+\frac{1}{2}}$$

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 = B D 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:

$$D_{ij} = \begin{cases} 0 & if \quad i \neq j \\ 1 & if \quad i = j = 0 \\ -1 & if \quad i = j = 1 \\ \dfrac{D_{i-1,i-1}\, 2(2i-1)}{i-1} & otherwise \end{cases}$$

C is an NxN matrix:

$$C_{i,j} = \begin{cases} \dfrac{1}{2} & if \quad i = j = 0 \\ 0 & if \quad j > i \\ -1^{i-j} S & otherwise \end{cases} \quad ; \quad S = \sum_{k=0}^{i} \binom{2i}{2k} \binom{k}{k+j-i}$$

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 38. 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 39. 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	1.4284562135094165802001953125	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].

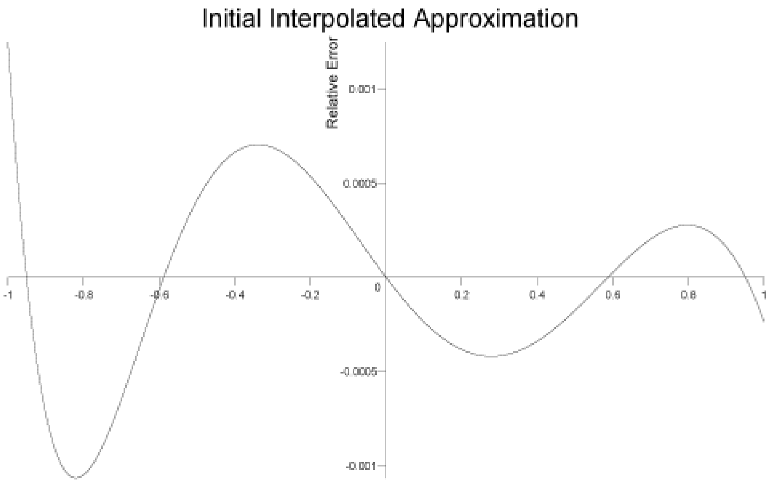
Before we can begin the Remez method, we must obtain an initial value for the location of the extrema of the error function. We could "guess" these, but a much closer first approximation can be obtained by first constructing an interpolated polynomial approximation to f(x).

In order to obtain the N+1 coefficients of the interpolated polynomial we need N+1 points ($x_0 ... x_N$): with our interpolated form passing through each of those points that yields N+1 simultaneous equations:

$f(x_i) = P(x_i) = c_0 + c_1 x_i ... + c_N x_i^N$

Which can be solved for the coefficients $c_0 ... c_N$ 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.

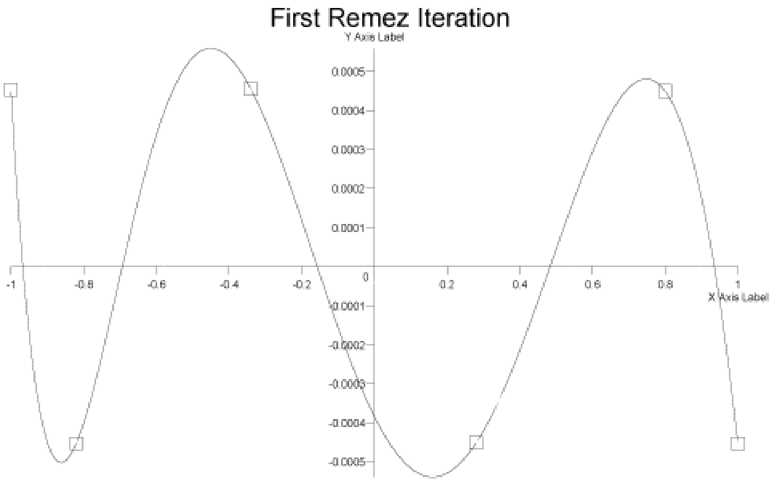
Remez Step 1

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.

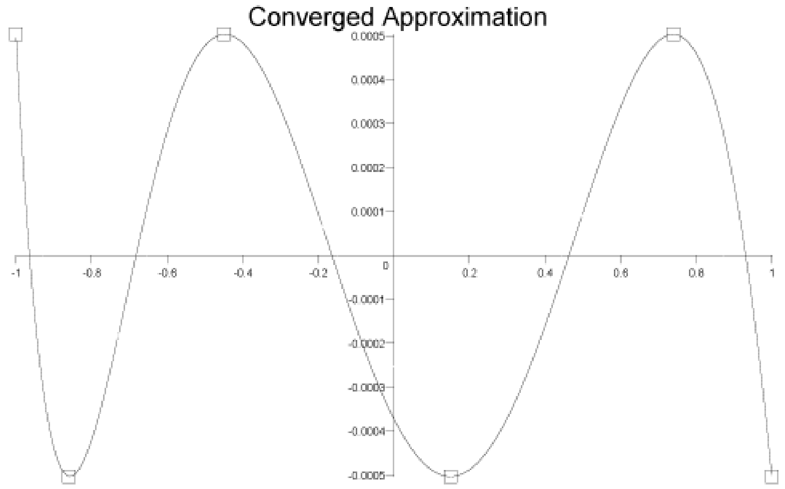
In multi-point exchange we swap all the current control points, for the locations of the extrema.

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 $5x10^{-4}$ 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 it's 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*.

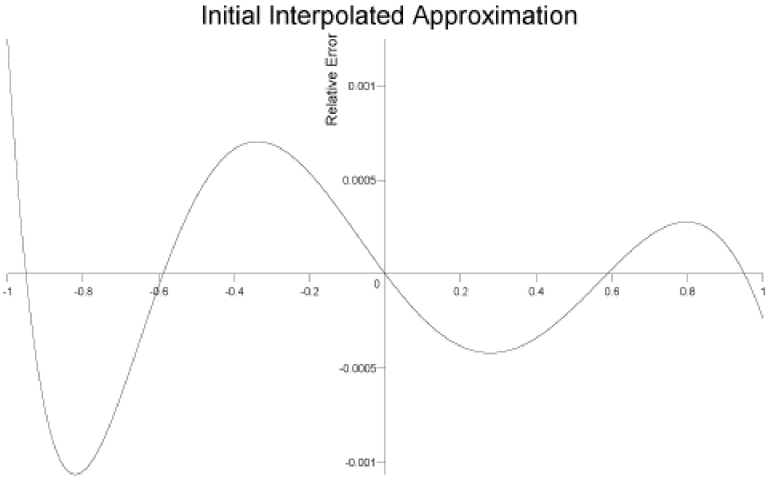
The difficult part is obviously finding the right g(x) to extract from your function: often the asymptotic behaviour of the function will give a clue, so for example the function erfc becomes proportional to e^{-x^2}/x as x becomes large. Therefore using:

erfc(z) = (C + R(x)) e^{-x^2}/x

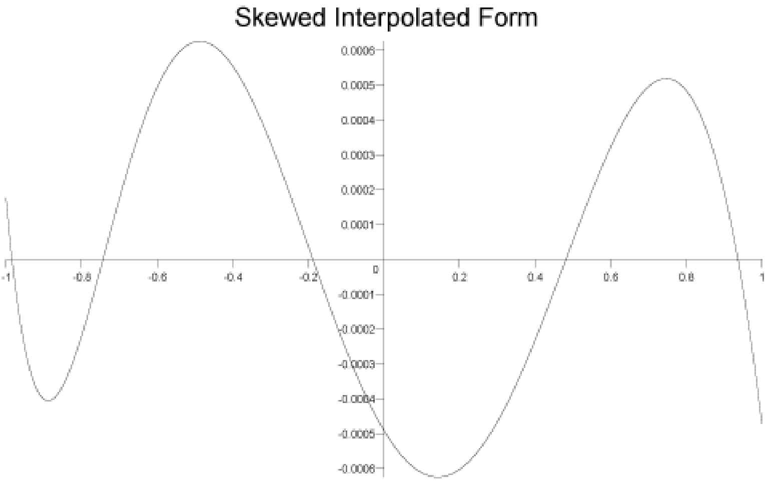
as the approximating form seems like an obvious thing to try, and does indeed yield a useful approximation.

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 intended to be a replacement for the legendary Abramowitz and Stegun's Handbook of Mathematical Functions, now scheduled to be completed in 2007.

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..

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.

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.

pugh.pdf (application/pdf Object) Pugh Msc Thesis on the Lanczzos 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* that we found (and used to cross-check - as far as their widely-varying accuracy allowed).

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.

100-decimal digit calculator provided some spot values.

C++ version.

CDFLIB Library of Fortran Routines for Cumulative Distribution functions.

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.

Status and Roadmap

History and Roadmap

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.

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 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 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 5: Post Review First Official Release (in Progress)

The documentation for this first release is still being worked upon.

- 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.

Future

- Some TR1 Special functions are still needed.
- Higher level statistical tests, perhaps integrated with Eric Neiblers accumulator library.

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.

The code has been compiled and tested with:

Table 40. Compiler Notes

Compiler	Platform	Notes
Intel 9.1, 8.1	Win32 and Linux	The tests cases tend to generate a lot of warnings relating to numeric underflow of the test data: these are harmless. The headers all appear to be warning free.
g++	Linux and HP-UX	The test suite doesn't compile with -pedantic (a problem with system headers included by Boost.Test not compiling with that option), otherwise our headers should be warning free.
HP aCC	HP-UX	Unfortunately this emits quite a few warnings from libraries upon which we depend (TR1, Array etc).
Borland 5.8.2	Windows	Almost works: some effort has been put into porting to this compiler. However, during testing a number of instances were encountered where this compiler generated incorrect code: completely omitting a function call seemingly at random. For this reason, we cannot recoment using this library with this compiler, as the correct operation of the code cannot be guarenteed.
MSVC 8.0	Windows	Warning free at level 4

Known Issues, and Todo List

This section lists those issues that are known about.

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

- Carlson's algorithms are essentially unchanged from Xiaogang Zhang's Google Summer of Code student project, and are based on Carlson's original papers. However, Carlson has revised his algorithms since then (refer to the references in the elliptic integral docs for a list), to improve performance and accuracy, we may be able to take advantage of these improvements too (Low Priority).
- Carlson's algorithms (mainly R_J) are somewhat prone to internal overflow/underflow when the arguments are very large or small. The homogeneity relations:

  $R_F(ka, kb, kc) = k^{-1/2} R_F(a, b, c)$

  and

  $R_J(ka, kb, kc, kr) = k^{-3/2} R_J(a, b, c, r)$

  could be used to sidestep trouble here: provided the problem domains can be accurately identified. (Medium Priority).
- Carlson's R_C can be reduced to elementary funtions (asin and log), would it be more efficient evaluated this way, rather than by Carlson's algorithms? (Low Priority).
- Should we add an implementation of Carlson's R_G? It's not required for the Legendre form integrals, but some people may find it useful (Low Priority).
- There are a several other integrals: $D(\phi, k)$, $Z(\beta, k)$, $\Lambda_0(\beta, k)$ and 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).
- There is a sub-domain of ellint_3 that is unimplemented (see the docs for details), currently it's not clear how to solve this issue, or if it's ever likely to be an real problem in practice - especially as most other implementations don't support this domain either (Medium Priority).

Inverse Hyperbolic Functions

- These functions are inherited from previous Boost versions, before log1p became widely available. Would they be better expressed in terms of this function? This is probably only an issue for very high precision types (Low Priority).

Statistical distributions

- Student's t Perhaps switch to normal distribution as a better approximation for very large degrees of freedom?

Credits and Acknowledgements

Hubert Holin started the Boost.Math library. The inverse hyperbolic functions, and the sinus cardinal functions are his.

John Maddock started this library, 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.

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.