PDF Test

John Maddock

Paul A. Bristow

Table of Contents

Introduction

This document is purely a test case to test out PDF generation and style.

This is some body text.

```
int main()
{
 double d = 2.345;
 return d;
}
```

We can count in Greek too: α, β, γ.

Try some superscrips and subscripts: x^2, x_i^3, α^2, β^α, x , α , a .

test HTML4 symbols

ƒ, A, B, Γ, Δ, E, Z, H, Θ, I, K, Λ, M, N, Ξ, O, Π, P, Σ, T, Y, Φ, X, Ψ, Ω, α, β, γ, δ, ε, ζ, η, θ, ι, κ, λ, μ, ν, ξ, ο, π, ρ, ς, σ, τ, υ, φ, χ, ψ, ω, ϑ, ϒ, ϖ, •, …, ′, ″, ‾, ⁄, ℘, ℑ, ℜ, ™, ℵ, ←, ↑, →, ↓, ↔, ↵, ⇐, ⇑, ⇒, ⇓, ⇔, ∀, ∂, ∃, ∅, ∇, ∈, ∉, ∋, ∏, ∑, −, ∗, √, ∝, ∞, ∠, ∧, ∨, ∩, ∪, ∫, ∴, ∼, ≅, ≈, ≠, ≡, ≤, ≥, ⊂, ⊃, ⊄, ⊆, ⊇, ⊕, ⊗, ⊥, ⋅, ◊, ♠, ♣, ♥, ♦,

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

Distributions are Objects

Each kind of distribution in this library is a class type: this 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 name: beta, gamma, binomial, in which case using the default 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 binomial distribution
// with probability of success 0.3
// and 20 trials in total:
binomial_distribution<> d2(20, 0.3); // Note: _distribution<> !
```

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.

The distributions in the random number library are typically much faster, but much less accurate than those in this library. This reflects the differing uses of the two libraries: if you want to conduct statistical tests, then you will need the reliable computation of this library, where as random numbers are *random*: so accuracy is not a concern, where as speed most certainly is.

Tip

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.

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)*.

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 (effectly 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. Mathematically, the random variate may take an (+ or -) infinite value, but this implementation limits it 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`.

Tip

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.

Likewise the `quantile` function will return a real value: users should use either the `floor` or `ceil` functions to convert the result to the nearest integer. However, whether it makes more sense to take the floor or the ceiling of the result, depends both on the distribution, and the particular use case. So some common sense needs to be applied as well!

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.

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 estimated

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 estimators 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::estimate_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 estimated from other information.

Now that you have the basics, the next section looks at some worked examples.

Worked Examples

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.]

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 accepted or rejected at a given significance level:

```
// 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	Complement of CDF for |t| > significance level / 2: `cdf(complement(dist, fabs(t))) > alpha / 2`
The Alternative-hypothesis: there is **difference** in means	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.	CDF of t < significance level: `cdf(dist, t) < alpha`
The Alternative-hypothesis: the sample mean is **greater** than the true mean.	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              ACCEPTED
Mean  < 5.000              REJECTED
Mean  > 5.000              ACCEPTED
```

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 accepted.

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            =  9.343e-002

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            =  9.343e-002

Results for Alternative Hypothesis and alpha            =  0.1000

Alternative Hypothesis     Conclusion
Mean != 38.900            REJECTED
Mean  < 38.900            ACCEPTED
Mean  > 38.900            REJECTED
```

In this case we really have a borderline result, and more data should be collected.

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_estimate_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 `estimate_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 `estimate_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::estimate_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::estimate_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:

$$t = \frac{Sm_1 - Sm_2}{s_p \sqrt{\frac{1}{Sn_1} + \frac{1}{Sn_2}}}$$

$$s_p = \sqrt{\frac{(Sn_1 - 1)Sd_1^2 + (Sn_2 - 1)Sd_2^2}{Sn_1 + Sn_2 - 2}}$$

$$\nu = Sn_1 + Sn_2 - 2$$

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 << 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	C + + Code
The Null-hypothesis: there is **no difference** in means	Complement of CDF for |t| > significance level / 2: `cdf(complement(dist, fabs(t))) > alpha / 2`	
The Alternative-hypothesis: there is a **difference** in means	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.	CDF of t < significance level: `cdf(dist, t) < alpha`	
The Alternative-hypothesis: Sample 1 Mean is **greater** than Sample 2 Mean.	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         =  2.637e-030

Results for Alternative Hypothesis and alpha         =  0.0500

Alternative Hypothesis              Conclusion
Sample 1 Mean != Sample 2 Mean       ACCEPTED
Sample 1 Mean <  Sample 2 Mean       ACCEPTED
Sample 1 Mean >  Sample 2 Mean       REJECTED
```

So with a probability that the difference is due to chance of just 2.637e-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           =  7.855e-026

Results for Alternative Hypothesis and alpha           =  0.0500

Alternative Hypothesis              Conclusion
Sample 1 Mean != Sample 2 Mean       ACCEPTED
Sample 1 Mean <  Sample 2 Mean       ACCEPTED
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 (7.855e-026 compared to 2.637e-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:

$$\sqrt{\frac{(N-1)s^2}{\chi^2_{\left(\frac{\alpha}{2},N-1\right)}}} \leq \sigma \leq \sqrt{\frac{(N-1)s^2}{\chi^2_{\left(1-\frac{\alpha}{2},N-1\right)}}}$$

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

Where $\chi^2_{(alpha;\ N-1)}$ is the upper critical value of the Chi Squared distribution, and $\chi^2_{(1-alpha;\ N-1)}$ is the lower critical value.

Note

The table above is framed in terms of acceptance of hypothesis. However, good experimental design dictates that you should formulate a hypothesis and then try and reject it. You should therefore decide which of the above situations corresponds to your null-hypothesis, and reject it if the test given above *fails*.

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 `estimate_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 `estimate_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::estimate_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::estimate_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)	$F_{(1-alpha/2;\ N1-1,\ N2-1)} < F < F_{(alpha/2;\ N1-1,\ N2-1)}$
The alternative hypothesis: there is a difference in means (two sided test)	$F < F_{(1-alpha/2;\ N1-1,\ N2-1)}$ or $F > F_{(alpha/2;\ N1-1,\ N2-1)}$
The alternative hypothesis: Standard deviation of sample 1 is greater than that of sample 2	$F > F_{(alpha;\ N1-1,\ N2-1)}$
The alternative hypothesis: Standard deviation of sample 1 is less than that of sample 2	$F < F_{(1-alpha;\ N1-1,\ N2-1)}$

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.

Note

The above tests are formulated in terms of *acceptance* because it's easier to understand the concepts involved. However, good experimental design suggests that hypothesis should be *rejected* not *accepted*. Therefore you should decide which of the above situations represents your null-hypothesis, and then reject the hypothesis only if the test above *fails*.

The upper and lower critical values can be computed using the quantile function:

$F_{(1\text{-alpha; N1-1, N2-1})}$ = `quantile(fisher_f(N1-1, N2-1), alpha)`

$F_{(\text{alpha; N1-1, N2-1})}$ = `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.)

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<>::estimate_lower_bound_on_p` and `binomial_distribution<>::estimate_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) / trials << "
```

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 `estimate_lower_bound_on_p` and `estimate_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 $\alpha$is not the same thing as calculating a two sided interval. Had we calculate two single-sided intervals each with a risk that the true value is outside the interval of α, then:

- The risk that it is less than the lower bound is α.

and

- The risk that it is greater than the upper bound is also α.

So the risk it is outside **upper or lower bound**, is **twice** alpha, and the probability that it is inside the bounds is therefore not nearly as high as one might have thought. This is why α/2 must be used in the calculations below.

In contrast, had we been calculating a single-sided interval, for example: *"Calculate a lower bound so that we are P% sure that the true occurrence frequency is greater than some value"* then we would **not** have divided by two.

Finally note that `binomial_distribution` provides a choice of two methods for the calculation, we print out the results from both methods in this example:

```
   for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
   {
      // Confidence value:
      cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]);
      // Calculate Clopper Pearson bounds:
      double l = binomial_distribution<>::estimate_lower_bound_on_p(
                     trials, successes, alpha[i]/2);
      double u = binomial_distribution<>::estimate_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<>::estimate_lower_bound_on_p(
            trials, successes, alpha[i]/2,
            binomial_distribution<>::jeffreys_prior_interval);
      u = binomial_distribution<>::estimate_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<>::estimate_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 estimate_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::estimate_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 `estimate_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". If, by experiment, you want to measure the frequency with which successes occur, 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<>::estimate_lower_bound_on_p` and `negative_binomial_distribution<>::estimate_upper_bound_on_p` allow you to calculate the confidence intervals for your estimate of the occurrence frequency.

The sample program neg_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/negative_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 negative 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) / trials << "
```

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          Upper\n"
        " Value (%)        Limit          Limit\n"
        "_____________________________________________\n";
```

And now for the important part - the intervals themselves - for each value of *alpha*, we call `estimate_lower_bound_on_p` and `estimate_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.

```
   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 l = binomial::estimate_lower_bound_on_p(trials, successes, alpha[i]/2);
      double u = binomial::estimate_upper_bound_on_p(trials, successes, alpha[i]/2);
      // Print Limits:
      cout << fixed << setprecision(5) << setw(15) << right << l;
      cout << fixed << setprecision(5) << setw(15) << right << u << endl;
   }
   cout << endl;
}
```

And that's all there is to it. Let's see some sample output for a 1 in 10 success ratio, first for 20 trials:

```
_____________________________________________
2-Sided Confidence Limits For Success Ratio
_____________________________________________

Number of Observations                 =  20
Number of successes                    =  2
Sample frequency of occurrence         =  0.1


_____________________________________________
Confidence        Lower          Upper
 Value (%)        Limit          Limit
_____________________________________________
    50.000        0.08701        0.18675
    75.000        0.06229        0.23163
    90.000        0.04217        0.28262
    95.000        0.03207        0.31698
    99.000        0.01764        0.38713
    99.900        0.00786        0.47093
    99.990        0.00358        0.54084
    99.999        0.00165        0.60020
```

As you can see, even at the 95% confidence level the bounds are really quite wide. Compare that with the program output for 2000 trials:

```
_____________________________________________
2-Sided Confidence Limits For Success Ratio
_____________________________________________

Number of Observations                 =  2000
Number of successes                    =  200
Sample frequency of occurrence         =  0.1000000


_____________________________________________
Confidence        Lower          Upper
 Value (%)        Limit          Limit
_____________________________________________
    50.000        0.09585        0.10491
    75.000        0.09277        0.10822
    90.000        0.08963        0.11172
    95.000        0.08767        0.11399
    99.000        0.08390        0.11850
    99.900        0.07966        0.12385
    99.990        0.07621        0.12845
    99.999        0.07325        0.13256
```

Now even when the confidence level is very high, the limits are really quite close to the experimentally calculated value of 0.1.

Estimating Sample Sizes for the Negative Binomial.

Imagine you have an event (let's call it a "failure") that you know will occur in 1 in N trials. You may want to know how many trials you need to conduct to be 100P% sure of observing at least k such failures. If the failure events follow a negative binomial distribution (each trial either succeeds or does not) then the static member function `negative_binomial_distibution<>::estimate_minimum_number_of_trials` can be used to estimate the minimum number of trials required to be 100P% 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 estimate_max_sample_size(
   double p,              // success fraction.
   unsigned failures)    // Total number of observed failures required.
{
```

The routine then declares a table of probability thresholds: 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 };
```

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:

```
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 table rows.
for(unsigned i = 0; i < sizeof(alpha)/sizeof(alpha[0]); ++i)
{
   // Confidence values %:
   cout << fixed << setprecision(3) << setw(10) << right << 100 * (1-alpha[i]) << "      "
      // estimate_minimum_number_of_trials
      << setw(6) << right
      << ceil(negative_binomial::estimate_minimum_number_of_trials(
                  failures, p, alpha[i])) << endl;
}
cout << endl;
```

Note that 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 `estimate_maximum_number_of_trials` like this:

```
floor(negative_binomial::estimate_maximum_number_of_trials(
        failures, p, alpha[i]))
```

which would give us the largest number of trials we could conduct and still be 100P% 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 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.

Error Handling Example

See error handling for a detailed explanation of the mechanism of handling errors, including the common "bad" arguments to distributions and functions.

But, by default, **none of these exceptions will be raised**, and the most appropriate value, usually a NaN, or zero, will be returned. So, for example, without any attempt to deal with errors, if we write

```
cout << "Probability of Student's t is "
  << boost::math::cdf(students_t(-1), -1) << endl;
```

the output will be:

```
Probability of Student#s t is 1.#QNAN
```

However, macros control whether an exception is raised, or not, for the following errors:

```
BOOST_MATH_THROW_ON_DOMAIN_ERROR
BOOST_MATH_THROW_ON_OVERFLOW_ERROR
BOOST_MATH_THROW_ON_OVERFLOW_ERROR
BOOST_MATH_THROW_ON_UNDERFLOW_ERROR
BOOST_MATH_THROW_ON_DENORM_ERROR
BOOST_MATH_THROW_ON_LOGIC_ERROR
```

Important

These #defines **MUST** be **BEFORE** the function or distribution #include!

If defined, for example with

```
#define BOOST_MATH_THROW_ON_DOMAIN_ERROR
```

before

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

then a domain_error exception will throw if we write

```
try
{
   cout << "Probability of Student's t is "
    << boost::math::cdf(students_t(-1), -1) << endl; // Bad arguments!
}
catch(const std::exception& e)
{
  std::cout <<
    "\n""Message from thrown exception was:\n   " << e.what() << std::endl;
}
```

and will output:

```
Message from thrown exception was:
Error in function __thiscall
boost::math::students_t_distribution<double>::students_t_distribution(double):
Degrees of freedom argument is -1, but must be > 0 !
```

See 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, the is not done for most examples.

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.

Beta Distribution

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

```
namespace boost{ namespace math{

template <class RealType = double>
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 beta_distribution
{
public:
   // 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 estimate_alpha(
     RealType mean, // Expected value of mean.
     RealType variance); // Expected value of variance.

   static RealType estimate_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 estimate_alpha(
     RealType beta, // from beta.
     RealType x, //  x.
     RealType probability); // cdf

   static RealType estimate_beta(
     RealType alpha, // alpha.
     RealType x, // probability x.
     RealType probability); // probability cdf.
};

}} // namespaces
```

The class type `beta_distribution` represents a beta probability distribution function.

The beta distribution is used as a prior distribution for binomial proportions in Bayesian analysis.

See also: beta distribution and Bayesian statistics.

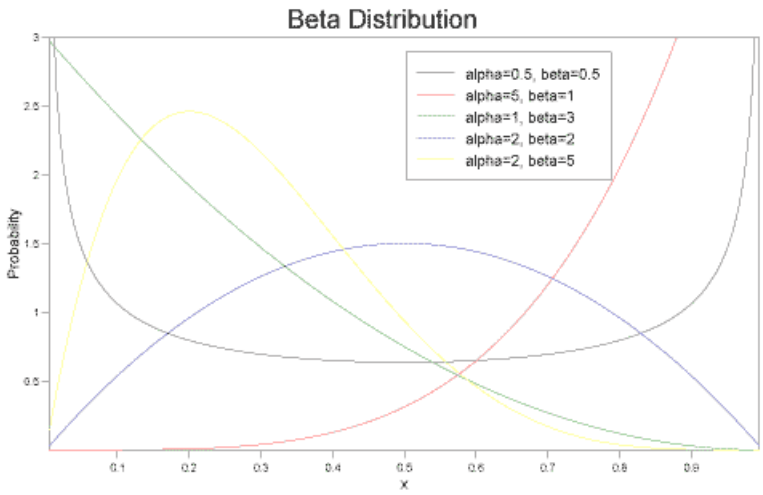
How the beta distribution is used for Bayesian analysis of one parameter models is discussed by Jeff Grynaviski.

The probability density function PDF for the beta distribution defined on the interval [0,1] is given by:

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

where B(α, β) is the beta function, implemented in this library as beta. Division by the beta function ensures that the pdf is normalized to the range zero to unity.

The following graph illustrates examples of the pdf for various values of the shape parameters. Note the α = β = 2 (blue line) is dome-shaped, and might be approximated by a symmetrical triangular distribution.



If α = β = 1, then it is a 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 estimate_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 estimate_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 estimate_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 estimate_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)

Function	Implementation Notes
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 `estimate_alpha` and `estimate_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.
`estimate_alpha`	`ibeta_inva(beta, x, probability)`
`estimate_beta`	`ibeta_invb(alpha, x, probability)`

References

Wikipedia Beta distribution

NIST Exploratory Data Analysis

Wolfram MathWorld

Digamma

Caution

This is not an official Boost library, it is a library under construction, the code is fully functional and robust, but interfaces, library structure, and function and distribution names may be changed without notice.

Synopsis

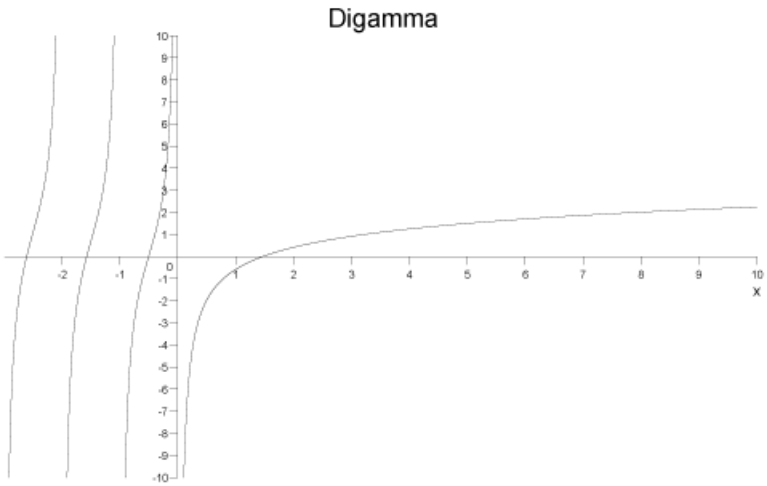
```
#include <boost/math/special_functions/digamma.hpp>
```

```
namespace boost{ namespace math{

template <class T>
calculated-result-type digamma(T z);

}} // namespaces
```

Description

Returns the digamma or psi function of *x*. Digamma is defined as the logarithmic derivative of the gamma function:

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



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 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:

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

can be used. However, this expansion is divergent after a few terms: exactly how many terms depends on the size of *x*. Therefore the value of *BIG* must be chosen so that the series can be truncated at a term that is too small to have any effect on the result when evaluated at *BIG*. Choosing BIG=10 for up to 80-bit reals, and BIG=20 for 128-bit reals allows the series to truncated after a suitably small number of terms and evaluated as a polynomial in `1/(x*x)`.

The rational approximation in the range [1,2] is derived as follows.

First a high precision approximation to digamma was constructed using a 60-term differentiated Lanczos approximation, the form used is:

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

Where P(x) and Q(x) are the polynomials from the rational form of the Lanczos sum, and P'(x) and Q'(x) are their first derivatives. The Lanzos part of this approximation has a theoretical precision of ~100 decimal digits. However, cancellation in the above sum will reduce that to around `99-(1/y)` digits if *y* is the result. This approximation was used to calculate the positive root of digamma, and was found to agree with the value used by Cody to 25 digits (See Math. Comp. 27, 123-127 (1973) by Cody, Strecok and Thacher) and with the value used by Morris to 35 digits (See TOMS Algorithm 708).

Likewise a few spot tests agreed with values calculated using functions.wolfram.com to >40 digits. That's sufficiently precise to insure that the approximation below is accurate to double precision. Achieving 128-bit long double precision requires that the location of the root is known to ~70 digits, and it's not clear whether the value calculated by this method meets that requirement: the difficulty lies in independently verifying the value obtained.

The rational approximation 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***.