# Using Boost.Math with Boost.Multiprecision

January 14, 2013

We will now describe a variety of ways to use Boost.Math in combination with Boost.Multiprecision in order to perform floating-point numerical calculations with high precision.

The Boost.Multiprecision library can be used for computations requiring precision exceeding that of standard built-in types such as float, double and long double. For extended-precision calculations, Boost.Multiprecision supplies a template data type called cpp_dec_float. The number of decimal digits of precision is fixed at compile-time via template parameter. For example,

```
#include <boost/multiprecision/cpp_dec_float.hpp>

using boost::multiprecision::cpp_dec_float;

// A floating-point type with 50 digits of precision.
typedef cpp_dec_float<50> float50;
```

Here, we have defined the local data type float50 with with 50 decimal digits of precision. We can use this data type to print, for example, the value of 1/7 to 50 digits.

```
#include <iostream>
#include <limits>

// 1/7 with 50 digits of precision.
float50 seventh = float50(1) / 7;

const int d
  = std::numeric_limits<float50>::digits10;

// Print 1/7 with 50 digits of precision.
std::cout << std::setprecision(d)
          << seventh
```

```
          << std::endl;
```

For the sake of convenience, `Boost.Multiprecision` defines a variety of data types with fixed precision. These include, among others, data types with 50 and 100 decimal digits of precision, respectively. In particular,

```
using boost::multiprecision;

cpp_dec_float_50  f50;  // Has  50 digits.
cpp_dec_float_100 f100; // Has 100 digits.
```

For more information on the `cpp_dec_float` data type, see the tutorial in `Boost.Multiprecision`.

Mathematical software usually requires one or more known constant values such as $\pi$ or $\log 2$, etc. It often makes sense to initialize a constant value stored in a built-in type from a multiprecision value of higher precision. This guarantees that the built-in type will be initialized to the the very last bit of its own precision.

We will now obtain the value of $\pi$ with 50 decimal digits of precision from `Boost.Math`'s template `pi()` function.

```
using boost::multiprecision;

cpp_dec_float_50 pi_50;
  = boost::math::constants::pi<cpp_dec_float_50>();
```

A multiprecision value can be converted to a built-in type using the member function `convert_to()`. In particular, the code below initializes a value of type `long double` from a multiprecision value having 50 digits of precision.

```
const long double pi_ld
  = pi_50.convert_to<long_double>();
```

One usually needs to compute tables of numbers in mathematical software. A fast Fourier transform (FFT), for example, may use a table of the values of $\sin(\pi/2^n)$ in its implementation details. In order to maximize the precision in the FFT implementation, the precision of the tabulated trigonometric values should exceed that of the built-in floating-point type used in the FFT.

The sample below computes a table of the values of $\sin(\pi/2^n)$ in the range $1 \le n \le 31$. A precision of 50 decimal digits is used.

```
#include <vector>
#include <algorithm>
```

```
#include <iostream>
#include <iomanip>
#include <iterator>
#include <limits>
#include <boost/multiprecision/cpp_dec_float.hpp>

using boost::multiprecision::cpp_dec_float_50;
using boost::math::constants::pi;

typedef cpp_dec_float_50 float50;

int main(int, char**)
{
  std::vector<float50> sin_values(31U);

  unsigned n = 1U;

  // Generate the sine values.
  std::for_each(
    sin_values.begin(),
    sin_values.end(),
    [&n](float50& y)
    {
      y = sin(pi<float50>() / pow(float50(2), n));
      ++n;
    });

  // Set the output precision.
  const int d
    = std::numeric_limits<float50>::digits10;

  std::cout.precision(d);

  // Print the sine table.
  std::copy(sin_values.begin(),
            sin_values.end(),
    std::ostream_iterator<float50>(std::cout, "\n"));
}
```

This program makes use of, among other program elements, the data type `cpp_dec_float_50` from `Boost.Multiprecision`, the value of $\pi$ retrieved from `Boost.Math` and a C++11 lambda function combined with `std::for_each()`.

The output of this program is shown below.

```
1
0.70710678118654752440084436210484903928483593768847
0.38268343236508977172845998403039886676134456248563
0.19509032201612826784828486847702224092769161775195
0.09801714032956060199419556388864184586136673167501
0.04906767432741801425495497694268265831474536302575 3
0.02454122852291228803173452945928292506546611923945 1
0.01227153828571992607940826195100321214037231959176 9
0.00613588464915447535964023459037258091705788631739 13
0.00306795676296597627014536549091984251894461021345 2
0.00153398018628476561230369715026407907995486457523 74
0.00076699031874270452693856835794857664314091945206328
0.00038349518757139558907246168118138126339502603496474
0.00019174759731070330743990956198900093346887403385916
9.58737990959773458705172109764763511870656128511 45e-05
4.79368996030668845490039904946588727468666876857 67e-05
2.39684498084182187291865771650218200947614748956 73e-05
1.19842249050697064215215615969889848047319775383 87e-05
5.99211245264242784287971180889086172998717787809 51e-06
2.99605622633466075045481280835705981182518786834 08e-06
1.49802811316901122885427884615536112069175858615 27e-06
7.49014056584715721130498566730655637155959302172 07e-07
3.74507028292384123903169179084633177397404762972 48e-07
1.87253514146195344868824576593563617120452720982 87e-07
9.36267570730980827990672866808856201932365071694 73e-08
4.68133785365490926951155181385400969595036270166 7e-08
2.34066892682745527595054934190348440378862072237 79e-08
1.17033446341372771812462135032381037980934566399 76e-08
5.85167231706863869080979010083413969439000850517 57e-09
2.92583615853431935792823046906895590201758571500 74e-09
1.46291807926715968052953216186596371037426152278 34e-09
```

This output can be copied as text and readily integrated into a given source code file. Alternatively, the output can be written to a text file or even be used as part of a self-written automatic code generator.

A computer algebra system can be used to verify the results obtained from `Boost.Math` and `Boost.Multiprecision`. For example, the Mathematica® computer algebra system [1] can obtain a similar table with the command:

```
Table[N[Sin[Pi / (2^n)], 50], {n, 1, 31, 1}]
```

The WolframAlpha® Computational Knowledge Engine [2] can also be used to generate this table. The same command can be used.

In general, calculations of special functions inherently lose a few bits of precision (or more) due to cancellations in series expansions, recurrence relations and the like. It often arises, however, that data tables originating from special function calculations are used as the basis of an even higher-level calculation. Computing data tables with extended precision can improve the overall accuracy of these kinds of calculations because the data table has more precision than necessary.

It may be wise to warm-cache pre-computed high-precision table values in a static container such as `std::vector`. It can also be convenient to store pre-computed high-precision data tables in text-based files that can be parsed whenever the values need to be initialized and warm-cached.

For our final example, we will use `Boost.Math` and `Boost.Multiprecision` to generate a table of high-precision real-valued roots of cylindrical Bessel functions $J_\nu(x)$ on the positive real axis.

The roots of cylindrical Bessel functions can be found by solving

$$J_\nu\left(j_{\nu,\,m}\right) \;=\; 0\,, \tag{1}$$

where $j_{\nu,\,m}$ represents the $m^{\text{th}}$ root of the cylindrical Bessel function of order $\nu$.

The calculation uses McMahon's approximation to find initial estimates for $j_{\nu,\,m}$, as described in Sect. 10.21(`vi`) of [3] and the references therein. The precision of a given root is subsequently refined to the desired level using Newton iteration.

McMahon's asymptotic approximation for $j_{\nu,\,m}$ with $\nu \geq 0$ and $m \to \infty$ is given by Eq. 10.21.19 in [3]

$$\begin{aligned} j_{\nu,\,m} \;=\; & a \,-\, \frac{\mu-1}{8a} \,-\, \frac{4(\mu-1)(7\mu-31)}{3(8a)^3} \,-\, \frac{32(\mu-1)(83\mu^2-982\mu+3779)}{15(8a)^5} \\ & -\, \frac{64(\mu-1)(6949\mu^3-1\,53855\mu^2+15\,85743\mu-62\,77237)}{105(8a)^7} \,-\, \cdots\,, \end{aligned} \tag{2}$$

where $\mu = 4\nu^2$ and $a = \left(m + \frac{1}{2}\nu - \frac{1}{4}\right)\pi$.

The specialized expression for $j_{\nu,\,1}$ is explicitly given by Eq. 10.21.19 in [3]

$$\begin{aligned} j_{\nu,\,1} \;=\; & \nu + 1.85575\,71\nu^{\frac{1}{3}} + 1.03315\,0\nu^{-\frac{1}{3}} - 0.00397\nu^{-1} \\ & -0.0908\nu^{-\frac{5}{3}} + 0.043\nu^{-\frac{7}{3}} + \cdots\,, \end{aligned} \tag{3}$$

Newton iteration requires both $J_\nu(x)$ as well as its derivative. The derivative of $J_\nu(x)$ with respect to $x$ is given by Eq. 10.6.2 in [3]

$$\frac{d}{dx}J_\nu(x) \;=\; J_{\nu-1}(x) \,-\, \frac{\nu}{x}\,J_\nu(x)\,. \tag{4}$$

The sample code shown below can be used to find the roots of $J_\nu(x)$. This program can also be found in the link here. In this example, a subroutine

called `cyl_bessel_j_zero()` is created and used to compute the first 20 roots of $J_{\frac{71}{19}}(x)$ for $x$ on the positive real axis with 50 decimal digits of precision.

TBD: Make and show sample code.

The output of the program is shown below.

TBD: Show the program output.

The results of this program can be verified with Mathematica® using the following command:

```
Table[N[BesselJZero[71/19, n], 50], {n, 1, 20, 1}]
```

# References


[1] Wolfram: *Wolfram Mathematica®*, `http://www.wolfram.com/mathematica` (2013)

[2] Wolfram: *WolframAlpha® Comutational Knowledge Engine*, `http://www.wolframalpha.com` (2013)

[3] F. W. J. Olver, D. W. Lozier, R. F. Boisvert and C. W. Clark: *NIST Handbook of Mathematical Functions*, (NIST National Instutute of Standards and Technology US Department of Commerce and Cambridge University Press, New York, 2010)