# Countering Precision Loss with `Boost.Multiprecision`

January 29, 2013

The following example shows how multiprecision calculations can be used to obtain full precision in a numerical derivative calculation that suffers from precision loss. Consider some well-known central difference rules for numerically computing the first derivative of a function $f'(x)$ with $x \in \mathbb{R}$.

$$\begin{aligned} f'(x) &\approx m_1 + O(dx^2) \\ f'(x) &\approx \frac{4}{3}m_1 - \frac{1}{3}m_2 + O(dx^4) \\ f'(x) &\approx \frac{3}{2}m_1 - \frac{3}{5}m_2 + \frac{1}{10}m_3 + O(dx^6)\,. \end{aligned} \tag{1}$$

Here, the difference terms $m_n$ are given by

$$\begin{aligned} m_1 &= \frac{f(x+dx) - f(x-dx)}{2dx} \\ m_2 &= \frac{f(x+2dx) - f(x-2dx)}{4dx} \\ m_3 &= \frac{f(x+3dx) - f(x-3dx)}{6dx}\,, \end{aligned} \tag{2}$$

where $dx$ is the incremental step-size used when calculating the derivative. The third equation represents a three-point central difference rule with precision $O(dx^6)$.

One possible implementation of this three-point central difference rule is shown below.

```
template<typename value_type,
         typename function_type>
value_type derivative(const value_type x,
                      const value_type dx,
                      function_type function)
{
```

```
  // Compute the derivative of function using a
  // three-point central difference rule of O(dx^6).

  const value_type dx2(dx * 2U);
  const value_type dx3(dx * 3U);

  const value_type m1((  function(x + dx)
                       - function(x - dx))  / 2U);
  const value_type m2((  function(x + dx2)
                       - function(x - dx2)) / 4U);
  const value_type m3((  function(x + dx3)
                       - function(x - dx3)) / 6U);

  const value_type fifteen_m1(m1 * 15U);
  const value_type six_m2    (m2 *  6U);
  const value_type ten_dx    (dx * 10U);

  return ((fifteen_m1 - six_m2) + m3) / ten_dx;
}
```

Here, the implementation uses a C++ template that can be instantiated with various floating-point types such as `float`, `double`, `long double`, or even a user-defined floating-point type.

We will now use the `derivative` template with the built-in type `double` in order to numerically compute the derivative of a function. Consider the function shown below.

$$f(x) = \sqrt{x^2 - 1} - \cos^{-1}\left(\frac{1}{x}\right) \tag{3}$$

We will now take the derivative of this function with respect to $x$ evaluated at $x = 3/2$. In other words,

$$\frac{d}{dx}\left[\sqrt{x^2 - 1} - \cos^{-1}\left(\frac{1}{x}\right)\right]\Bigg|_{x=\frac{3}{2}} . \tag{4}$$

The expected result is

$$\frac{\sqrt{x^2 - 1}}{x}\Bigg|_{x=\frac{3}{2}} = \frac{\sqrt{5}}{3} \approx 0.74535\,59924\,99929\,89880\,. \tag{5}$$

The program below uses the `derivative` template in order to perform the numerical calculation of this derivative. The program also compares the numerically-obtained result with the expected result and reports the absolute relative error scaled to a deviation that can easily be related to the number of bits of lost precision.

```
#include <iostream>
#include <iomanip>
#include <limits>
#include <cmath>

int main(void)
{
  const double d =
    derivative(1.5,
               std::ldexp(1.0, -9),
               [](const double& x_) -> double
               {
                 return   std::sqrt((x_ * x_) - 1.0)
                        - std::acos(1.0 / x_);
               });

  const double rel_error =
      (d - 0.74535599249992989880)
    /      0.74535599249992989880;

  const double bit_error =
      std::abs(rel_error)
    / std::numeric_limits<double>::epsilon();

  std::cout.precision(
    std::numeric_limits<double>::digits10);

  std::cout << "derivative: "
            << d
            << std::endl;

  std::cout << "expected  : "
            << 0.74535599249992989880
            << std::endl;

  std::cout << "bit_error : "
            << unsigned long(bit_error)
            << std::endl;
}
```

The result of this program on a system with an eight-byte, IEEE-754 conforming floating-point representation for `double` is:

```
derivative: 0.745355992499951
```

```
expected  : 0.74535599249993
bit_error : 130
```

Here, the `bit_error` variable is calculated from the absolute relative error scaled with $\epsilon$, where $\epsilon$ is `std::numeric_limits<double>::epsilon()` and is equal to the smallest number that differs from one that is representable in the floating-point system. The derivative has a `bit_error` of 130, corresponding to approximately 7 bits of precision loss.

In this example, we have carefully selected a step-size $dx$ of $2^{-9}$ using the library function `ldexp()`. This is done because numerical differentiation generally produces the best results when a step-size that is exactly representable in the floating-point system is used, in other words a pure power of 2 for an IEEE-754 representation.

If the calculation is repeated using step sizes of $2^{-n}$ with $n = 7, 8, 9, 10,$ and 11, the best result is in fact found using a step-size of $2^{-9}$. Empirical experiments with other step-sizes in various bases on this system have all resulted in worse values. So using the step-size of $2^{-9}$ may, in fact, produce the best numerical result that can be obtained for the built-in type `double` when used for this numerical derivative on this system.

With $dx = 2^{-9}$, the derivative central difference rule produces a very good result. It still, however, suffers from approximately 7 bits of precision. A better result can be obtained by using a multiprecision type from `Boost.Multiprecision` such as `boost::multiprecision::cpp_dec_float`.

We will now repeat the numerical derivative calculation shown above using a multiprecision type. First, we need to define a multiprecision type using the facilities in `Boost.Multiprecision`.

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

using boost::multiprecision::number;
using boost::multiprecision::cpp_dec_float;

#define MP_DIGITS10 \
  unsigned( \
  std::numeric_limits<double>::max_digits10 + 5)

typedef cpp_dec_float<MP_DIGITS10> mp_backend;

typedef number<mp_backend> mp_type;
```

Here, the user-defined data type `mp_type` has been defined with 5 decimal digits of precision more than the built-in type `double`.

The following program repeats the derivative calculation. This time, however, the multiprecision type `mp_type` is used. In particular,

```
int main(void)
{
  const mp_type mp =
    derivative(mp_type(mp_type(3) / 2U),
               mp_type(mp_type(1) / 10000000U),
               [](const mp_type& x_) -> mp_type
               {
                 return   sqrt((x_ * x_) - 1.0)
                        - acos(1.0 / x_);
               });

  const double d = mp.convert_to<double>();

  const double rel_error =
      (d - 0.74535599249992989880)
    /      0.74535599249992989880;

  const double bit_error =
      std::abs(rel_error)
    / std::numeric_limits<double>::epsilon();

  std::cout.precision(
    std::numeric_limits<double>::digits10);

  std::cout << "derivative: "
            << d
            << std::endl;

  std::cout << "expected  : "
            << 0.74535599249992989880
            << std::endl;

  std::cout << "bit_error : "
            << unsigned long(bit_error)
            << std::endl;
```

Note that the derivative calculation is carried out with the multiprecision type. The result of the multiprecision calculation is subsequently converted to the built-in type `double` using the template `convert_to()` member function.

The result of this program is:

```
derivative: 0.74535599249993
expected  : 0.74535599249993
bit_error : 0
```

The resulting bit error is 0. This means that the result of the derivative calculation is bit-identical with the `double` representation of the expected result, and this is the best result possible for the built-in type.

The derivative in this example has a known closed form. There are, however, countless situations in numerical analysis (and not only for numerical derivatives) for which the calculation at hand does not have a known closed-form solution or for which the closed-form solution is highly inconvenient to use. In such cases, this technique may be useful.

This example has shown how multiprecision can be used to add extra digits to an ill-conditioned calculation that suffers from precision loss. When the result of the multiprecision calculation is converted to a built-in type such as `double`, the entire precision of the result in `double` is preserved.