1 Generic Numeric Programming

Generic numeric programming employs templates to use the same code for different floating-point types and functions. Consider the area of a circle a of radius r, given by

$$a = \pi r^2 . \tag{1}$$

The area of a circle can be computed in generic programming using Boost.Math as shown below.

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

using boost::math::constants::pi;

template<typename T>
inline T area_of_a_circle(T r)
{
  return pi<T>() * (r * r);
}
```

It is possible to use `area_of_a_circle()` with built-in floating-point types as well as floating-point types from Boost.Multiprecision. In particular, consider a system with 4–byte single-precision **float**, 8–byte double-precision **double** and also the `cpp_dec_float_50` data type from Boost.Multiprecision with 50 decimal digits of precision.

We can compute and print the approximate area of a circle with radius 123/ 100 for **float**, **double** and `cpp_dec_float_50` with the program below.

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

using boost::multiprecision::cpp_dec_float_50;

int main(int , char**)
{
  const float r_f(float(123) / 100);
  const float a_f = area_of_a_circle(r_f);

  const double r_d(double(123) / 100);
  const double a_d = area_of_a_circle(r_d);

  const cpp_dec_float_50 r_mp(cpp_dec_float_50(123) / 100);
  const cpp_dec_float_50 a_mp = area_of_a_circle(r_mp);

  // 4.75292
```

```
  std::cout
    << std::setprecision(std::numeric_limits<float>::digits10)
    << a_f
    << std::endl;

  // 4.752915525616
  std::cout
    << std::setprecision(std::numeric_limits<double>::digits10)
    << a_d
    << std::endl;

  // 4.7529155256159981904701331745635599135018975843146
  std::cout
    << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
    << a_mp
    << std::endl;
}
```

Let's add even more power to generic numeric programming using not only different floating-point types but also function objects as template parameters. Consider some well-known central difference rules for numerically computing the first derivative of a function $f'(x)$ with $x \in \mathbb{R}$.

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

where the difference terms m_n are given by

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

and dx is the step-size of the derivative.

The third formula in Equation 2 is a three-point central difference rule. It calculates the first derivative of $f(x)$ to $O(dx^6)$, where dx is the given step-size. For example, if the step-size is 0.01 this derivative calculation has about 6 decimal digits of precision—just about right for the 7 decimal digits of single-precision **float**.

Let's make a generic template subroutine using this three-point central difference rule. In particular,

```
template<typename value_type,
         typename function_type>
value_type derivative(const value_type x,
                      const value_type dx,
                      function_type func)
{
  // Compute d/dx[func(*first)] using a three-point
  // central difference rule of O(dx^6).

  const value_type dx1 = dx;
  const value_type dx2 = dx1 * 2;
  const value_type dx3 = dx1 * 3;

  const value_type m1 = (  func(x + dx1)
                         - func(x - dx1)) / 2;
  const value_type m2 = (  func(x + dx2)
                         - func(x - dx2)) / 4;
  const value_type m3 = (  func(x + dx3)
                         - func(x - dx3)) / 6;

  const value_type fifteen_m1 = 15 * m1;
  const value_type six_m2     =  6 * m2;
  const value_type ten_dx1    = 10 * dx1;

  return ((fifteen_m1 - six_m2) + m3) / ten_dx1;
}
```

The `derivative()` template function can be used to compute the first derivative of any function to $O(dx^6)$. For example, consider the first derivative of $\sin x$ evaluated at $x = \pi/3$. In other words,

$$\frac{d}{dx} \sin x \bigg|_{x=\frac{\pi}{3}} = \cos \frac{\pi}{3} = \frac{1}{2} . \tag{4}$$

The code below computes the derivative in Equation 4 for **float**, **double** and boost's multiple-precision type `cpp_dec_float_50`. The code uses the `derivative()` function in combination with a lambda expression.

```
#include <iostream>
#include <iomanip>
#include <boost/multiprecision/cpp_dec_float.hpp>
#include <boost/math/constants/constants.hpp>

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

```
int main(int, char**)
{
  const float d_f =
    derivative(float(pi<float>() / 3),
               0.01F,
               [](const float x) -> float
               {
                 return ::sin(x);
               });

  const double d_d =
    derivative(double(pi<double>() / 3),
               0.001,
               [](const double x) -> double
               {
                 return ::sin(x);
               });

  const cpp_dec_float_50 d_mp =
    derivative(cpp_dec_float_50(pi<cpp_dec_float_50>() / 3),
               cpp_dec_float_50(1.0E-9),
               [](const cpp_dec_float_50 x) -> cpp_dec_float_50
               {
                 return boost::multiprecision::sin(x);
               });

  // 0.500003
  std::cout
    << std::setprecision(std::numeric_limits<float>::digits10)
    << d_f
    << std::endl;

  // 0.499999999999888
  std::cout
    << std::setprecision(std::numeric_limits<double>::digits10)
    << d_d
    << std::endl;

  // 0.50000000000000000000000000000000000000003925935
  std::cout
    << std::setprecision(std::numeric_limits<cpp_dec_float_50>::digits10)
    << d_mp
    << std::endl;
}
```

The expected value of the derivative is 0.5. This central difference rule in this example is ill-conditioned, meaning it suffers from slight loss of precision. With that in mind, the results agree with the expected value of 0.5.

A generic numerical integration template similar to the `derivative()` template is shown below.

```
template<typename value_type,
         typename function_type>
inline value_type integral(const value_type a,
                           const value_type b,
                           const value_type tol,
                           function_type func)
{
  unsigned n = 1U;

  value_type h = (b - a);
  value_type I = (func(a) + func(b)) * (h / 2);

  for(unsigned k = 0U; k < 8U; k++)
  {
    h /= 2;

    value_type sum(0);
    for(unsigned j = 1U; j <= n; j++)
    {
      sum += func(a + (value_type((j * 2) - 1) * h));
    }

    const value_type I0 = I;
    I = (I / 2) + (h * sum);

    const value_type ratio     = I0 / I;
    const value_type delta     = ratio - 1;
    const value_type delta_abs = ((delta < 0) ? -delta : delta);

    if((k > 1U) && (delta_abs < tol))
    {
      break;
    }

    n *= 2U;
  }

  return I;
}
```