Generic Numeric Programming with Boost.Math and Boost.Multiprecision

Anton Bikineev

Generic Numeric Programming

“Generic Numeric Programming employs templates to use the same code for different floating-point types and functions”

Christopher Kormanyos, “Real-time C++”

“Generic numeric programming refers to implementations of algorithms and data structures that can be used with different underlying numeric types”

Erik Osheim, “Generic Numeric Programming
Through Specialized Type Classes“

Boost.Math overview

Statistical Distributions:
- continuous;
- discrete;

Mathematical Special Functions:
- gamma, beta & erf;
- factorials and binomial coefficients;
- Bessel functions;
- elliptic integrals and functions;
- …;

Implementation Toolkit:
- infinite series;
- continued fractions;
- rational approximations;
- …;

Constants, Quaternions, Octonions...

Distributions overview

namespace boost{ **namespace** math{

Bernoulli	bernoulli_distribution<T>
Poisson	poisson_distribution<T>
Binomial	binomial_distribution<T>
Normal (Gaussian)	normal_distribution<T>, lognormal_distribution<T>
Geometric	geometric_distribution<T>
Beta	beta_distribution<T>
Gamma	gamma_distribution<T>, inverse_gamma_distribution<T>
Uniform	uniform_distribution<T>
Students t	students_t_distribution<T>
Chi Squared	chi_squared_distribution<T>, inverse_chi_squared_distribution<T>
...	fisher_f_distribution<T>, exponential_distribution<T>, hypergeometric_distribution<T>, laplace_distribution<T>, ...

}}

Free functions for distributions

```
namespace boost{ namespace math{

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

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

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

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

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

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

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

…

}}
```

Special functions overview

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

Gamma Functions	tgamma, lgamma, digamma, ...
Factorials and Binomial Coefficients	factorial, double_factorial, binomial_coefficient, ...
Beta Functions	beta, ibeta, ibeta_inv, ...
Error Functions	erf, erf_inv, ...
Polynomials	legendre_p, laguerre, hermite, ...
Bessel Functions	cyl_bessel_j, cyl_neumann, sph_bessel, ...
Hankel Functions	cyl_hankel_1, cyl_hankel_2, sph_hankel_1, ...
Airy functions	airi_ai, airi_bi, airi_ai_prime, ...
Elliptic Integrals	ellint_rf, ellint_1, ellint_2, ...
Jacobi Elliptic Functions	jacobi_elliptic, jacobi_cd, jacobi_cn, ...
Zeta Functions	zeta
...	sin_pi, log1p, expm1, cbrt, hypot, sinc_pi, acosh, asinh, ...

```
}}
```

Policies

```
template <class RealType, class Policy>
class binomial_distribution;
```

```
template <class RealType, class Policy>
promoted_type tgamma(RealType z, const Policy& pol);
```

So what is a Policy?

Policies. Main goals

- error handling;
- enabling internal promotion;
- precision for calculating the result.

Policies. Error handling. Error types

- domain error;
- pole error;
- overflow error;
- underflow error;
- denorm error;
- rounding error;
- evaluation error;
- indeterminate result error.

Policies. Error handling. Error actions

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

    enum error_policy_type
    {
      throw_on_error = 0,        // throw an exception.
      errno_on_error = 1,        // set ::errno & return 0, NaN, infinity or best guess.
      ignore_error = 2,          // return 0, NaN, infinity or best guess.
      user_error = 3             // call a user-defined error handler.
    };

}}} // namespaces
```

Policies. Error handling. Default behavior

domain error	throw_on_error
pole error	throw_on_error
overflow error	throw_on_error
underflow error	ignore_error
denorm error	ignore_error
rounding error	throw_on_error
evaluation error	throw_on_error
indeterminate result error	ignore_error

Policies. Error handling. Usage

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

typedef policy<domain_error<ignore_error> > mypolicy;

typedef policy<
   domain_error<errno_on_error>,
   pole_error<errno_on_error>,
   overflow_error<errno_on_error>,
   evaluation_error<errno_on_error>
> c_policy;

int main()
{
   errno = 0;                                                  // Reset.
   cout << "Result of tgamma(30000) is: "
     << tgamma(30000, c_policy()) << endl;                    // Too big parameter
   cout << "errno = " << errno << endl;                       // errno 34 Numerical result out of range.
   cout << "Result of tgamma(-10) is: "
     << boost::math::tgamma(-10, c_policy()) << endl;     // Negative parameter.
   cout << "errno = " << errno << endl;                  // error 33 Numerical argument out of domain.
}
```

Special functions concepts

- internal promotion

```
template <class Real, class Policy>
struct evaluation
{
   typedef Real type;
};

template <class Policy>
struct evaluation<float, Policy>
{
   typedef typename mpl::if_<typename Policy::promote_float_type, double, float>::type type;
};

template <class Policy>
struct evaluation<double, Policy>
{
   typedef typename mpl::if_<typename Policy::promote_double_type, long double, double>::type type;
};
```

Special functions concepts

- result type promotion

A prvalue of an integer type other than bool, char16_t, char32_t, or wchar_t whose integer conversion rank (4.13) is less than the rank of int can be converted to a prvalue of type int if int can represent all the values of the source type; otherwise, the source prvalue can be converted to a prvalue of type unsigned int.

…

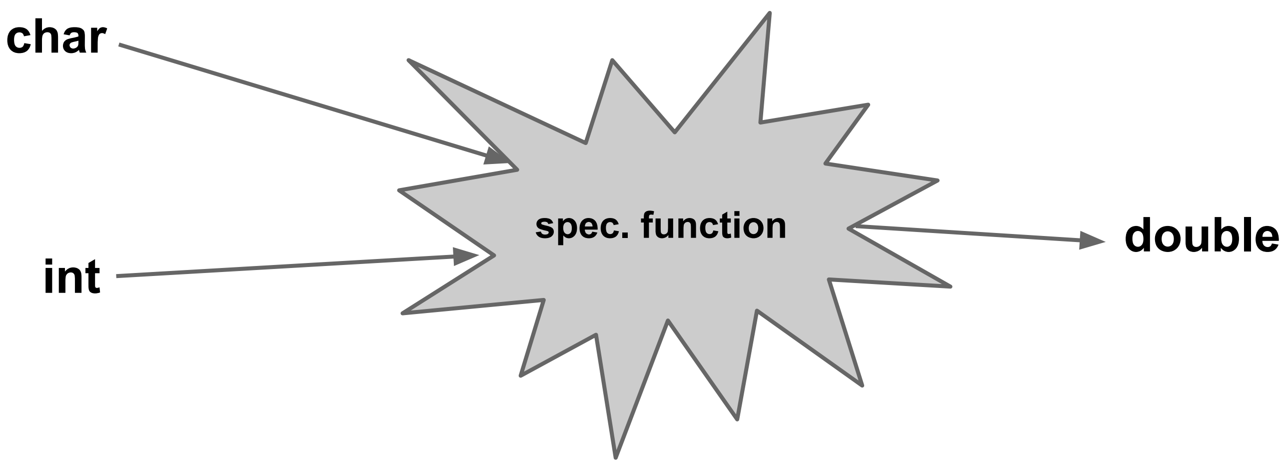
A prvalue of type float can be converted to a prvalue of type double. The value is unchanged. This conversion is called floating point promotion.

N3936: Working Draft

Special functions concepts

- result type promotion

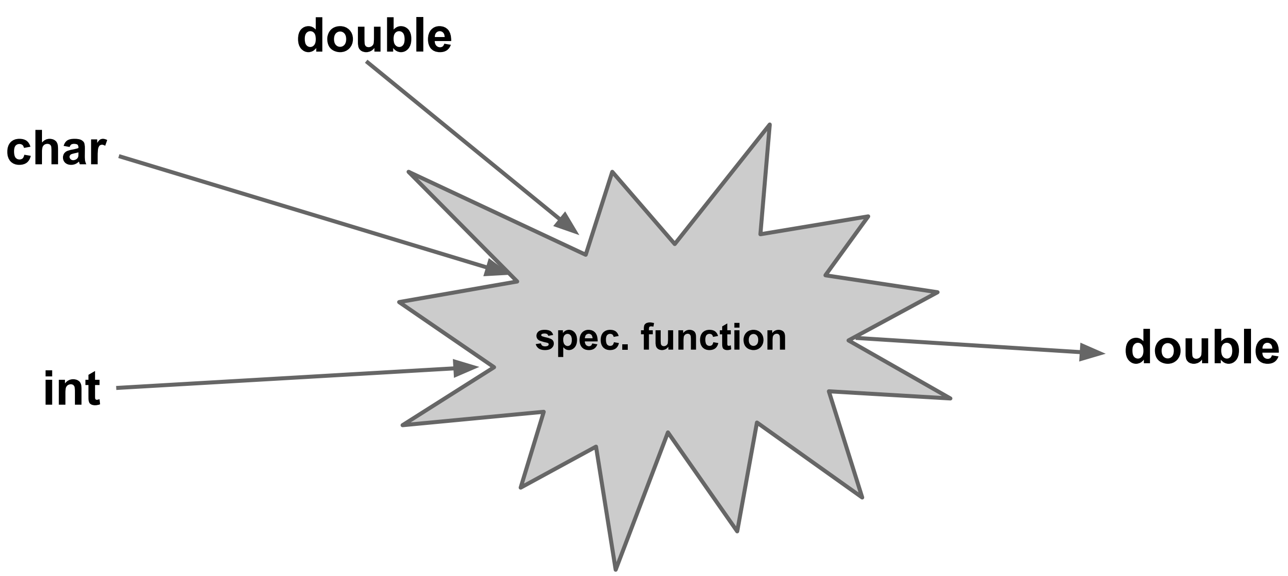
```
template <class… T>
struct promote_args;
```



Special functions concepts

- result type promotion

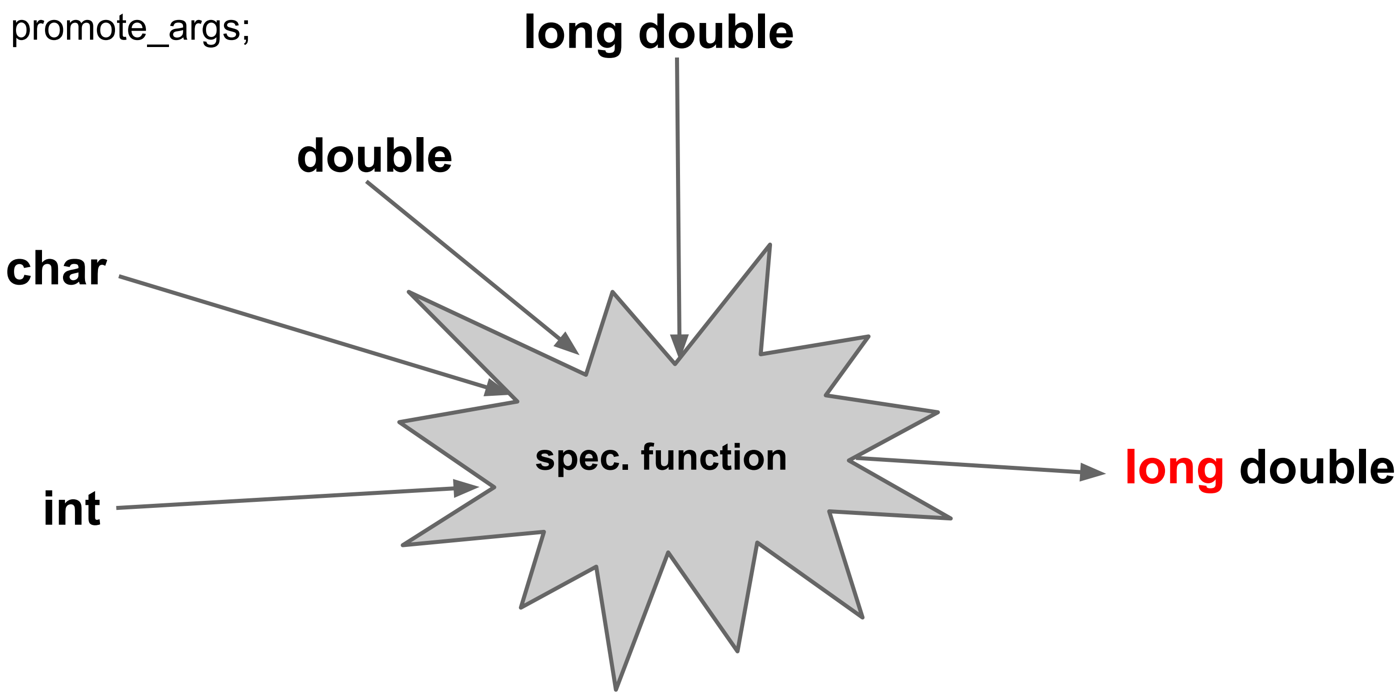
```
template <class… T>
struct promote_args;
```



Special functions concepts

- result type promotion

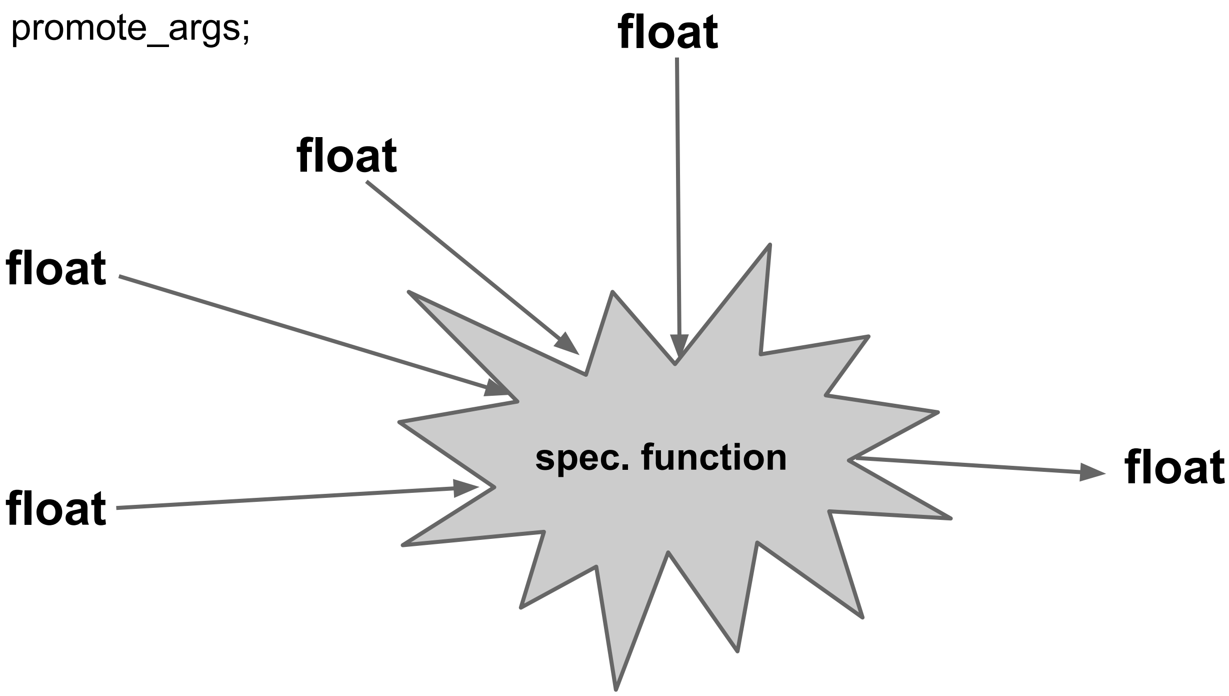
```
template <class… T>
struct promote_args;
```



Special functions concepts

- result type promotion

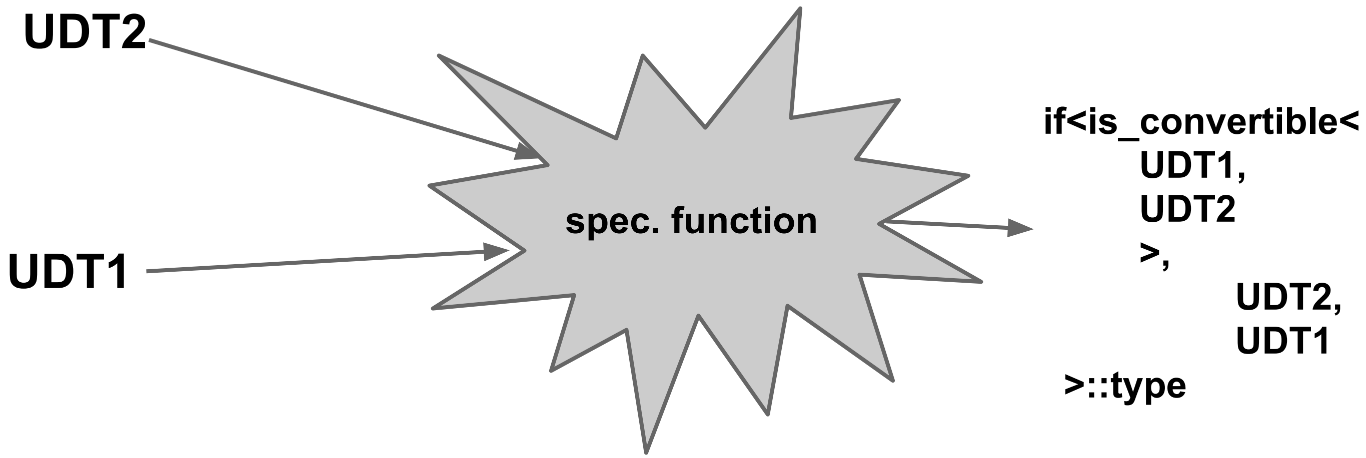
```
template <class… T>
struct promote_args;
```



Special functions concepts

- result type promotion

```
template <class… T>
struct promote_args;
```



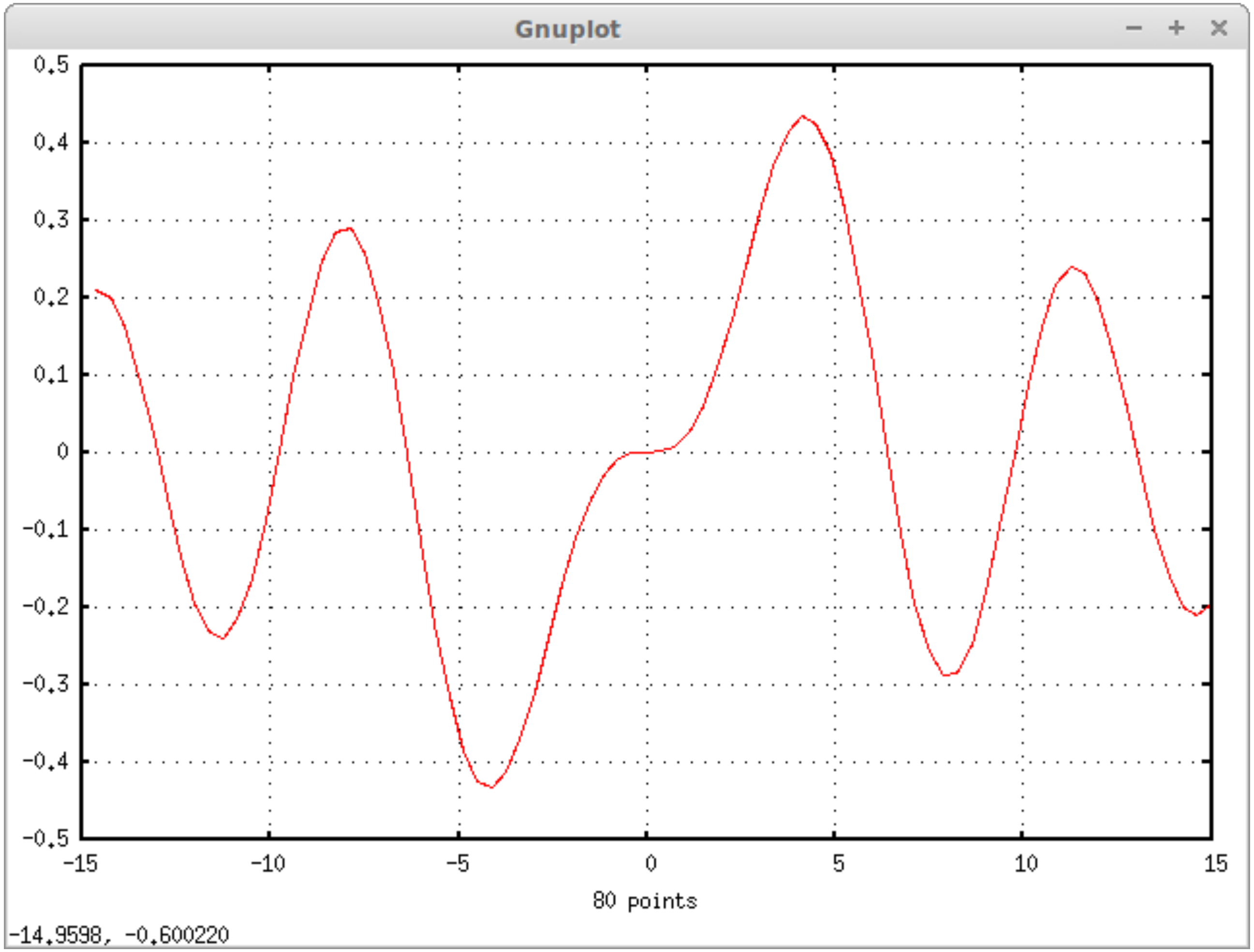
Example: function plotting

```
template <class T, class Func>
void plot_function(const Func& func, T a, T b, size_t num = 40u)
{
  using std::vector;
  using std::generate;
  using std::transform;
  using std::string;
  using boost::lexical_cast;

  BOOST_ASSERT(num != 0);

  vector<T> x(num);
  vector<T> y(num);

  const T d = (b - a) / num;

  generate(x.begin(), x.end(), [&](){
            const T result = a;
            a += d;
            return a;
        });
  transform(x.begin(), x.end(), y.begin(), func);

  static Gnuplot gplot;
  gplot.set_xlabel(lexical_cast<string>(num) + " points");
  gplot.set_style("lines");
  gplot.set_grid();
  gplot.plot_xy(x, y);
}
```

Example: Bessel function

```
template <class T, class Func>
void plot_function(const Func& func, T a, T b, size_t num = 40u);



void foo()
{
  using std::bind;
  using std::placeholders::_1;
  using boost::math::cyl_bessel_j;

  auto cyl_bessel_3 = bind(cyl_bessel_j<double, double>, 3u, _1);
  plot_function(cyl_bessel_3, -15., 15., 80u);
}
```

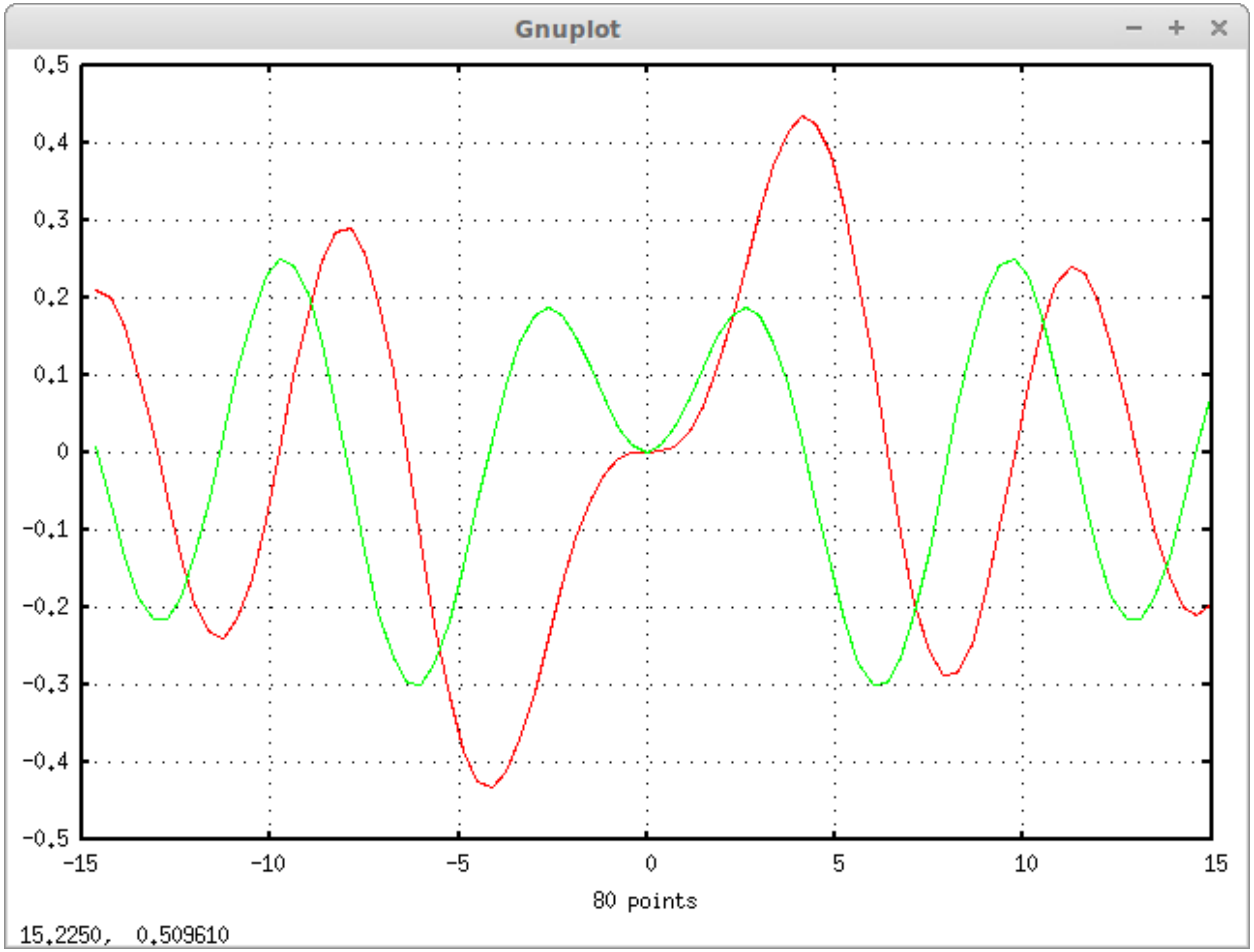
Example: Bessel function



Example: derivative of Bessel function

```
template <class T, class Func>
void plot_function(const Func& func, T a, T b, size_t num = 40u);



void foo()
{
  using std::bind;
  using std::placeholders::_1;
  using boost::math::cyl_bessel_j;
  using boost::math::cyl_bessel_j_prime;

  auto cyl_bessel_3 = bind(cyl_bessel_j<double, double>, 3u, _1);
  plot_function(cyl_bessel_3, -15., 15., 80u);

  auto cyl_bessel_3_prime = bind(cyl_bessel_j_prime<double, double>, 3u, _1);
  plot_function(cyl_bessel_3_derivative, -15., 15., 80u);
}
```

Example: derivative of Bessel function



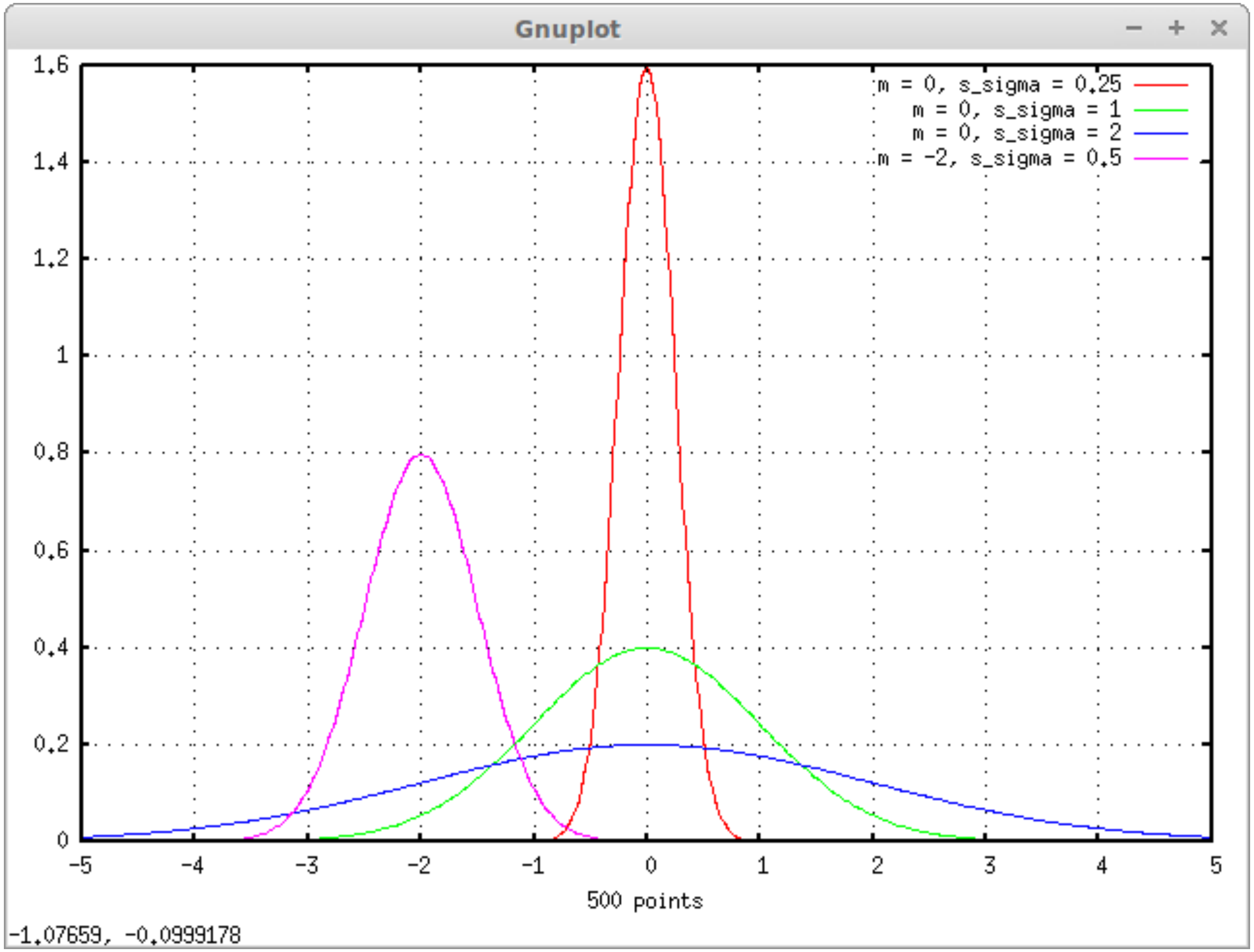
Example: probability density function plotting

```
template <class T, class Distr>
void plot_pdf(const Distr& distr, T a, T b, size_t num = 40)
{
   using std::bind;
   using std::placeholders::_1;
   using boost::math::pdf;
   using boost::math::policies::policy;

   auto binded_pdf = [&distr](T x){ return pdf(distr, x); };

   plot_function(binded_pdf, a, b, num);
}
```

Example: PDFs of normal distributions

```
void foo()
{
  using boost::math::normal;

  const double a = -5., b = 5.;
  const size_t number = 500u;

  plot_pdf(normal(0, 0.25), a, b, number);
  plot_pdf(normal(0, 1), a, b, number);
  plot_pdf(normal(0, 2), a, b, number);
  plot_pdf(normal(-2, 0.5), a, b, number);
}
```

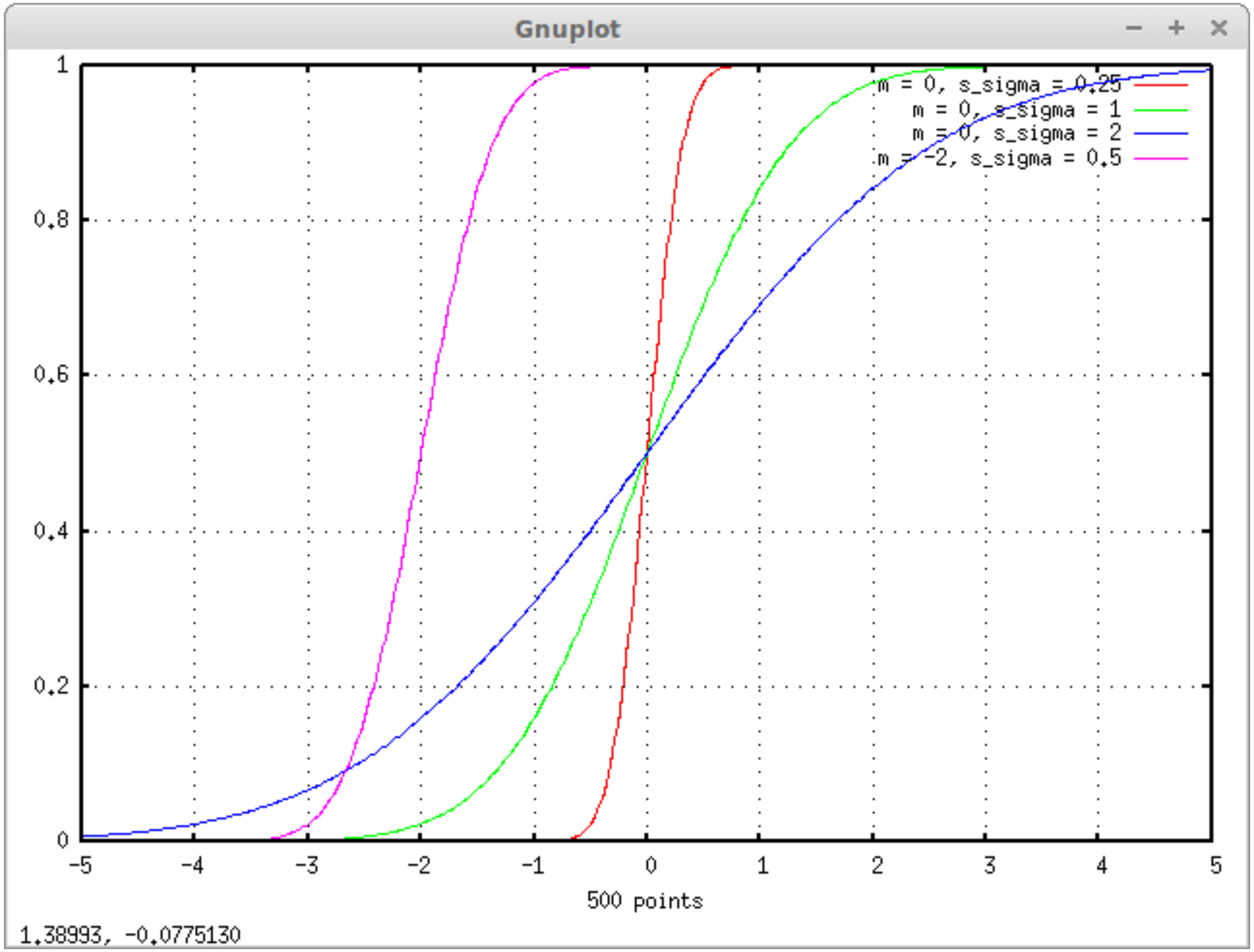
Example: PDFs of normal distributions



Example: cummulative density function plotting

```
template <class T, class Distr>
void plot_pdf(const Distr& distr, T a, T b, size_t num = 40)
{
   using std::bind;
   using std::placeholders::_1;
   using boost::math::cdf;
   using boost::math::policies::policy;

   auto binded_pdf = [&distr](T x){ return cdf(distr, x); };

   plot_function(binded_pdf, a, b, num);
}
```

Example: CDFs of normal distributions



Boost.Multiprecision overview

Front-end:
- expression templates enabling;
- operator overloading;
- code reduction;

Back-end:
- GMP;
- MPFR;
- MPIR;
- TomMath;
- cpp_dec/bin_int/float/rational

number front-end

```
namespace boost{ namespace multiprecision{


enum expression_template_option { et_on = 1, et_off = 0 };

template <class Backend> struct expression_template_default
{ static const expression_template_option value = et_on; };

template <class Backend, expression_template_option
   ExpressionTemplates = expression_template_default<Backend>::value>
class number;


}} // namespaces
```

cpp_int backend

```
namespace boost{ namespace multiprecision{

template <unsigned MinDigits = 0, unsigned MaxDits = 0,
        cpp_integer_type SignType = signed_magnitude,
        cpp_int_check_type Checked = unchecked,
        class Allocator = std::allocator<limb_type> >
class cpp_int_backend;

typedef number<cpp_int_backend<> >              cpp_int;    // arbitrary precision integer

// Fixed precision unsigned types:
typedef number<cpp_int_backend<128, 128, unsigned_magnitude, unchecked, void> >   uint128_t;
typedef number<cpp_int_backend<256, 256, unsigned_magnitude, unchecked, void> >   uint256_t;

// Fixed precision signed types:
typedef number<cpp_int_backend<128, 128, signed_magnitude, unchecked, void> >     int128_t;
typedef number<cpp_int_backend<256, 256, signed_magnitude, unchecked, void> >     int256_t;

}} // namespaces
```

cpp_dec_float and cpp_bin_float backends

```
namespace boost{ namespace multiprecision{

template <unsigned Digits10, class ExponentType = boost::int32_t, class Allocator = void>
class cpp_dec_float;

template <unsigned Digits,
        digit_base_type base = digit_base_10,
        class Allocator = void,
        class Exponent = int,
        ExponentMin = 0,
        ExponentMax = 0>
class cpp_bin_float;

typedef number<cpp_bin_float<50> > cpp_bin_float_50;
typedef number<cpp_bin_float<100> > cpp_bin_float_100;

typedef number<cpp_dec_float<50> > cpp_dec_float_50;
typedef number<cpp_dec_float<100> > cpp_dec_float_100;

}} // namespaces
```

gmp backends

```
namespace boost{ namespace multiprecision{

class gmp_int;

template <unsigned Digits10>
class gmp_float;

typedef number<gmp_int >              mpz_int;

typedef number<gmp_float<50> >        mpf_float_50;
typedef number<gmp_float<100> >       mpf_float_100;
typedef number<gmp_float<500> >       mpf_float_500;
typedef number<gmp_float<1000> >      mpf_float_1000;
typedef number<gmp_float<0> >         mpf_float;

}} // namespaces
```

Toolkit: continued fraction

```
template <class Gen, class U>
typename detail::fraction_traits<Gen>::result_type
   continued_fraction_a(Gen& g, const U& tolerance, boost::uintmax_t& max_terms);
```

$$\frac{a_1}{b_1 + \frac{a_2}{b_2 + \frac{a_3}{b_3 + \frac{a_4}{b_4 + \cdots}}}}$$

Concept for Gen:

```
typedef std::pair<T, T> result_type;
result_type operator()();
```

Continued fraction: pi example

$$\pi = 3 + \boxed{\cfrac{1^2}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{6 + \cfrac{7^2}{6 + \cfrac{9^2}{6 + \dots}}}}}}$$

boost::math::continued_fraction_a:

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

pi example: defining the term struct

```
template <class T>
struct pi_continued_fraction_term
{
   typedef std::pair<T, T> result_type;

   std::pair<T, T> operator()()
   {
      BOOST_MATH_STD_USING
      a += 2;
      return std::make_pair(pow(a, 2), T(6));
   }

private:
   T a = -1;
};
```

$$\pi = 3 + \boxed{\cfrac{1^2}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{6 + \cfrac{7^2}{6 + \cfrac{9^2}{6 + \ldots}}}}}}$$

pi example: defining the function

```
template <class T>
T pi_fraction()
{
   using boost::math::policies::get_max_series_iterations;
   using boost::math::policies::get_epsilon;
   using boost::math::policies::policy;

   pi_continued_fraction_term<T> s;
   boost::uintmax_t max_iter = get_max_series_iterations<policy<> >();

   T result = boost::math::tools::continued_fraction_a(
      s,
      get_epsilon<T, policy<> >(),
      max_iter);

   result = 3 + result;
   return result;
}
```

$$\pi = 3 + \boxed{\cfrac{1^2}{6 + \cfrac{3^2}{6 + \cfrac{5^2}{6 + \cfrac{7^2}{6 + \cfrac{9^2}{6 + \dots}}}}}}$$

pi example: testing

```
int main()
{
  using namespace boost::multiprecision;
  using gmp_type = number<backends::gmp_float<100u>, ET_OPT>;

  const auto gmp_pi = pi_fraction<gmp_type>();
  std::cout << gmp_pi.str();

  return 0;
}
```

```
$ g++ -DET_OPT=et_off -o pi_contfrac pi_contfrac.cc -std=c++14 -lgmp
$ time pi_contfrac
3.14159265358979323871264188328519036732221160240626227031435322607832332034602626890147343
80117016748192756
real    0m8.347s
user    0m8.312s

$ g++ -DET_OPT=et_on -o pi_contfrac pi_contfrac.cc -std=c++14 -lgmp
$ time pi_contfrac
3.14159265358979323871264188328519036732221160240626227031435322607832332034602626890147343
80117016748192756
real    0m8.190s
user    0m8.180s
```

Chudnovsky pi series example

Algorithm for calculation the number pi based on rapidly convergent series:

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

Chudnovsky pi series example

Let's get rid of fractional power:

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}} =$$

$$= \frac{12}{640320\sqrt{640320}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k}} =$$

$$= \frac{12}{426880\sqrt{10005}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k}}$$

Chudnovsky pi: defining the term struct

$$\frac{12}{426880\sqrt{10005}}\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k}}$$

```
template <class T>
struct chudnovsky_series_term
{
  typedef T result_type;

  T operator()() const
  {
    BOOST_MATH_STD_USING
    using boost::math::factorial;

    static size_t k = 0;
    const T num = factorial<T>(6 * k) * (a1 + (a2 * k));
    const T denom = (factorial<T>(3 * k) * pow(factorial<T>(k), 3)) * pow(b1, (3 * k));
    const T result = num / denom;

    return (k++ & 1) ? -result : result;
  }

private:
  static const T a1, a2, b1;
};

template<class T>
const T chudnovsky_series_term<T>::a1 = 13591409u;
template<class T>
const T chudnovsky_series_term<T>::a2 = 545140134u;
template<class T>
const T chudnovsky_series_term<T>::b1 = 640320u;
```

Chudnovsky pi: defining the function

```
template <class T>
inline T chudnovsky_pi()
{
   BOOST_MATH_STD_USING
   using boost::math::policies::get_max_series_iterations;
   using boost::math::policies::get_epsilon;
   using boost::math::policies::policy;

   chudnovsky_series_term<T> s;
   boost::uintmax_t max_iter = get_max_series_iterations<policy<> >();

   const T result = boost::math::tools::sum_series(s, get_epsilon<T, policy<> >(), max_iter);
   return (426880u * sqrt(T(10005u))) / result;
}
```

$$\frac{1}{\pi} = \frac{12}{426880\sqrt{10005}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k}}$$

Chudnovsky pi example: testing

```
int main()
{
   using namespace boost::multiprecision;
   using gmp_type = number<backends::gmp_float<100u>, ET_OPT>;

   const auto gmp_pi = chudnovsky_pi<gmp_type>();
   std::cout << gmp_pi.str();

   return 0;
}
```

```
$ g++ -DET_OPT=et_off -o pi_chudnovsky pi_chudnovsky.cc -std=c++14 -lgmp
$ time pi_chudnovsky
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482
253421170679821481
real    0m0.009s
user    0m0.008s

$ g++ -DET_OPT=et_on -o pi_chudnovsky pi_chudnosvky.cc -std=c++14 -lgmp
$ time pi_chudnosvky
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482
253421170679821481
real    0m0.009s
user    0m0.004s
```

Newton Raphson Method

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

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

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

}}} // namespaces
```

Concept for f:

```
std::pair<T, T> operator()(const T& x);
```

pair<T, T>::first - function value;
pair<T, T>::second - function derivative value.

Newton Raphson: generic root finding

Let’s create a generic root finding function that uses Newton Raphson Method:

```
template <class T, class Function>
T find_root(Function function, const T& guess, const T& min, const T& max);
```

Newton Raphson: generic root finding

Let's create a generic root finding function that uses Newton Raphson Method:

```
template <class T, class Function>
T find_root(Function function, const T& guess, const T& min, const T& max);
```

But how are we supposed to get a derivative of any function?

Newton Raphson: numerical derivative

$$f'(x) \approx m_1 + O(dx^2)$$

$$f'(x) \approx \frac{4}{3}m_1 - \frac{1}{3}m_2 + O(dx^4)$$

$$f'(x) \approx \frac{3}{2}m_1 - \frac{3}{4}m_2 + \frac{1}{10}m_3 + O(dx^6)$$

$$m_1 = \frac{f(x+dx) - f(x-dx)}{2dx}$$

$$m_2 = \frac{f(x+2dx) - f(x-2dx)}{4dx}$$

$$m_3 = \frac{f(x+3dx) - f(x-3dx)}{6dx}$$

Newton Raphson: numerical derivative

```
namespace detail{

// from Christopher Kormanyos's "Real-time C++"
template <class T, class Function>
T derivative(const T& x, const T& dx, Function function)
{
  const T dx2(dx * 2U);
  const T dx3(dx * 3U);

  const T m1 ((function (x + dx) - function(x - dx)) / 2U);
  const T m2 ((function (x + dx2) - function(x - dx2)) / 4U);
  const T m3 ((function (x + dx3) - function(x - dx3)) / 6U);
  const T fifteen_m1 (m1 * 15U);
  const T six_m2 (m2 * 6U);
  const T ten_dx (dx * 10U);

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

} // detail
```

Newton Raphson: generic version

```
namespace detail{

template <class T, class Function>
T derivative(const T& x, const T& dx, Function function);

}



template <class Function, class T>
T find_root(Function function, const T& guess, const T& min, const T& max)
{
  static const auto get_tuple = [&function](const T& x)
  {
    const T dx = x != 0 ?  T(x * 0.05) : T(0.05);
    const T val = function(x);
    const T derivative_val = detail::derivative(x, dx, function);

    return std::make_tuple(val, derivative_val);
  };

  constexpr size_t digits = std::numeric_limits<T>::digits;
  return boost::math::tools::newton_raphson_iterate(get_tuple, guess, min, max, digits);
}
```

Newton Raphson: testing

```
template <class Function, class T>
T find_root(Function function, const T& guess, const T& min, const T& max);




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

  const auto test_fun_1 = [](const double& x){ return x * x * x - 27; };
  std::cout << find_root(test_fun_1, 2., 0., 5.) << std::endl;          // 3

  const auto test_fun_2 = [](const double& x){ return std::cos(x) - 2 * x; };
  std::cout << find_root(test_fun_2, 2., 0., 5.) << std::endl;          // 0.450183611294874

  const auto test_fun_3 = [](const double& x){ return std::cos(x / 2); };
  std::cout << find_root(test_fun_3, 2., 0., 5.) << std::endl;          // 3.14159265358979

  return 0;
}
```

Thank you for your attention!