Autodi

Automatic Di erentiation C++ Library

Matthew Pulver

June 22, 2019

Contents

1 Description

Autodiff is a header-only C++ library that facilitates the automatic differentiation (forward mode) of mathematical functions of single and multiple variables.

This implementation is based upon the Taylor series expansion of an analytic function f at the point x_0:

$$\begin{aligned} f(x_0+\varepsilon) &= f(x_0) + f'(x_0)\varepsilon + \frac{f''(x_0)}{2!}\varepsilon^2 + \frac{f'''(x_0)}{3!}\varepsilon^3 + \cdots \\ &= \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\varepsilon^n + O\left(\varepsilon^{N+1}\right). \end{aligned}$$

The essential idea of autodiff is the substitution of numbers with polynomials in the evaluation of $f(x_0)$. By substituting the number x_0 with the first-order polynomial $x_0+\varepsilon$, and using the same algorithm to compute $f(x_0+\varepsilon)$, the resulting polynomial in ε contains the function's derivatives $f'(x_0)$, $f''(x_0)$, $f'''(x_0)$, ... within the coefficients. Each coefficient is equal to the derivative of its respective order, divided by the factorial of the order.

In greater detail, assume one is interested in calculating the first N derivatives of f at x_0. Without loss of precision to the calculation of the derivatives, all terms $O\left(\varepsilon^{N+1}\right)$ that include powers of ε greater than N can be discarded. (This is due to the fact that each term in a polynomial depends only upon equal and lower-order terms under arithmetic operations.) Under these truncation rules, f provides a polynomial-to-polynomial transformation:

$$f \quad : \quad x_0+\varepsilon \quad \mapsto \quad \sum_{n=0}^{N} y_n\varepsilon^n = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}\varepsilon^n.$$

C++'s ability to overload operators and functions allows for the creation of a class `fvar` (forward-mode autodiff variable) that represents polynomials in ε. Thus the same algorithm f that calculates the numeric value of $y_0 = f(x_0)$, when written to accept and return variables of a generic (template) type, is also used to calculate the polynomial $\sum_{n=0}^{N} y_n\varepsilon^n = f(x_0+\varepsilon)$. The derivatives $f^{(n)}(x_0)$ are then found from the product of the respective factorial $n!$ and coefficient y_n:

$$\frac{d^n f}{dx^n}(x_0) = n!y_n.$$

2 Examples

2.1 Example 1: Single-variable derivatives

2.1.1 Calculate derivatives of $f(x) = x^4$ at $x = 2$.

In this example, `make_fvar<double, Order>(2.0)` instantiates the polynomial $2+\varepsilon$. The `Order=5` means that enough space is allocated (on the stack) to hold a polynomial of up to degree 5 during the proceeding computation.

Internally, this is modeled by a `std::array<double,6>` whose elements {`2, 1, 0, 0, 0, 0`} correspond to the 6 coefficients of the polynomial upon initialization. Its fourth power, at the end of the computation, is a polynomial with coefficients `y = {16, 32, 24, 8, 1, 0}`. The derivatives are obtained using the formula $f^{(n)}(2) = n! * \mathtt{y[n]}$.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>

template <typename T>
T fourth_power(T const& x) {
  T x4 = x * x;  // retval in operator*() uses x4's memory via NRVO.
  x4 *= x4;      // No copies of x4 are made within operator*=() even when squaring.
  return x4;     // x4 uses y's memory in main() via NRVO.
}

int main() {
  using namespace boost::math::differentiation;
```

```
  constexpr unsigned Order = 5;                    // Highest order derivative to be calculated.
  auto const x = make_fvar<double, Order>(2.0);  // Find derivatives at x=2.
  auto const y = fourth_power(x);
  for (unsigned i = 0; i <= Order; ++i)
    std::cout << "y.derivative(" << i << ") = " << y.derivative(i) << std::endl;
  return 0;
}
/*
Output:
y.derivative(0) = 16
y.derivative(1) = 32
y.derivative(2) = 48
y.derivative(3) = 48
y.derivative(4) = 24
y.derivative(5) = 0
*/
```

The above calculates

$$\begin{aligned}
\texttt{y.derivative(0)} = \quad f(2) &= \quad x^4\big|_{x=2} = 16 \\
\texttt{y.derivative(1)} = \quad f'(2) &= \quad 4\cdot x^3\big|_{x=2} = 32 \\
\texttt{y.derivative(2)} = \quad f''(2) &= \quad 4\cdot 3\cdot x^2\big|_{x=2} = 48 \\
\texttt{y.derivative(3)} = f'''(2) &= 4\cdot 3\cdot 2\cdot x\big|_{x=2} = 48 \\
\texttt{y.derivative(4)} = f^{(4)}(2) &= \quad 4\cdot 3\cdot 2\cdot 1 = 24 \\
\texttt{y.derivative(5)} = f^{(5)}(2) &= \quad 0
\end{aligned}$$

2.2 Example 2: Multi-variable mixed partial derivatives with multi-precision data type

2.2.1 Calculate $\frac{\partial^{12} f}{\partial w^3 \partial x^2 \partial y^4 \partial z^3}(11,12,13,14)$ with a precision of about 50 decimal digits, where $f(w,x,y,z) = \exp\left(w\sin\left(\frac{x\log(y)}{z}\right) + \sqrt{\frac{wz}{xy}}\right) + \frac{w^2}{\tan(z)}$.

In this example, `make_ftuple<float50, Nw, Nx, Ny, Nz>(11, 12, 13, 14)` returns a `std::tuple` of 4 independent `fvar` variables, with values of 11, 12, 13, and 14, for which the maximum order derivative to be calculated for each are 3, 2, 4, 3, respectively. The order of the variables is important, as it is the same order used when calling `v.derivative(Nw, Nx, Ny, Nz)` in the example below.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <boost/multiprecision/cpp_bin_float.hpp>
#include <iostream>

using namespace boost::math::differentiation;

template <typename W, typename X, typename Y, typename Z>
promote<W, X, Y, Z> f(const W& w, const X& x, const Y& y, const Z& z) {
  using namespace std;
  return exp(w * sin(x * log(y) / z) + sqrt(w * z / (x * y))) + w * w / tan(z);
}

int main() {
  using float50 = boost::multiprecision::cpp_bin_float_50;

  constexpr unsigned Nw = 3;  // Max order of derivative to calculate for w
  constexpr unsigned Nx = 2;  // Max order of derivative to calculate for x
```

```
  constexpr unsigned Ny = 4;  // Max order of derivative to calculate for y
  constexpr unsigned Nz = 3;  // Max order of derivative to calculate for z
  // Declare 4 independent variables together into a std::tuple.
  auto const variables = make_ftuple<float50, Nw, Nx, Ny, Nz>(11, 12, 13, 14);
  auto const& w = std::get<0>(variables);  // Up to Nw derivatives at w=11
  auto const& x = std::get<1>(variables);  // Up to Nx derivatives at x=12
  auto const& y = std::get<2>(variables);  // Up to Ny derivatives at y=13
  auto const& z = std::get<3>(variables);  // Up to Nz derivatives at z=14
  auto const v = f(w, x, y, z);
  // Calculated from Mathematica symbolic differentiation.
  float50 const answer("1976.319600747797717779881875290418720908121189218755");
  std::cout << std::setprecision(std::numeric_limits<float50>::digits10)
            << "mathematica   : " << answer << '\n'
            << "autodiff      : " << v.derivative(Nw, Nx, Ny, Nz) << '\n'
            << std::setprecision(3)
            << "relative error: " << (v.derivative(Nw, Nx, Ny, Nz) / answer - 1) << '\n';
  return 0;
}
/*
Output:
mathematica   : 1976.319600747797717779881875290418720908121189218755
autodiff      : 1976.319600747797717779881875290418720908121189218755
relative error: 2.67e-50
*/
```

2.3 Example 3: Black-Scholes Option Pricing with Greeks Automatically Calculated

2.3.1 Calculate greeks directly from the Black-Scholes pricing function.

Below is the standard Black-Scholes pricing function written as a function template, where the price, volatility (sigma), time to expiration (tau) and interest rate are template parameters. This means that any Greek based on these 4 variables can be calculated using autodiff. The below example calculates delta and gamma where the variable of differentiation is only the price. For examples of more exotic greeks, see `example/black_scholes.cpp`.

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>

using namespace boost::math::constants;
using namespace boost::math::differentiation;

// Equations and function/variable names are from
// https://en.wikipedia.org/wiki/Greeks_(finance)#Formulas_for_European_option_Greeks

// Standard normal cumulative distribution function
template <typename X>
X Phi(X const& x) {
  return 0.5 * erfc(-one_div_root_two<X>() * x);
}

enum class CP { call, put };

// Assume zero annual dividend yield (q=0).
template <typename Price, typename Sigma, typename Tau, typename Rate>
promote<Price, Sigma, Tau, Rate> black_scholes_option_price(CP cp,
                                                            double K,
                                                            Price const& S,
                                                            Sigma const& sigma,
                                                            Tau const& tau,
```

```
                                                        Rate const& r) {
  using namespace std;
  auto const d1 = (log(S / K) + (r + sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
  auto const d2 = (log(S / K) + (r - sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
  switch (cp) {
    case CP::call:
      return S * Phi(d1) - exp(-r * tau) * K * Phi(d2);
    case CP::put:
      return exp(-r * tau) * K * Phi(-d2) - S * Phi(-d1);
  }
}

int main() {
  double const K = 100.0;                    // Strike price.
  auto const S = make_fvar<double, 2>(105);  // Stock price.
  double const sigma = 5;                    // Volatility.
  double const tau = 30.0 / 365;             // Time to expiration in years. (30 days).
  double const r = 1.25 / 100;               // Interest rate.
  auto const call_price = black_scholes_option_price(CP::call, K, S, sigma, tau, r);
  auto const put_price = black_scholes_option_price(CP::put, K, S, sigma, tau, r);

  std::cout << "black-scholes call price = " << call_price.derivative(0) << '\n'
            << "black-scholes put  price = " << put_price.derivative(0) << '\n'
            << "call delta = " << call_price.derivative(1) << '\n'
            << "put  delta = " << put_price.derivative(1) << '\n'
            << "call gamma = " << call_price.derivative(2) << '\n'
            << "put  gamma = " << put_price.derivative(2) << '\n';
  return 0;
}
/*
Output:
black-scholes call price = 56.5136
black-scholes put  price = 51.4109
call delta = 0.773818
put  delta = -0.226182
call gamma = 0.00199852
put  gamma = 0.00199852
*/
```

3 Advantages of Automatic Differentiation

The above examples illustrate some of the advantages of using autodiff:

- Elimination of code redundancy. The existence of N separate functions to calculate derivatives is a form of code redundancy, with all the liabilities that come with it:
  - Changes to one function require N additional changes to other functions. In the 3^{rd} example above, consider how much larger and inter-dependent the above code base would be if a separate function were written for each Greek value.
  - Dependencies upon a derivative function for a different purpose will break when changes are made to the original function. What doesn't need to exist cannot break.
  - Code bloat, reducing conceptual integrity. Control over the evolution of code is easier/safer when the code base is smaller and able to be intuitively grasped.
- Accuracy of derivatives over finite difference methods. Single-iteration finite difference methods always include a Δx free variable that must be carefully chosen for each application. If Δx is too small, then numerical errors become large. If Δx is too large, then mathematical errors become large. With autodiff, there are no free

variables to set and the accuracy of the answer is generally superior to finite difference methods even with the best choice of Δx.

4 Mathematics

In order for the usage of the autodiff library to make sense, a basic understanding of the mathematics will help.

4.1 Truncated Taylor Series

4.2 Arithmetic

What was essentially done above was to take a formula/algorithm for calculating $f(x_0)$ from a number x_0, and instead apply the same formula/algorithm to a polynomial $x_0 + \varepsilon$. Intermediate steps operate on values of the form

$$\mathbf{x} = x_0 + x_1\varepsilon + x_2\varepsilon^2 + \cdots + x_N\varepsilon^N$$

and the final return value is of this polynomial form as well. In other words, the normal arithmetic operators $+, -, \times, \div$ applied to numbers x are instead applied to polynomials $\mathbf{x}$. Through the overloading of C++ operators and functions, floating point data types are replaced with data types that represent these polynomials. More specifically, C++ types such as `double` are replaced with `std::array<double,N+1>`, which hold the above $N + 1$ coefficients x_i, and are wrapped in a `class` that overloads all of the arithmetic operators.

The logic of these arithmetic operators simply mirror that which is applied to polynomials. We'll look at each of the 4 arithmetic operators in detail.

4.2.1 Addition

The addition of polynomials $\mathbf{x}$ and $\mathbf{y}$ is done component-wise:

$$\begin{aligned}
\mathbf{z} &= \mathbf{x} + \mathbf{y} \\
&= \left(\sum_{i=0}^{N} x_i\varepsilon^i\right) + \left(\sum_{i=0}^{N} y_i\varepsilon^i\right) \\
&= \sum_{i=0}^{N} (x_i + y_i)\varepsilon^i \\
z_i &= x_i + y_i \qquad \text{for } i \in \{0, 1, 2, ..., N\}.
\end{aligned}$$

4.2.2 Subtraction

Subtraction follows the same form as addition:

$$\begin{aligned}
\mathbf{z} &= \mathbf{x} - \mathbf{y} \\
&= \left(\sum_{i=0}^{N} x_i\varepsilon^i\right) - \left(\sum_{i=0}^{N} y_i\varepsilon^i\right) \\
&= \sum_{i=0}^{N} (x_i - y_i)\varepsilon^i \\
z_i &= x_i - y_i \qquad \text{for } i \in \{0, 1, 2, ..., N\}.
\end{aligned}$$

4.2.3 Multiplication

Multiplication produces higher-order terms:

$$
\begin{aligned}
\mathbf{z} &= \mathbf{x} \times \mathbf{y} \\
&= \left(\sum_{i=0}^{N} x_i \varepsilon^i \right) \left(\sum_{i=0}^{N} y_i \varepsilon^i \right) \\
&= x_0 y_0 + (x_0 y_1 + x_1 y_0)\varepsilon + (x_0 y_2 + x_1 y_1 + x_2 y_0)\varepsilon^2 + \cdots + \left(\sum_{j=0}^{N} x_j y_{N-j} \right) \varepsilon^N + O\left(\varepsilon^{N+1}\right) \\
&= \sum_{i=0}^{N} \sum_{j=0}^{i} x_j y_{i-j} \varepsilon^i + O\left(\varepsilon^{N+1}\right) \\
z_i &= \sum_{j=0}^{i} x_j y_{i-j} \qquad \text{for } i \in \{0, 1, 2, ..., N\}.
\end{aligned}
$$

In the case of multiplication, terms involving powers of ε greater than N, collectively denoted by $O\left(\varepsilon^{N+1}\right)$, are simply discarded. Fortunately, the values of z_i for $i \leq N$ do not depend on any of these discarded terms, so there is no loss of precision in the final answer. The only information that is lost are the values of higher order derivatives, which we are not interested in anyway. If we were, then we would have simply chosen a larger value of N to begin with.

4.2.4 Division

Division is not directly calculated as are the others. Instead, to find the components of $\mathbf{z} = \mathbf{x} \div \mathbf{y}$ we require that $\mathbf{x} = \mathbf{y} \times \mathbf{z}$. This yields a recursive formula for the components z_i:

$$
\begin{aligned}
x_i &= \sum_{j=0}^{i} y_j z_{i-j} \\
&= y_0 z_i + \sum_{j=1}^{i} y_j z_{i-j} \\
z_i &= \frac{1}{y_0} \left(x_i - \sum_{j=1}^{i} y_j z_{i-j} \right) \qquad \text{for } i \in \{0, 1, 2, ..., N\}.
\end{aligned}
$$

In the case of division, the values for z_i must be calculated sequentially, since z_i depends on the previously calculated values $z_0, z_1, ..., z_{i-1}$.

4.3 General Functions

Calling standard mathematical functions such as `log()`, `cos()`, etc. should return accurate higher order derivatives. For example, `exp(x)` may be written internally as a specific 14^{th}-degree polynomial to approximate e^x when $0 < x < 1$. This would mean that the 15^{th} derivative, and all higher order derivatives, would be 0, however we know that $\frac{d^{15}}{dx^{15}} e^x = e^x$. How should such functions whose derivatives are known be written to provide accurate higher order derivatives? The answer again comes back to the function's Taylor series.

To simplify notation, for a given polynomial $\mathbf{x} = x_0 + x_1\varepsilon + x_2\varepsilon^2 + \cdots + x_N\varepsilon^N$ define

$$
\mathbf{x}_\varepsilon = x_1\varepsilon + x_2\varepsilon^2 + \cdots + x_N\varepsilon^N = \sum_{i=1}^{N} x_i \varepsilon^i.
$$

This allows for a concise expression of a general function f of $\mathbf{x}$:

$$
\begin{aligned}
f(\mathbf{x}) &= f(x_0 + \mathbf{x}_\varepsilon) \\
&= f(x_0) + f'(x_0)\mathbf{x}_\varepsilon + \frac{f''(x_0)}{2!}\mathbf{x}_\varepsilon^2 + \frac{f'''(x_0)}{3!}\mathbf{x}_\varepsilon^3 + \cdots + \frac{f^{(N)}(x_0)}{N!}\mathbf{x}_\varepsilon^N + O\left(\varepsilon^{N+1}\right) \\
&= \sum_{i=0}^{N} \frac{f^{(i)}(x_0)}{i!}\mathbf{x}_\varepsilon^i + O\left(\varepsilon^{N+1}\right)
\end{aligned}
$$

where ε has been substituted with $\mathbf{x}_\varepsilon$ in the ε-taylor series for $f(x)$. This form gives a recipe for calculating $f(\mathbf{x})$ in general from regular numeric calculations $f(x_0)$, $f'(x_0)$, $f''(x_0)$, ... and successive powers of the epsilon terms $\mathbf{x}_\varepsilon$.

For an application in which we are interested in up to N derivatives in x the data structure to hold this information is an $(N+1)$-element array `v` whose general element is

$$
\texttt{v[i]} = \frac{f^{(i)}(x_0)}{i!} \qquad \text{for } i \in \{0, 1, 2, ..., N\}.
$$

4.4 Multiple Variables

In C++, the generalization to mixed partial derivatives with multiple independent variables is conveniently achieved with recursion. To begin to see the recursive pattern, consider a two-variable function $f(x, y)$. Since x and y are independent, they require their own independent epsilons ε_x and ε_y, respectively.

Expand $f(x, y)$ for $x = x_0 + \varepsilon_x$:

$$
\begin{aligned}
f(x_0 + \varepsilon_x, y) &= f(x_0, y) + \frac{\partial f}{\partial x}(x_0, y)\varepsilon_x + \frac{1}{2!}\frac{\partial^2 f}{\partial x^2}(x_0, y)\varepsilon_x^2 + \frac{1}{3!}\frac{\partial^3 f}{\partial x^3}(x_0, y)\varepsilon_x^3 + \cdots + \frac{1}{M!}\frac{\partial^M f}{\partial x^M}(x_0, y)\varepsilon_x^M + O\left(\varepsilon_x^{M+1}\right) \\
&= \sum_{i=0}^{M} \frac{1}{i!}\frac{\partial^i f}{\partial x^i}(x_0, y)\varepsilon_x^i + O\left(\varepsilon_x^{M+1}\right).
\end{aligned}
$$

Next, expand $f(x_0 + \varepsilon_x, y)$ for $y = y_0 + \varepsilon_y$:

$$
\begin{aligned}
f(x_0 + \varepsilon_x, y_0 + \varepsilon_y) &= \sum_{j=0}^{N} \frac{1}{j!}\frac{\partial^j}{\partial y^j}\left(\sum_{i=0}^{M} \varepsilon_x^i \frac{1}{i!}\frac{\partial^i f}{\partial x^i}\right)(x_0, y_0)\varepsilon_y^j + O\left(\varepsilon_x^{M+1}\right) + O\left(\varepsilon_y^{N+1}\right) \\
&= \sum_{i=0}^{M}\sum_{j=0}^{N} \frac{1}{i!j!}\frac{\partial^{i+j} f}{\partial x^i \partial y^j}(x_0, y_0)\varepsilon_x^i\varepsilon_y^j + O\left(\varepsilon_x^{M+1}\right) + O\left(\varepsilon_y^{N+1}\right).
\end{aligned}
$$

Similar to the single-variable case, for an application in which we are interested in up to M derivatives in x and N derivatives in y, the data structure to hold this information is an $(M+1) \times (N+1)$ array `v` whose element at (i, j) is

$$
\texttt{v[i][j]} = \frac{1}{i!j!}\frac{\partial^{i+j} f}{\partial x^i \partial y^j}(x_0, y_0) \qquad \text{for } (i, j) \in \{0, 1, 2, ..., M\} \times \{0, 1, 2, ..., N\}.
$$

The generalization to additional independent variables follows the same pattern.

4.4.1 Declaring Multiple Variables

Internally, independent variables are represented by vectors within orthogonal vector spaces. Because of this, one must be careful when declaring more than one independent variable so that they do not end up in parallel vector spaces. This can easily be achieved by following one rule:

- When declaring more than one independent variable, call `make_ftuple<>()` once and only once.

The tuple of values returned are independent. Though it is possible to achieve the same result with multiple calls to `make_fvar`, this is an easier and less error-prone method. See Section 2.2 for example usage.

5 Writing Functions for Autodiff Compatibility

In this section, a general procedure is given for writing new, and transforming existing, C++ mathematical functions for compatibility with autodiff.

There are 3 categories of functions that require different strategies:

1. Piecewise-rational functions. These are simply piecewise quotients of polynomials. All that is needed is to turn the function parameters and return value into generic (template) types. This will then allow the function to accept and return autodiff's `fvar` types, thereby using autodiff's overloaded arithmetic operators which calculate the derivatives automatically.

2. Functions that call existing autodiff functions. This is the same as the previous, but may also include calls to functions that are in the autodiff library. Examples: `exp()`, `log()`, `tgamma()`, etc.

3. New functions for which the derivatives can be calculated. This is the most general technique, as it allows for the development of a function which do not fall into the previous two categories.

Functions written in any of these ways may then be added to the autodiff library.

5.1 Piecewise-Rational Functions

$$f(x) = \frac{1}{1+x^2}$$

By simply writing this as a template function, autodiff can calculate derivatives for it:

```
#include <boost/math/differentiation/autodiff.hpp>
#include <iostream>

template <typename T>
T rational(T const& x) {
  return 1 / (1 + x * x);
}

int main() {
  using namespace boost::math::differentiation;
  auto const x = make_fvar<double, 10>(0);
  auto const y = rational(x);
  std::cout << std::setprecision(std::numeric_limits<double>::digits10)
            << "y.derivative(10) = " << y.derivative(10) << std::endl;
  return 0;
}
/*
Output:
y.derivative(10) = -3628800
*/
```

As simple as $f(x)$ may seem, the derivatives can get increasingly complex as derivatives are taken. For example, the 10^{th} derivative has the form

$$f^{(10)}(x) = -3628800\frac{1 - 55x^2 + 330x^4 - 462x^6 + 165x^8 - 11x^{10}}{(1+x^2)^{11}}.$$

Derivatives of $f(x)$ are useful, and in fact used, in calculating higher order derivatives for $\arctan(x)$ for instance, since

$$\arctan^{(n)}(x) = \left(\frac{d}{dx}\right)^{n-1}\frac{1}{1+x^2} \qquad \text{for} \quad 1 \le n.$$

5.2 Functions That Call Existing Autodiff Functions

Many of the standard library math function are overloaded in autodiff. It is recommended to use argument-dependent lookup (ADL) in order for functions to be written in a way that is general enough to accommodate standard types (`double`) as well as autodiff types (`fvar`).
Example:

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

using namespace boost::math::constants;

// Standard normal cumulative distribution function
template <typename T>
T Phi(T const& x)
{
  return 0.5 * std::erfc(-one_div_root_two<T>() * x);
}
```

Though `Phi(x)` is general enough to handle the various fundamental floating point types, this will not work if `x` is an autodiff `fvar` variable, since `std::erfc` does not include a specialization for `fvar`. The recommended solution is to remove the namespace prefix `std::` from `erfc`:

```
#include <boost/math/constants/constants.hpp>
#include <boost/math/differentiation/autodiff.hpp>
#include <cmath>

using namespace boost::math::constants;

// Standard normal cumulative distribution function
template <typename T>
T Phi(T const& x)
{
  using std::erfc;
  return 0.5 * erfc(-one_div_root_two<T>() * x);
}
```

In this form, when `x` is of type `fvar`, the C++ compiler will search for and find a function `erfc` within the same namespace as `fvar`, which is in the autodiff library, via ADL. Because of the using-declaration, it will also call `std::erfc` when `x` is a fundamental type such as `double`.

5.3 New Functions For Which The Derivatives Can Be Calculated

Mathematical functions which do not fall into the previous two categories can be constructed using autodiff helper functions. This requires a separate function for calculating the derivatives. In case you are asking yourself what good is an autodiff library if one needs to supply the derivatives, the answer is that the new function will fit in with the rest of the autodiff library, thereby allowing for the creation of additional functions via all of the arithmetic operators, plus function composition, which was not readily available without the library.

The example given here is for `cos`:

```
template <typename RealType, size_t Order>
fvar<RealType, Order> cos(fvar<RealType, Order> const& cr) {
  using std::cos;
  using std::sin;
  using root_type = typename fvar<RealType, Order>::root_type;
  constexpr size_t order = fvar<RealType, Order>::order_sum;
  root_type const d0 = cos(static_cast<root_type>(cr));
  if constexpr (order == 0)
    return fvar<RealType, Order>(d0);
  else {
```

```
      root_type const d1 = -sin(static_cast<root_type>(cr));
      root_type const derivatives[4]{d0, d1, -d0, -d1};
      return cr.apply_derivatives(order,
                                  [&derivatives](size_t i) { return derivatives[i & 3]; });
    }
  }
```

This uses the helper function `fvar::apply_derivatives` which takes two parameters:

1. The highest order derivative to be calculated.
2. A function that maps derivative order to derivative value.

The highest order derivative necessary to be calculated is generally equal to `fvar::order_sum`. In the case of `sin` and `cos`, the derivatives are cyclical with period 4. Thus it is sufficient to store only these 4 values into an array, and take the derivative order modulo 4 as the index into this array.

A second helper function, not shown here, is `apply_coefficients`. This is used the same as `apply_derivatives` except that the supplied function calculates coefficients instead of derivatives. The relationship between a coefficient C_n and derivative D_n for derivative order n is

$$C_n = \frac{D_n}{n!}.$$

Internally, `fvar` holds coefficients rather than derivatives, so in case the coefficient values are more readily available than the derivatives, it can save some unnecessary computation to use `apply_coefficients`. See the definition of `atan` for an example.

Both of these helper functions use Horner's method when calculating the resulting polynomial `fvar`. This works well when the derivatives are finite, but in cases where derivatives are infinite, this can quickly result in NaN values as the computation progresses. In these cases, one can call non-Horner versions of both function which better "isolate" infinite values so that they are less likely to evolve into NaN values.

The four helper functions available for constructing new autodiff functions from known coefficients/derivatives are:

1. `fvar::apply_coefficients`
2. `fvar::apply_coefficients_nonhorner`
3. `fvar::apply_derivatives`
4. `fvar::apply_derivatives_nonhorner`

6 Function Writing Guidelines

At a high level there is one fairly simple principle, loosely and intuitively speaking, to writing functions for which autodiff can effectively calculate derivatives:

Autodiff Function Principle (AFP)

> A function whose branches in logic correspond to piecewise analytic calculations over non-singleton intervals, with smooth transitions between the intervals, and is free of indeterminate forms in the calculated value and higher order derivatives, will work fine with autodiff.

Stating this with greater mathematical rigor can be done. However what seems to be more practical, in this case, is to give examples and categories of examples of what works, what doesn't, and how to remedy some of the common problems that may be encountered. That is the approach taken here.

6.1 Example 1: $f(x) = \max(0, x)$

One potential implementation of $f(x) = \max(0, x)$ is:

```
template<typename T>
T f(const T& x)
{
    return 0 < x ? x : 0;
}
```

Though this is consistent with Section 5, there are two problems with it:

1. `f(nan) = 0`. This problem is independent of autodiff, but is worth addressing anyway. If there is an indeterminate form that arises within a calculation and is input into f, then it gets "covered up" by this implementation leading to an unknowingly incorrect result. Better for functions in general to propagate NaN values, so that the user knows something went wrong and doesn't rely on an incorrect result, and likewise the developer can track down where the NaN originated from and remedy it.

2. $f'(0) = 0$ when autodiff is applied. This is because `f` returns 0 as a constant when `x==0`, wiping out any of the derivatives (or sensitivities) that `x` was holding as an autodiff variable. Instead, let us apply the AFP and identify the two intervals over which f is defined: $(-\infty, 0] \cup (0, \infty)$. Though the function itself is not analytic at $x = 0$, we can attempt somewhat to smooth out this transition point by averaging the calculation of $f(x)$ at $x = 0$ from each interval. If $x < 0$ then the result is simply 0, and if $0 < x$ then the result is x. The average is $\frac{1}{2}(0 + x)$ which will allow autodiff to calculate $f'(0) = \frac{1}{2}$. This is a more reasonable answer.

A better implementation that resolves both issues is:

```
template<typename T>
T f(const T& x)
{
    if (x < 0)
        return 0;
    else if (x == 0)
        return 0.5*x;
    else
        return x;
}
```

6.2 Example 2: $f(x) = \mathrm{sinc}(x)$

The definition of $\mathrm{sinc} : \mathbb{R} \to \mathbb{R}$ is

$$\mathrm{sinc}(x) = \begin{cases} 1 & \text{if } x = 0 \\ \frac{\sin(x)}{x} & \text{otherwise.} \end{cases}$$

A potential implementation is:

```
template<typename T>
T sinc(const T& x)
{
    using std::sin;
    return x == 0 ? 1 : sin(x) / x;
}
```

Though this is again consistent with Section 5, and returns correct non-derivative values, it returns a constant when `x==0` thereby losing all derivative information contained in `x` and contributions from sinc. For example, $\mathrm{sinc}''(0) = -\frac{1}{3}$, however `y.derivative(2) == 0` when `y = sinc(make_fvar<double,2>(0))` using the above incorrect implementation. Applying the AFP, the intervals upon which separate branches of logic are applied are $(-\infty, 0) \cup [0, 0] \cup (0, \infty)$. The violation occurs due to the singleton interval $[0, 0]$, even though a constant function of 1 is technically analytic. The remedy is to define a custom sinc overload and add it to the autodiff library. This has been done. Mathematically, it is well-defined and free of indeterminate forms, as is the 3^{rd} expression in the equalities

$$\frac{1}{x}\sin(x) = \frac{1}{x}\sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)!}x^{2n+1} = \sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)!}x^{2n}.$$

The autodiff library contains helper functions to help write function overloads when the derivatives of a function are known. This is an advanced feature and documentation for this may be added at a later time.

For now, it is worth understanding the ways in which indeterminate forms can occur within a mathematical calculation, and avoid them when possible by rewriting the function. Table 1 compares 3 types of indeterminate forms. Assume the product `a*b` is a positive finite value.

	$f(x) = \left(\frac{a}{x}\right) \times (bx^2)$	$g(x) = \left(\frac{a}{x}\right) \times (bx)$	$h(x) = \left(\frac{a}{x^2}\right) \times (bx)$
Mathematical Limit	$\lim_{x\to 0} f(x) = 0$	$\lim_{x\to 0} g(x) = ab$	$\lim_{x\to 0} h(x) = \infty$
Floating Point Arithmetic	`f(0) = inf*0 = nan`	`g(0) = inf*0 = nan`	`h(0) = inf*0 = nan`

Table 1: Automatic differentiation does not compute limits. Indeterminate forms must be simplified manually. (These cases are not meant to be exhaustive.)

Indeterminate forms result in NaN values within a calculation. Mathematically, if they occur at locally isolated points, then we generally prefer the mathematical limit as the result, even if it is infinite. As demonstrated in Table 1, depending upon the nature of the indeterminate form, the mathematical limit can be 0 (no matter the values of a or b), or ab, or ∞, but these 3 cases cannot be distinguished by the floating point result of nan. Floating point arithmetic does not perform limits (directly), and neither does the autodiff library. Thus it is up to the diligence of the developer to keep a watchful eye over where indeterminate forms can arise.

6.3 Example 3: $f(x) = \sqrt{x}$ and $f'(0) = \infty$

When working with functions that have infinite higher order derivatives, this can very quickly result in nans in higher order derivatives as the computation progresses, as `inf-inf`, `inf/inf`, and `0*inf` result in `nan`. See Table 2 for an example.

$f(x)$	$f(0)$	$f'(0)$	$f''(0)$	$f'''(0)$
`sqrt(x)`	`0`	`inf`	`-inf`	`inf`
`sqr(sqrt(x)+1)`	`1`	`inf`	`nan`	`nan`
`x+2*sqrt(x)+1`	`1`	`inf`	`-inf`	`inf`

Table 2: Indeterminate forms in higher order derivatives. `sqr(x) == x*x`.

Calling the autodiff-overloaded implementation of $f(x) = \sqrt{x}$ at the value `x==0` results in the 1st row (after the header row) of Table 2, as is mathematically correct. The 2nd row shows $f(x) = (\sqrt{x} + 1)^2$ resulting in `nan` values for $f''(0)$ and all higher order derivatives. This is due to the internal arithmetic in which `inf` is added to `-inf` during the squaring, resulting in a `nan` value for $f''(0)$ and all higher orders. This is typical of `inf` values in autodiff. Where they show up, they are correct, however they can quickly introduce `nan` values into the computation upon the addition of oppositely signed `inf` values, division by `inf`, or multiplication by `0`. It is worth noting that the infection of `nan` only spreads upward in the order of derivatives, since lower orders do not depend upon higher orders (which is also why dropping higher order terms in an autodiff computation does not result in any loss of precision for lower order terms.)

The resolution in this case is to manually perform the squaring in the computation, replacing the 2nd row with the 3rd: $f(x) = x + 2\sqrt{x} + 1$. Though mathematically equivalent, it allows autodiff to avoid `nan` values since $\sqrt{x}$ is more "isolated" in the computation. That is, the `inf` values that unavoidably show up in the derivatives of `sqrt(x)` for `x==0` do not have the chance to interact with other `inf` values as with the squaring.

6.4 Summary

The AFP gives a high-level unified guiding principle for writing C++ template functions that autodiff can effectively evaluate derivatives for.

Examples have been given to illustrate some common items to avoid doing:

1. It is not enough for functions to be piecewise continuous. On boundary points between intervals, consider returning the average expression of both intervals, rather than just one of them. Example: $\max(0, x)$ at $x = 0$.

In cases where the limits from both sides must match, and they do not, then `nan` may be a more appropriate value depending on the application.

2. Avoid returning individual constant values (e.g. $\mathrm{sinc}(0) = 1$.) Values must be computed uniformly along with other values in its local interval. If that is not possible, then the function must be overloaded to compute the derivatives precisely using the helper functions from Section 5.3.

3. Avoid intermediate indeterminate values in both the value ($\mathrm{sinc}(x)$ at $x = 0$) and derivatives ($(\sqrt{x} + 1)^2$ at $x = 0$). Try to isolate expressions that may contain infinite values/derivatives so that they do not introduce NaN values into the computation.

7 Acknowledgments

- Kedar Bhat — C++11 compatibility, Boost Special Functions compatibility testing, codecov integration, and feedback.
- Nick Thompson — Initial feedback and help with Boost integration.
- John Maddock — Initial feedback and help with Boost integration.

References

[1] `https://en.wikipedia.org/wiki/Automatic_differentiation`

[2] Andreas Griewank, Andrea Walther. *Evaluating Derivatives*. SIAM, 2nd ed. 2008.