Autodiff
Automatic Differentiation C++ Library

Matthew Pulver

February 3, 2019

1 Synopsis

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

namespace boost { namespace math { namespace differentiation {

// Type for variables and constants.
template<typename RealType, size_t Order, size_t... Orders>
using autodiff_fvar = typename detail::nest_fvar<RealType,Order,Orders...>::type;

// Function returning a variable of differentiation.
template<typename RealType, size_t Order, size_t... Orders>
autodiff_fvar<RealType,Order,Orders...> make_fvar(const RealType& ca);

// Type of combined autodiff types.
template<typename RealType, typename... RealTypes>
using promote = typename detail::promote_args_n<RealType,RealTypes...>::type;

namespace detail {

// Single autodiff variable. Independent variables are created by nesting.
template<typename RealType, size_t Order>
class fvar
{
  public:

    // Query return value of function to get the derivatives.
    template<typename... Orders>
    get_type_at<RealType, sizeof...(Orders)-1> derivative(Orders... orders) const;

    // All of the arithmetic and comparison operators are overloaded.
    template<typename RealType2, size_t Order2>
    fvar& operator+=(const fvar<RealType2,Order2>&);

    fvar& operator+=(const root_type&);

    // ...
};

// Standard math functions are overloaded and called via argument-dependent lookup (ADL).
template<typename RealType, size_t Order>
fvar<RealType,Order> floor(const fvar<RealType,Order>&);

template<typename RealType, size_t Order>
fvar<RealType,Order> exp(const fvar<RealType,Order>&);

// ...

} // namespace detail

} } } // namespace boost::math::differentiation
```

2 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 replacement of numbers with polynomials in the evaluation of f. By inputting the first-order polynomial $x_0+\varepsilon$, the resulting polynomial in ε contains the function's derivatives within the coefficients. Each coefficient is equal to a derivative of its respective order, divided by the factorial of the order.

Assume one is interested in the first N derivatives of f at x_0. Then without any 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, and under these truncation rules, f provides a polynomial-to-polynomial transformation:

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

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

$$f^{(n)}(x_0) = n!y_n.$$

3 Examples

3.1 Example 1: Single-variable derivatives

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

In this example, `autodiff_fvar<double,5>` is a data type that can hold a polynomial of up to degree 5, and the `make_fvar<double,5> x(2.0)` represents the polynomial $2+\varepsilon$. 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 is a polynomial with coefficients `y = `$\{$`16, 32, 24, 8, 1, 0`$\}$. The derivatives are obtained using the formula $f^{(n)}(2) = n! * \texttt{y[n]}$.

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

template<typename T>
T fourth_power(T x)
{
    x *= x;
    return x *= x;
}

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

    constexpr int Order=5; // The highest order derivative to be calculated.
    const autodiff_fvar<double,Order> x = make_fvar<double,Order>(2); //Find derivatives at x=2.
    const autodiff_fvar<double,Order> y = fourth_power(x);
    for (int 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

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

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

3.2.1 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 100 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, the data type `autodiff_fvar<cpp_dec_float_100,Nw,Nx,Ny,Nz>` represents a multivariate polynomial in 4 independent variables, where the highest powers of each are `Nw`, `Nx`, `Ny` and `Nz`. The underlying arithmetic data type, referred to as `root_type`, is `boost::multiprecision::cpp_dec_float_100`. The internal data type is `std::array<std::array<std::array<std::array<cpp_dec_float_100,Nz+1>,Ny+1>,Nx+1>,Nw+1>`. In general, the `root_type` is always the first template parameter to `autodiff_fvar<>` followed by the maximum derivative order that is to be calculated for each independent variable.

When variables are initialized with `make_var<...>()`, the position of the last derivative order given in the template parameter pack determines which variable is taken to be independent. In other words, it determines which of the 4 different polynomial variables $\varepsilon_w, \varepsilon_x, \varepsilon_y$, or ε_z are to be added to the constant term:

$$\begin{aligned}
\texttt{make\_fvar<cpp\_dec\_float\_100,Nw>(11)} &= 11 + \varepsilon_w\\
\texttt{make\_fvar<cpp\_dec\_float\_100,0,Nx>(12)} &= 12 + \varepsilon_x\\
\texttt{make\_fvar<cpp\_dec\_float\_100,0,0,Ny>(13)} &= 13 + \varepsilon_y\\
\texttt{make\_fvar<cpp\_dec\_float\_100,0,0,0,Nz>(14)} &= 14 + \varepsilon_z
\end{aligned}$$

Instances of different types are automatically promoted to the smallest multi-variable type that accommodates both when they are arithmetically combined (added, subtracted, multiplied, divided.)

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

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

int main()
{
  using cpp_dec_float_100 = boost::multiprecision::cpp_dec_float_100;
  using namespace boost::math::differentiation;

  constexpr int Nw=3; // Max order of derivative to calculate for w
  constexpr int Nx=2; // Max order of derivative to calculate for x
  constexpr int Ny=4; // Max order of derivative to calculate for y
  constexpr int Nz=3; // Max order of derivative to calculate for z
  using var = autodiff_fvar<cpp_dec_float_100,Nw,Nx,Ny,Nz>;
  const var w = make_fvar<cpp_dec_float_100,Nw>(11);
  const var x = make_fvar<cpp_dec_float_100,0,Nx>(12);
  const var y = make_fvar<cpp_dec_float_100,0,0,Ny>(13);
  const var z = make_fvar<cpp_dec_float_100,0,0,0,Nz>(14);
  const var v = f(w,x,y,z);
  // Calculated from Mathematica symbolic differentiation. See multiprecision.nb for script.
  const cpp_dec_float_100 answer("1976.3196007477977177798818752904187209081211892187499076"
    "5825359511118457691105604218209405164232553l4");
  std::cout << std::setprecision(std::numeric_limits<cpp_dec_float_100>::digits10)
    << "mathematica   : " << answer << '\n'
    << "autodiff      : " << v.derivative(Nw,Nx,Ny,Nz) << '\n'
    << "relative error: " << std::setprecision(3) << (v.derivative(Nw,Nx,Ny,Nz)/answer-1)
    << std::endl;
  return 0;
}
```

The relative error between the calculated and actual value is about 6.47×10^{-99}.

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

Basic calculus courses teach that a real analytic function $f : D \to \mathbb{R}$ is one which can be expressed as a Taylor series at a point $x_0 \in D \subseteq \mathbb{R}$:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f'''(x_0)}{3!}(x - x_0)^3 + \cdots$$

One way of thinking about this form is that given the value of an analytic function $f(x_0)$ and its derivatives $f'(x_0), f''(x_0), f'''(x_0), \ldots$ evaluated at a point x_0, then the value of the function $f(x)$ can be obtained at any other point $x \in D$ using the above formula.

Let us make the substitution $x = x_0 + \varepsilon$ and rewrite the above equation to get:

$$f(x_0 + \varepsilon) = f(x_0) + f'(x_0)\varepsilon + \frac{f''(x_0)}{2!}\varepsilon^2 + \frac{f'''(x_0)}{3!}\varepsilon^3 + \cdots$$

Now consider ε as *an abstract algebraic entity that never acquires a numeric value*, much like one does in basic algebra with variables like x or y. For example, we can still manipulate entities like xy and $(1 + 2x + 3x^2)$ without having to assign specific numbers to them.

Using this formula, autodiff goes in the other direction. Given a general formula/algorithm for calculating $f(x_0 + \varepsilon)$, the derivatives are obtained from the coefficients of the powers of ε in the resulting computation. The general coefficient for ε^n is

$$\frac{f^{(n)}(x_0)}{n!}.$$

Thus to obtain $f^{(n)}(x_0)$, the coefficient of ε^n is multiplied by $n!$.

4.1.1 Example

Apply the above technique to calculate the derivatives of $f(x) = x^4$ at $x_0 = 2$.

The first step is to evaluate $f(x_0 + \varepsilon)$ and simply go through the calculation/algorithm, treating ε as an abstract algebraic entity:

$$\begin{aligned} f(x_0+\varepsilon) &= f(2+\varepsilon) \\ &= (2+\varepsilon)^4 \\ &= \left(4+4\varepsilon+\varepsilon^2\right)^2 \\ &= 16+32\varepsilon+24\varepsilon^2+8\varepsilon^3+\varepsilon^4. \end{aligned}$$

Equating the powers of ε from this result with the above ε-taylor expansion yields the following equalities:

$$f(2)=16, \qquad f'(2)=32, \qquad \frac{f''(2)}{2!}=24, \qquad \frac{f'''(2)}{3!}=8, \qquad \frac{f^{(4)}(2)}{4!}=1, \qquad \frac{f^{(5)}(2)}{5!}=0.$$

Multiplying both sides by the respective factorials gives

$$f(2)=16, \qquad f'(2)=32, \qquad f''(2)=48, \qquad f'''(2)=48, \qquad f^{(4)}(2)=24, \qquad f^{(5)}(2)=0.$$

These values can be directly confirmed by the power rule applied to $f(x) = x^4$.

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 a

rithmetic-addition Addition

Given polynomials $\mathbf{x}$ and $\mathbf{y}$, how is $\mathbf{z} = \mathbf{x} + \mathbf{y}$ calculated?

To answer this, one simply expands $\mathbf{x}$ and $\mathbf{y}$ into their polynomial forms and add them together:

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

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

When calling standard mathematical functions such as `log()`, `cos()`, etc. how should these be written in order to support autodiff variable types? That is, how should they be written to provide accurate derivatives?

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 $y = y_0 + \varepsilon_y$:

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

Similarly 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. This is made more concrete with C++ code in the next section.

5 Usage

5.1 Single Variable

To calculate derivatives of a single variable x, at a particular value x_0, the following must be specified at compile-time:

1. The numeric data type `T` of x_0. Examples: `double`, `boost::multiprecision::cpp_dec_float_100`, etc.

2. The maximum derivative order M that is to be calculated with respect to x.

Note that both of these requirements are entirely analogous to declaring and using a `std::array<T,N>`. `T` and `N` must be set as compile-time, but which elements in the array are accessed can be determined at run-time, just as the choice of what derivatives to query in autodiff can be made during run-time.

To declare and initialize x:

```
using namespace boost::math::differentiation;
autodiff_fvar<T,M> x = make_fvar<T,M>(x0);
```

where `x0` is a run-time value of type `T`. Assuming `0 < M`, this represents the polynomial $x_0+\varepsilon$. Internally, the member variable of type `std::array<T,M>` is `v = { x0, 1, 0, 0, ... }`, consistent with the above mathematical treatise.

To find the derivatives $f^{(n)}(x_0)$ for $0 \leq n \leq M$ of a function $f : \mathbb{R} \to \mathbb{R}$, the function can be represented as a template

```
template<typename T>
T f(T x);
```

Using a generic type `T` allows for `x` to be of a regular type such as `double`, but also allows for `boost::math::differentiation::autodiff_fvar<>` types.

Internal calls to mathematical functions must allow for argument-dependent lookup (ADL). Many standard library functions are overloaded in the `boost::math::differentiation` namespace. For example, instead of calling `std::cos(x)` from within `f`, include the line `using std::cos;` and call `cos(x)` without a namespace prefix.

Calling f and retrieving the calculated value and derivatives:

```
using namespace boost::math::differentiation;
autodiff_fvar<T,M> x = make_fvar<T,M>(x0);
autodiff_fvar<T,M> y = f(x);
for (int n=0 ; n<=M ; ++n)
    std::cout << "y.derivative("<<n<<") == " << y.derivative(n) << std::endl;
```

`y.derivative(0)` returns the undifferentiated value $f(x_0)$, and `y.derivative(n)` returns $f^{(n)}(x_0)$. Casting `y` to type `T` also gives the undifferentiated value. In other words, the following 3 values are equal:

1. `f(x0)`

2. `y.derivative(0)`

3. `static_cast<T>(y)`

5.2 Multiple Variables

Independent variables are represented in autodiff as independent dimensions within a multi-dimensional array. This is perhaps best illustrated with examples. The `namespace boost::math::differentiation` is assumed.

The following instantiates a variable of $x = 13$ with up to 3 orders of derivatives:

```
autodiff_fvar<double,3> x = make_fvar<double,3>(13);
```

This instantiates **an independent** value of $y = 14$ with up to 4 orders of derivatives:

```
autodiff_fvar<double,0,4> y = make_fvar<double,0,4>(14);
```

Combining them together **promotes** their data type automatically to the smallest multidimensional array that accommodates both.

```
    // z is promoted to autodiff_fvar<double,3,4>
    auto z = 10*x*x + 50*x*y + 100*y*y;
```

The object `z` holds a 2-dimensional array, thus `derivative(...)` is a 2-parameter method:

$$\texttt{z.derivative(i,j)} = \frac{\partial^{i+j} f}{\partial x^i \partial y^j}(13,14) \qquad \text{for } (i,j) \in \{0,1,2,3\} \times \{0,1,2,3,4\}.$$

A few values of the result can be confirmed through inspection:

```
    z.derivative(2,0) == 20
    z.derivative(1,1) == 50
    z.derivative(0,2) == 200
```

Note how the position of the parameters in `derivative(...)` match how `x` and `y` were declared. This will be clarified next.

5.2.1 Two Rules of Variable Initialization

In general, there are two rules to keep in mind when dealing with multiple variables:

1. Independent variables correspond to parameter position, in both the initialization `make_fvar<T,...>` and calls to `derivative(...)`.
2. The last template position in `make_fvar<T,...>` determines which variable a derivative will be taken with respect to.

Both rules are illustrated with an example in which there are 3 independent variables x, y, z and 1 dependent variable $w = f(x, y, z)$, though the following code readily generalizes to any number of independent variables, limited only by the C++ compiler/memory/platform. The maximum derivative order of each variable is `Nx`, `Ny`, and `Nz`, respectively. Then the type for `w` is `boost::math::differentiation::autodiff_fvar<T,Nx,Ny,Nz>` and all possible mixed partial derivatives are available via

$$\texttt{w.derivative(nx,ny,nz)} = \frac{\partial^{n_x+n_y+n_z} f}{\partial x^{n_x} \partial y^{n_y} \partial z^{n_z}}(x_0, y_0, z_0)$$

for $(n_x, n_y, n_z) \in \{0,1,2,...,N_x\} \times \{0,1,2,...,N_y\} \times \{0,1,2,...,N_z\}$ where x_0, y_0, z_0 are the numerical values at which the function f and its derivatives are evaluated.

In code:

```
    using namespace boost::math::differentiation;

    using var = autodiff_fvar<double,Nx,Ny,Nz>; // Nx, Ny, Nz are constexpr size_t.

    var x = make_fvar<double,Nx>(x0);       // x0 is of type double
    var y = make_fvar<double,Nx,Ny>(y0);    // y0 is of type double
    var z = make_fvar<double,Nx,Ny,Nz>(z0); // z0 is of type double

    var w = f(x,y,z);

    for (size_t nx=0 ; nx<=Nx ; ++nx)
        for (size_t ny=0 ; ny<=Ny ; ++ny)
            for (size_t nz=0 ; nz<=Nz ; ++nz)
                std::cout << "w.derivative("<<nx<<','<<ny<<','<<nz<<") == "
                    << w.derivative(nx,ny,nz) << std::endl;
```

Note how `x`, `y`, and `z` are initialized: the last template parameter determines which variable x, y, or z a derivative is taken with respect to. In terms of the ε-polynomials above, this determines whether to add $\varepsilon_x, \varepsilon_y$, or ε_z to x_0, y_0, or z_0, respectively.

In contrast, the following initialization of `x` would be INCORRECT:

```
    var x = make_fvar<T,Nx,0>(x0); // WRONG
```

Mathematically, this represents $x_0 + \varepsilon_y$, since the last template parameter corresponds to the y variable, and thus the resulting value will be invalid.

5.2.2 Type Promotion

The previous example can be optimized to save some unnecessary computation, by declaring smaller arrays, and relying on autodiff's automatic type-promotion:

```
using namespace boost::math::differentiation;

autodiff_fvar<double,Nx> x = make_fvar<double,Nx>(x0);
autodiff_fvar<double,0,Ny> y = make_fvar<double,0,Ny>(y0);
autodiff_fvar<double,0,0,Nz> z = make_fvar<double,0,0,Nz>(z0);

autodiff_fvar<double,Nx,Ny,Nz> w = f(x,y,z);

for (size_t nx=0 ; nx<=Nx ; ++nx)
    for (size_t ny=0 ; ny<=Ny ; ++ny)
        for (size_t nz=0 ; nz<=Nz ; ++nz)
            std::cout << "w.derivative("<<nx<<','<<ny<<','<<nz<<") == "
                << w.derivative(nx,ny,nz) << std::endl;
```

For example, if one of the first steps in the computation of f was `z*z`, then a significantly less number of multiplications and additions may occur if `z` is declared as `autodiff_fvar<double,0,0,Nz>` as opposed to `autodiff_fvar<double,Nx,Ny,Nz>`. There is no loss of precision with the former, since the extra dimensions represent 0 values. Once `z` is combined with `x` and `y` during the computation, the types will be promoted as necessary. This is the recommended way to initialize variables in autodiff.

6 Function Guidelines

In order to write a function that accurately calculates derivatives for use with autodiff, there are a few guidelines that will help in getting accurate derivatives:

- Operate on open intervals of real numbers as a whole, rather than individual points. Avoid returning values based on individual point values in the domain. For example, if an implementation of the `fourth_power()` function above has as its first statement `if (x == 2) return 16;` then no derivatives will be available for $x = 2$; instead `y.derivative(i) == 0` for all `i>0`. Each value within a connected subset of real numbers must be calculated in terms of an analytic function. The extent to which it is not will be reflected in the inaccuracy of the derivatives.

- Avoid intermediate singularities in both the calculated value and its derivatives. For example, $y = \sqrt{x}$ followed by $z = y^2$ may appear to be safe for $x \geq 0$ however when $x = 0$ then all non-zero derivatives of z will be NaN. This is due to the infinite derivative of $y = \sqrt{x}$ at $x = 0$.

- When constructing a piecewise analytic function, take the average value of each side of the domain for points on the boundary between analytic functions. For example, the function $g(x) = \max(0, x)$ should be defined in this manner:

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

  This give sensible values for both $g(0) = 0$ and $g'(0) = \frac{1}{2}$.

7 Acknowledgments

- Kedar Bhat — C++11 compatibility, 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