Derivative Library Tutorial

Wang Feng

March 18, 2014

1 Headers, Namespace and Compilation

The header file for derivative is

```
#include <derivative/derivative.hpp>
```

The header file for second derivative is

```
#include <derivative/second_derivative.hpp>
```

The namespace of derivative is

```
using namespace numeric;
```

A typical commandline compilation is

```
g++ -o test test.cc -std=c++11 -O2 -IPATH/TO/HEAD/FILE
```

2 Derivative

2.1 derivatives of normal functions

To calculate the derivative of a simple function $f(x, y) = \sin x \cos y$, we first need to define the function in c++ code:

```
double fxy ( double x, double y )
{
    return std::sin( x ) * std::cos( y );
}
```

then generate the $\frac{\partial f(x,y)}{\partial x}$ using

```
auto const& dfx = numeric::make_derivative<0>( fxy );
```

and $\frac{\partial f(x,y)}{\partial y}$ using

```
auto const& dfy = numeric::make_derivative<1>( fxy );
```

to evaluate $\left.\frac{\partial f(x,y)}{\partial x}\right|_{(1,2)}$, we simply call dfx as a normal c++ function

```
std::cout << "df / dx at ( 1, 2 ) is " << dfx( 1, 2 ) << "\n";
```

also, to evaluate $\left.\frac{\partial f(x,y)}{\partial y}\right|_{(1,2)}$

```
std::cout << "df / dy at ( 1, 2 ) is " << dfy( 1, 2 ) << "\n";
```

2.2 derivatives of functions that receive a pointer as argument

Same function as in the previous subsection, but we make some modification to make it receive only one pointer:

```
double fxy( double* x )
{
    return std::sin(x[0]) * std::cos(x[1]);
}
```

then define the derivatives:

```
auto const& dfdx = numeric::make_derivative( fxy, 0 );
auto const& dfdy = numeric::make_derivative( fxy, 1 );
```

to evaluate them at the point(1,2), we make an array x[], then call them as normal c functions:

```
double x[] = { 1.0, 2.0 };
std::cout << "\ndf/dx at (1.0, 2.0) is " << dfdx(x) << "\n";
std::cout << "\ndf/dy at (1.0, 2.0) is " << dfdy(x) << "\n";
```

2.3 more than functions

Our derivative can also deal with functors, lambda objects etc., here is an example with a functor:

```
#include <derivative/derivative.hpp>
#include <iostream>
#include <cmath>

struct sfxy
{
    double a;

    sfxy( double a_ = 2.0 ) : a(a_) {}

    double operator()( double* x ) const
    {
        return a * std::sin(x[0]) * std::cos(x[1]);
    }
};

int main()
{
    sfxy const fxy( 5.0 );

    auto const& dfdx = numeric::make_derivative( fxy, 0 );
    auto const& dfdy = numeric::make_derivative( fxy, 1 );

    double x[] = { 1.0, 2.0 };
    std::cout << "\ndf/dx at (1.0, 2.0) is " << dfdx(x) << "\n";
    std::cout << "\ndf/dy at (1.0, 2.0) is " << dfdy(x) << "\n";

    return 0;
}
```

3 Second Derivaitve

The code creating the second derivative objects is very similar to the code creating the first derivatives, but with one more parameter.

Second derivatives for a normal c function:

```
double fxyz( double x, double y, double z )
{
    return std::sin(std::pow( x, y ) * z);
}

void fxyz_test()
{
    auto const& fxyz_00 = numeric::make_second_derivative<0,0>( fxyz );
    auto const& fxyz_01 = numeric::make_second_derivative<0,1>( fxyz );
    auto const& fxyz_02 = numeric::make_second_derivative<0,2>( fxyz );
    auto const& fxyz_10 = numeric::make_second_derivative<1,0>( fxyz );
    auto const& fxyz_11 = numeric::make_second_derivative<1,1>( fxyz );
    auto const& fxyz_12 = numeric::make_second_derivative<1,2>( fxyz );
    auto const& fxyz_20 = numeric::make_second_derivative<2,0>( fxyz );
    auto const& fxyz_21 = numeric::make_second_derivative<2,1>( fxyz );
    auto const& fxyz_22 = numeric::make_second_derivative<2,2>( fxyz );

    std::cout << "\nfxyz_00 at(1.0, 2.0, 3.0) is " << fxyz_00( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_01 at(1.0, 2.0, 3.0) is " << fxyz_01( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_02 at(1.0, 2.0, 3.0) is " << fxyz_02( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_10 at(1.0, 2.0, 3.0) is " << fxyz_10( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_11 at(1.0, 2.0, 3.0) is " << fxyz_11( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_12 at(1.0, 2.0, 3.0) is " << fxyz_12( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_20 at(1.0, 2.0, 3.0) is " << fxyz_20( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_21 at(1.0, 2.0, 3.0) is " << fxyz_21( 1.0, 2.0, 3.0 ) << "\n";
    std::cout << "\nfxyz_22 at(1.0, 2.0, 3.0) is " << fxyz_22( 1.0, 2.0, 3.0 ) << "\n";
}
```

And for a function receiving a pointer as parameter:

```
double gxyz( double* x )
{
    return std::sin(std::pow( x[0], x[1] ) * x[2] );
}

void gxyz_test()
{
    auto const& gxyz_00 = numeric::make_second_derivative( gxyz, 0, 0 );
    auto const& gxyz_01 = numeric::make_second_derivative( gxyz, 0, 1 );
    auto const& gxyz_02 = numeric::make_second_derivative( gxyz, 0, 2 );
    auto const& gxyz_10 = numeric::make_second_derivative( gxyz, 1, 0 );
    auto const& gxyz_11 = numeric::make_second_derivative( gxyz, 1, 1 );
    auto const& gxyz_12 = numeric::make_second_derivative( gxyz, 1, 2 );
    auto const& gxyz_20 = numeric::make_second_derivative( gxyz, 2, 0 );
    auto const& gxyz_21 = numeric::make_second_derivative( gxyz, 2, 1 );
    auto const& gxyz_22 = numeric::make_second_derivative( gxyz, 2, 2 );

    double x[] = { 1.0, 2.0, 3.0 };
    std::cout << "\ngxyz_00 at(1.0, 2.0, 3.0) is " << gxyz_00( x ) << "\n";
    std::cout << "\ngxyz_01 at(1.0, 2.0, 3.0) is " << gxyz_01( x ) << "\n";
    std::cout << "\ngxyz_02 at(1.0, 2.0, 3.0) is " << gxyz_02( x ) << "\n";
    std::cout << "\ngxyz_10 at(1.0, 2.0, 3.0) is " << gxyz_10( x ) << "\n";
    std::cout << "\ngxyz_11 at(1.0, 2.0, 3.0) is " << gxyz_11( x ) << "\n";
    std::cout << "\ngxyz_12 at(1.0, 2.0, 3.0) is " << gxyz_12( x ) << "\n";
    std::cout << "\ngxyz_20 at(1.0, 2.0, 3.0) is " << gxyz_20( x ) << "\n";
    std::cout << "\ngxyz_21 at(1.0, 2.0, 3.0) is " << gxyz_21( x ) << "\n";
    std::cout << "\ngxyz_22 at(1.0, 2.0, 3.0) is " << gxyz_22( x ) << "\n";
}
```