

Automatic differentiation via C++ operators overloading

Link: [Implementation on GitHub](#)

Automatic differentiation is a powerful technique which allows calculation of sensitivities (derivatives) of a program output with respect to its input, owing to the fact that every computer program, no matter how complex, is essentially evaluation of a mathematical function.

In banking, automatic differentiation has many applications including, but not limited to risk management of financial derivatives, solving optimization problems and calculation of various valuation adjustments.

Motivation for AD

Calculation of sensitivities has been done long time before introduction of the AD. Traditional approach is to approximate derivative of a function using central finite difference:

$$\frac{\delta}{\delta x}f(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

The problem with this approach is the fact that too big differentiation step h ignores second-order risk (convexity).

On the other hand, differentiation step which is too small brings to the light another complication related to the way how floating-point model of a CPU works. Operations on floating-point operands with disproportionate exponents lead to serious rounding errors. Example of such operation is $x + h$ from the formula above, since x is considerably larger than h .

Furthermore, finite difference method is not only imprecise, but also extremely time-consuming for high-dimensional problems. In the context of banking, consider interest rate delta ladder calculation of a book of trades. In such situation, PV of every trade in a book needs to be calculated for every single interest rate risk factor (different currencies, indices and tenors), as illustrated by the following pseudo-code with multiple function calls:

```
pv_base      = calc_pv(trade, {rate1, rate2, rate3, rate4, ...})
pv_rate1_bump = calc_pv(trade, {rate1+bump, rate2, rate3, rate4, ...})
pv_rate2_bump = calc_pv(trade, {rate1, rate2+bump, rate3, rate4, ...})
pv_rate3_bump = calc_pv(trade, {rate1, rate2, rate3+bump, rate4, ...})
pv_rate4_bump = calc_pv(trade, {rate1, rate2, rate3, rate4+bump, ...})
...
```

The problem gets even more complicated for situations involving second-order risk. Moreover, due to increased regulatory requirements imposed on banks as part of the CCAR and FRTB frameworks, many banks are nowadays forced to reorganize their risk calculation and stress-testing infrastructures. In many of such situations, automatic differentiation can be the only plausible solution.

Principle of automatic differentiation

The fundamental advantage of automatic differentiation is the ability to calculate function's result together with sensitivities to all its inputs in a single function call, as illustrated by the following pseudo-code:

```
{pv_base, delta_rate1, delta_rate2, ...} = calc_pv(trade, {rate1, rate2, ...})
```

There are few approaches to AD such as "operators overloading-based", "handwritten" and "code-transformation". This article address approach which utilizes C++ templates and overloading of C++ operators. As each of the named approaches has its own advantages and disadvantages, the choice depends on a problem domain, maturity of the software project and constraints of the programming language.

AD via operators overloading

As mentioned, operators overloading-based AD utilizes two features of the C++ language, i.e. overloading of operators and function templates.

Operators overloading in the context of AD is used to alter behavior of elementary C++ operations such as "+", "-", "*", and "/" in order to record derivatives to its operands alongside the results as the calculation progress.

We will also show how **function templates** are used in order to redefine existing C++ functions with AD-aware data types (e.g. ADDouble), on which the C++ operators are overloaded.

For the illustration purposes, let's consider the following C++ program:

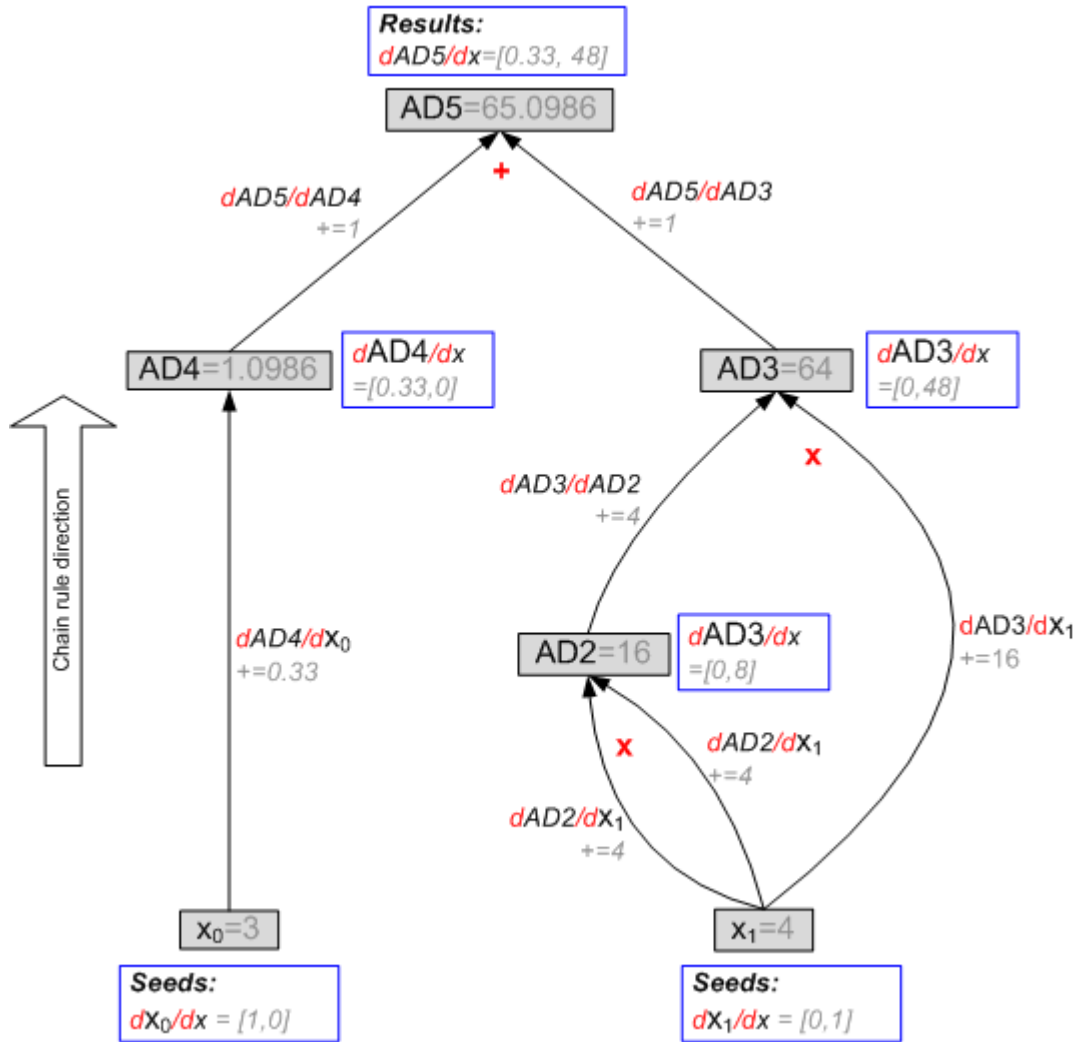
```
double f(double x0, double x1)
{
    return log(x0) + x1 * x1 * x1;
}
```

```
double x0 = 3;
double x1 = 4;
double y = f(x0, x1);
```

The sequence of mathematical operations together with temporary variables (denoted here as AD0 .. AD5) would look as follows:

```
AD0 = x0 = 3
AD1 = x1 = 4
AD2 = AD1 * AD1 = 4 * 4 = 16
AD3 = AD2 * AD1 = 16 * 4 = 64
AD4 = log(AD0) = log(3) = 1.09861
AD5 = AD4 + AD3 = 1.09861 + 64 = 65.09861
```

The tree representation of this calculation would look as below:



Now, the only question remains how to overload C++ operators so the derivatives will be recorded for each operation automatically. Let's define a new data type `ADDouble`. This variable behaves pretty much the same way as the standard `double`, except the fact that it has unique identifier. This identifier is used to track the sequence of operations as illustrated on the figure above. The tree-representation of this calculation is stored in a global storage called **AD Engine**.

```
struct ADDouble
{
    ADDouble(ADEngine& engine, double value) : _engine(engine), _value(value)
    {
        _id = engine._id_counter++;
    }
    double _value;
    NodeId _id;
};
```

The next step is to overload "+" and "*" operators and logarithm function for `ADDouble` data type. Each time when the operator is executed, the function will instantiate a new `ADDouble` data type and it will store direct derivatives of the result with respect to its input into the AD Engine tree.

```
inline ADDouble operator+(const ADDouble& l, const ADDouble& r)
{
    ADEngine& e = l._engine;
    ADDouble out(e, l._value + r._value);
    e.add_direct_derivative(out, l, 1.);
    e.add_direct_derivative(out, r, 1.);
    return out;
}
```

```

inline ADDouble operator*(const ADDouble& l, const ADDouble& r)
{
    ADDouble out(l._engine, l._value * r._value);
    ADEngine& e = out._engine;
    e.add_direct_derivative(out, l, r._value);
    e.add_direct_derivative(out, r, l._value);
    return out;
}
inline ADDouble log(const ADDouble& x)
{
    ADDouble out(x._engine, log(x._value));
    ADEngine& e = out._engine;
    e.add_direct_derivative(out, x, 1/x._value);
    return out;
}

```

Applying chain rule to the calculation tree

Calculation tree contains only derivatives of atomic operations with respect to their operands. In order to obtain sensitivity of the overall program output with respect to the initial program input, the chain rule for derivatives needs to be applied:

$$\frac{\delta}{\delta x} f(g(x)) = \frac{\delta}{\delta g(x)} f(g(x)) \frac{\delta}{\delta x} g(x)$$

Referring to the example above, let's say we are interested in sensitivity of function $y = f(x_0, x_1)$ to input variables x_0 and x_1 . Variables y , x_0 , x_1 are represented in the derivatives tree by nodes AD5, AD0 and AD1.

Applying chain rule means multiplying and summing the following direct derivatives together in order to get $\frac{\delta}{\delta x_0}$:

$$\begin{aligned} \frac{\delta}{\delta x_0} f(x_0, x_1) &= dAD5 / dAD4 \times dAD4 / dAD0 \\ &= 1 \times 0.33333 = 0.33333 \end{aligned}$$

$$\begin{aligned} \frac{\delta}{\delta x_1} f(x_0, x_1) &= dAD5 / dAD3 \times dAD3 / dAD2 + dAD3 / dAD1 \\ &= 1 \times (4 * (4 + 4) + 16) = 48 \end{aligned}$$

The chain rule is implemented in a function `ADEngine::get_derivative()` (see source code on [GitHub](#))

Operators overloading-based AD from user's perspective

In order to implement operators overloading-based automatic differentiation, user needs to template all functions and methods through which the AD-aware calculation will flow. The previously mentioned function would be for example templatised as follows:

```

template <typename T>
T f(T x0, T x1)
{
    return log(x0) + x1 * x1 * x1;
}

```

The final steps involve invocation of the AD-aware calculation from the [main program](#) function:

Step 1: Create AD engine which contains derivatives tree

```
ADEngine e;
```

Step 2: Create independent variables. Later, we will request derivative of the result with respect to these variables

```
ADDouble x0(e, 3);
```

```
ADDouble x1(e, 4);
```

Step 3: Perform the calculation by invoking the target function only once. All derivatives will be calculated automatically.

```
ADDouble y = f(x0, x1);
```

```
cout << "y = " << y.get_value() << endl;
```

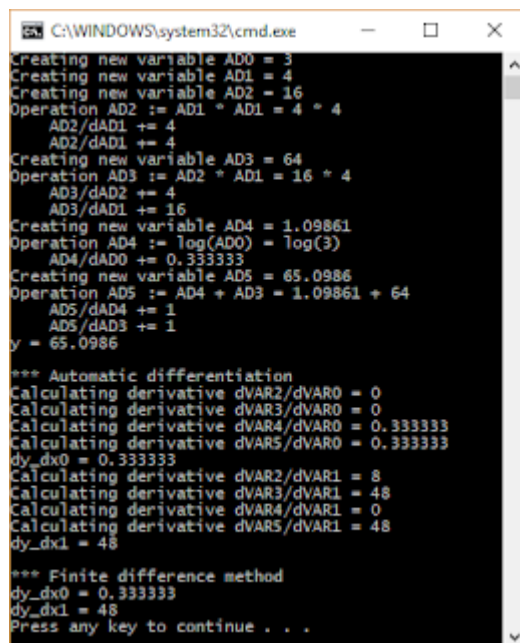
Step 4: Retrieve derivatives with respect to the input variables x0 and x1

```
cout << endl;
```

```
cout << "*** Automatic differentiation" << endl;
```

```
cout << "dy_dx0 = " << e.get_derivative(y, x0) << endl;
```

```
cout << "dy_dx1 = " << e.get_derivative(y, x1) << endl;
```



```
C:\WINDOWS\system32\cmd.exe
Creating new variable AD0 = 3
Creating new variable AD1 = 4
Creating new variable AD2 = 16
Operation AD2 := AD1 * AD1 = 4 * 4
AD2/dAD1 += 4
AD2/dAD1 += 4
Creating new variable AD3 = 64
Operation AD3 := AD2 * AD1 = 16 * 4
AD3/dAD2 += 4
AD3/dAD1 += 16
Creating new variable AD4 = 1.09861
Operation AD4 := log(AD0) = log(3)
AD4/dAD0 += 0.333333
Creating new variable AD5 = 65.0986
Operation AD5 := AD4 + AD3 = 1.09861 + 64
AD5/dAD4 += 1
AD5/dAD3 += 1
y = 65.0986

*** Automatic differentiation
Calculating derivative dVAR2/dVAR0 = 0
Calculating derivative dVAR3/dVAR0 = 0
Calculating derivative dVAR4/dVAR0 = 0.333333
Calculating derivative dVAR5/dVAR0 = 0.333333
dy_dx0 = 0.333333
Calculating derivative dVAR2/dVAR1 = 8
Calculating derivative dVAR3/dVAR1 = 48
Calculating derivative dVAR4/dVAR1 = 0
Calculating derivative dVAR5/dVAR1 = 48
dy_dx1 = 48

*** Finite difference method
dy_dx0 = 0.333333
dy_dx1 = 48
Press any key to continue . . .
```

Final thoughts

The aim of this article is to illustrate the principle of operators overloading-based automatic differentiation. The implementation is provided just for the sake of illustration and is not meant to be used in production environment. For this purpose, I advice to use one of the existing high-performance AD libraries such as [NAG DCO/C++](#).

Operators overloading-based approach to automatic differentiation has the following advantages:

- Performed on the level of atomic C++ operations. Therefore, it is fully transparent to higher-level language constructs such as function calls, classes or inheritance hierarchies.
- Suitable for legacy projects as adding AD support is just a matter of templatising the existing algorithms.
- Easy to comprehend from the user's perspective.
- Compared to the handwritten approach to AD, fairly resilient to bugs.
- Readily-available commercial libraries.

Disadvantages:

- In some circumstances quite memory consuming due to the fact that every single arithmetical operation leaves memory footprint.
- Due to memory concerns, not suitable for data-intensive algorithm which performs iterative calculations such as Monte Carlo.