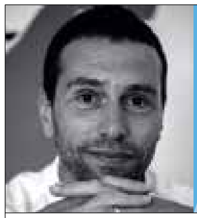
Computation Graphs for AAD and Machine Learning Part I: Introduction to Computation Graphs and Automatic Differentiation

Article *in* Wilmott · November 2019

DOI: 10.1002/wilm.10804

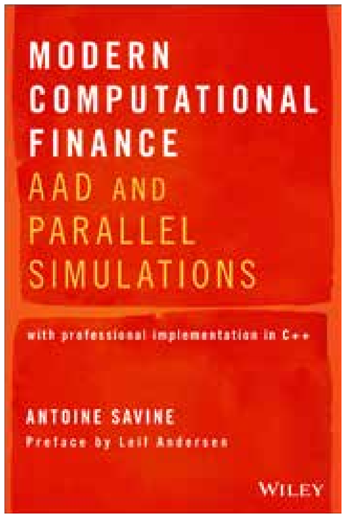
Antoine Savine
DANSKE BANK



Antoine Savine

Computation Graphs for AAD and Machine Learning Part I: Introduction to Computation Graphs and Automatic Differentiation

First in a series of three articles with code, exploring the notion of computation graph, with words, mathematics and code, and application in Machine Learning and finance to compute a vast number of derivative sensitivities with spectacular speed and accuracy.


MODERN COMPUTATIONAL FINANCE
AAD AND PARALLEL SIMULATIONS
with professional implementation in C++
ANTOINE SAVINE
Preface by Leif Andersen
WILEY

In this introductory article with code, adapted from Chapter 9 of the book *Modern Computational Finance* (Wiley, 2018) [10], we explore the automatic generation of computation graphs and the subsequent computation of the differentials of an arbitrary scalar calculation code in constant time, automatically and in a non-invasive manner.

Constant time differentiation gives us the ability to compute a large number of derivative sensitivities with spectacular accuracy and speed. It has been called 'holy grail' of sensitivity computation [6] and classified among the 30 greatest numerical algorithms of the 20th century [12]. It forms the backbone of both Deep Learning, and the risk management of financial Derivatives. In fact, without it, banks could not compute risk reports of large Derivatives books in real time, and the training of deep neural networks would take years on the fastest computers available today.

Historically, constant time differentiation was achieved with somewhat different implementations of similar ideas. In Derivatives finance and many other scientific fields, it has been implemented with *Automatic Adjoint Differentiation* or AAD. In the field of Machine Learning (ML), it is typically implemented in the form of *Back-Propagation* (BP).[1]

We shall call a 'differentiation framework', a library containing all the necessary pieces to perform constant time differentiation of arbitrary calculation code. Some frameworks are general (built to differentiate any kind of calculation), and some are specifically built to differentiate certain types of calculations with maximum efficiency. Examples include TensorFlow for Deep Learning or the companion code to [10] for the risk management of financial Derivatives. Many other commercial and open-source frameworks are available. We build a simple framework in this article, with C++ code to illustrate and clarify our purpose. This code was written for pedagogical purpose, not in order to scale to large production tasks. Although we produce a fully working framework, our code is not meant for maximum efficiency or suitability for parallel processing, it doesn't respect best practices like encapsulation, const or exception correctness, and it doesn't scale to production risk management libraries. For an implementation written for efficiency and suitable for large scale production, we refer to chapters 10 to 15 of the book [10] and its companion code, freely available on https://github.com/asavine/CompFinance.

Constant time differentiation: AAD and back-propagation, are widely applied in ML frameworks and Derivatives risk management systems. A vast amount of literature covers the application of ML frameworks to easily train algorithms like deep neural networks, leaving the details of a necessary efficient differentiation to the platform. The application of AAD to Derivatives risk management is also covered in a fair amount of literature, not least [5], who pioneered the application of AAD to finance. This article focuses on the algorithm itself and its development in C++, and postpones its practical application to a realistic problem in the financial Derivatives space to part III.

The article is divided into three parts, published sequentially. The first part introduces the main ideas and explores in detail the key notion of computation graph. The second part builds on these notions to produce a simple AAD framework. The third part applies the framework to quickly and accurately compute a large number of differentials in Dupire's [3] local volatility model, implemented with Monte-Carlo simulations, and demonstrates that even a simplistic implementation of AAD bears spectacular results in this context.

The code is available on https://github.com/asavine/WilmottArticle. It has been tested to compile and work on Windows 10, 64bit platform, with Visual Studio 2017.

1 Introduction to AAD and back-propagation

The purpose of AAD is the *automatic* generation of computation graphs and the subsequent computation of the differentials of an arbitrary *scalar* calculation code in *constant time*, *automatically* and in a *non-invasive manner*.

1.1 Scalar calculations

A *Scalar* calculation produces a single result out of a potentially large collection

Even a complicated financial calculation, like the Counterparty Value Adjustment (CVA) of a large portfolio of sophisticated Derivatives, is a scalar calculation, albeit a complex one, both in terms of modelling, development and code, and in terms of run time

of inputs, through a potentially large number of mathematical operations, coded in a possibly complex computer program, with various degrees of nested function calls, object manipulation, data structures and algorithms. As a trivial example, an implementation of the Black and Scholes formula [2] produces a scalar number (the option price C) out of five market and model parameters (depending on the exact implementation, the current time 0, underlying spot price S, interest rate r, dividend yield y and volatility σ, not counting transaction parameters expiry T and strike K). Consequently, it has five first order differentials to its five parameters, all well known and collectively referred to as 'Greeks'. We recall below Black and Scholes's formula, together with a simple implementation in C++ (in BlackScholes.h):

$$C(K,T) = DF(T)[F(T)\,N(d_1) - KN(d_2)],$$
$$DF(T) = e^{-rT},\ F(T) = Se^{(r-y)T},\ d_{1/2} = \frac{\log\frac{S}{K} \pm \frac{\sigma^2}{2}T}{\sigma\sqrt{T}}$$

N is the cumulative Gaussian distribution, implemented in file gaussians.h as *normalCdf()* with Zelen and Severo's classic approximation of 1964.

```
#include  "gaussians.h"

double blackScholes(
const double spot,
const double rate,
const double yield,
const double vol,
const double strike,
const double mat)
{
    double df = exp(-rate * mat),
        fwd = spot * exp ((rate - yield) * mat),
        std  =  vol * sqrt (mat);
    double  d = log(fwd / strike) / std;
    double  d1 = d  +  0.5  *  std ,  d2  =  d - 0.5 * std ;
    double  p1 = normalCdf(d1), p2 = normaCdf(d2);
    return  df * (fwd * p1 - strike * p2);
}
```

Perhaps less trivially, the Counterparty Value Adjustment (CVA) of a large portfolio of sophisticated Derivatives is also a scalar calculation, albeit a complex one, both in terms of modeling, development and coding, and in terms of run time. CVA generally involves the Monte-Carlo simulation of a hybrid model to produce a real number (the CVA) out of a large number (routinely in the thousands) of market and model parameters: credit curves, yield curves, asset prices, currency exchange rates, volatility curves and surfaces, large correlation matrices and possibly additional model parameters like stochastic volatility or mean-reversion. We will not build CVA code in this article, but we will develop and differentiate a simple Monte-Carlo simulation in Dupire's model [3] in Part III.

On the contrary, to compute the individual values of the transactions in a Derivatives book is *not* a scalar calculation, because it produces multiple outputs, the values of

>

all the transactions in the book. It is only the global value of the book that is a scalar calculation. It follows that the combined risk report of a trading book may be computed very efficiently with AAD, as we shall see, whereas the collection of the risks of its individual transactions does not fit the scalar criterion and would therefore take an order of magnitude longer run time. This is worth pointing out, because, with conventional differentiation using finite differences, run time would be identical for the risk of the combined portfolio or the risks of its individual transactions.[2]

AAD has two distinct facets: a rather simple, if somewhat mind-twisting, mathematical formalism, and a rather advanced practical implementation, which must extract computation graphs to run adjoint differentiation over the steps of a given calculation, in the reverse order

In the context of Deep Learning (DL), the loss function is also a scalar computation, which measures the prediction error of a sample of inputs fed through the network, compared to the corresponding labels, as a function of the weights which define the network, the number of which may be in the millions in a typical application to computer vision or natural language processing.

Derivatives risk management and Machine Learning therefore share a crucial dependency on the ability to quickly compute the large number of differentials of a heavy scalar calculation. Since the publication of the ground-breaking and award-winning "Smoking Adjoints" by Giles in Glasserman in 2006 [5], the Derivatives industry widely adopted AAD for this purpose. The ML community, on the other hand, adopted back-propagation, implemented in frameworks like TensorFlow to efficiently compute gradients of cost functions behind the scenes. Although the implementation details of AAD (in modern Derivatives risk management systems) and back-prop (in ML frameworks like TensorFlow) have important differences, they both achieve the key *constant time differentiation* property, and they both achieve it through identical mathematics (*adjoint differentiation* or AD) and similar implementation: extract the computation graph from the calculation and perform adjoint calculus through the graph, in reverse order.

1.2 Constant time differentiation

Constant time differentiation means that the differentials to all the inputs of a scalar calculation are computed in a time similar to *one* execution of the calculation, irrespective of the number of differentials.

Conventional differentiation by finite differences (FD) is linear in the number of differentials: each differential is computed by repeating the calculation after shifting the corresponding input by a small amount. With thousands or millions of inputs, the calculation must be repeated thousands or millions of times to produce all the differentials. Neither the risks of complex derivatives portfolios, nor the gradient of the loss in neural networks, could be computed by such means in reasonable time.

Constant time differentiation is achieved through *adjoint differentiation* (AD), a clever application of the derivatives chain rule to efficiently back-propagate the differentials of the final, scalar result, through the chain of operations that constitute the calculation, down to its inputs. Adjoint mathematics is simple, yet somewhat mind-twisting at first, and we shall review it shortly. For now, we point out that AD requires a representation of the chain of operations involved in the calculation, also called a *computation graph* or, in AAD lingo, a *tape*. This is one crucial difference with finite differences. All FD needs to compute differentials is a black-box executable to run the calculation repeatedly. In return, its complexity is linear in the number of differentials. AD, on the contrary, computes differentials an order of magnitude faster, in constant time. But it needs more than a black-box executable. It needs a representation of the entire computation graph in memory. AD differentiates calculations *analytically*, by application of the chain rule over the sequence of operations. It can only do this when the computation graph is known. Compared to conventional differentiation, we need to perform an additional step, the extraction of the computation graph.

Therefore, AAD has two distinct facets. Its theory consists in simple (yet, often hard to comprehend at first) mathematics, implemented as an iterative calculation through the nodes of a calculation graph. Its practice requires a rather advanced (depending on efficiency requirements) exercise in meta-programming, the (automatic) extraction of the computation graph of (arbitrary) calculation code.

We will review adjoint mathematics in detail in the second part of the article, but here is a primer to illustrate our prior statements. Denote the n scalar inputs to the calculation by y_i, with $1 \leq i \leq n$. Denote $N - n$ the number of elementary mathematical operations (sum, product, log, sqrt...) involved in the calculation, and y_i the sequence of the results of these operations for $n + 1 \leq i \leq N$.

Any calculation can be represented in these terms, from a simple expression like Black and Scholes's formula, to the CVA of a portfolio of complex Derivatives, computed with Monte-Carlo simulations of a sophisticated hybrid model. Everything always boils down to a sequence of elementary mathematical operations.

Each operation i in this sequence depends on zero, one or two previously calculated arguments y_j[3] (a sum depends on two arguments, a logarithm depends on one

The first A in AAD stands for Automatic and means that the library extracts the computation graph, not the developer

argument, a constant does not depend on any argument) with $j < i$ (an argument must have been computed before it is used in an operation), called *ancestors* of operation i. Denote A_i the set ancestors of operation i. Then, A_i is either empty, or it consists in one or two indices lower than i. Similarly, when j is an ancestor to i, we say that i is a *successor* to j. Denote S_j the set of successors to operation j. An operation may have many successors (all future operations that take j as an argument), all of index superior to j. The final result of the calculation is y_N. This is a scalar amount, and the only one we differentiate, by assumption of a scalar calculation. The computation graph is a representation of the whole sequence of operations, together with their nature (what is operation i: constant or input, sum, product, log, sqrt, ...) and dependency structure (successors to ancestors).

An evaluation of the calculation computes all the y_i in a sequence from the inputs $y_{1:n}$ to the output y_N. From the knowledge of the computation graph and the intermediate results y_i, available after an initial evaluation, we can trivially compute their derivatives $\frac{\partial y_i}{\partial y_j}$ for all i and $j \in A_i$. The set of elementary operations is very limited, so their derivatives may be easily tabulated. An application of the chain rule gives us the differentials of the final result Y_N to all intermediate results y_j, including the inputs:

$$\frac{\partial y_N}{\partial y_j} = \sum_{i \in S_j} \frac{\partial y_N}{\partial y_i} \frac{\partial y_i}{\partial y_j}$$

This formula expresses the differential of y_N to y_j as a function of the differentials of y_N to the successors to operation j, which index has to be above j, with known weights $\frac{\partial y_i}{\partial y_j}$, for all j. $\frac{\partial y_N}{\partial y_j}$ is also called the *adjoint* of y_j (implicitly, for the result y_N) and denoted a_j. Since, trivially, $a_N = \frac{\partial y_N}{\partial y_N} = 1$, the formula allows to compute all adjoints, iterating through the sequence of operations *in the reverse order* (because adjoints are known functions of *future* adjoints), starting with a_N and ending with a_1.

However, this formula is not practical or efficient, because it accumulates adjoints from ancestors, which all have a variable and potentially large number of successors. The exact same result is achieved by accumulating the adjoints of ancestors *from their successors*, which is more efficient, since a successor may only have up to two ancestors. We call this the *adjoint algorithm*: starting with $a_N = 1$ and having initialized $a_j = 0$ for $1 \le j < N$ (this is called 'seeding' in AAD lingo), iterate the following for $i = N, N-1, N-2, ..., 1$, adjusting the adjoints a_j of the zero, one or two ancestors $j \in A_i$ by:

$$a_j += \frac{\partial y_i}{\partial y_j} a_i$$

This algorithm produces the exact same results that the formula we found by a trivial application of the chain rule, and therefore, correctly computes all the differentials of the final result to all the inputs.[4]

Further, the algorithm visits every operation involved in the calculation exactly once, and, for every operation, it adjusts at most two ancestors. Its run time is therefore at most twice the time of one evaluation of the calculation, irrespective of the number of differentials, as promised. But it heavily relies on the knowledge of the sequence of the operations in the calculation, that is, its computation graph.

1.3 Automatic differentiation

The first A in AAD stands for *automatic*[5] and means that the computation graph, necessary for constant time differentiation, is extracted automatically by the AAD framework. Developers only code the calculation, that is, the steps to produce the scalar result from the inputs. The framework figures by itself the sequence of operations and dependencies (a.k.a the computation graph), and runs the adjoint algorithm through the graph to efficiently produce all the differentials, behind the scenes.

It is possible, in principle, to implement AD manually for every piece of code one wishes to differentiate, writing the computation graph on the back of an envelope and coding the steps of the adjoint algorithm in reverse order through the graph. This is even probably the most efficient possible implementation of AD. However, manual adjoint differentiation is a tedious, perilous exercise, and it takes an unrealistic amount of maintenance and testing effort to ensure that the differentiation code is, and remains, synchronized with the calculation code. Chapter 8 of [10] explains the manual development of adjoint code, with examples from Derivatives finance, and discusses in detail the benefits and drawbacks of doing so.

Most modern implementations of AD include some mechanism for the automatic extraction of the computation graph, allowing developers to write the calculation code and leave it to the framework to worry about its efficient differentiation. Depending on the application, goal and efficiency requirement, implementations vastly differ in both the *nature* of the nodes in the graph, and the practical means of extracting them from the calculation code.

Despite the promise that frameworks differentiate arbitrary code behind the scenes, it is highly unwise to attempt using a framework without an intimate understanding of the mechanisms it employs. This may work for a time, but more often than not, ends up producing wrong derivatives, or producing them with disappointing speed, sometimes slower than FD. A differentiation framework is a sophisticated, high precision instrument built for speed and efficiency, and meant for expert users. The goal of this article is to explain all the mechanisms one may need to build their own framework or efficiently reuse an existing one.

>

Modern Machine Learning frameworks are not implemented in terms of scalar operations, but in terms of tensor operations. Tensors are vectors, matrices and higher dimensional friends. Tensor operations take tensors for inputs and return tensors

1.3.1 Computations nodes

Let us discuss different possibilities for what exactly may be encoded in one node of the computation graph.

One node per mathematical operation In this introductory article, we consider a node in the graph to be an elementary mathematical operation, such as a scalar sum, difference, product, division, exponentiation, logarithm, or, perhaps, a standard function like the normal density or cumulative normal distribution. This is the conventional form of AAD, and one that is easiest to comprehend and implement. We have a closed, limited number of node types, all nodes have at most two child nodes (ancestors). How to evaluate a node knowing its ancestors, and differentiate it with respect to its ancestors, is known and tabulated once and for all. For example, we know that a multiplication node is evaluated by multiplication of its arguments. Its derivative to the right hand side is the left hand side and vice-versa.

One node per mathematical expression Cutting edge implementation of AAD, introduced in [7], represents in every single node on the graph, not one mathematical operation, but one whole mathematical expression. For example, a call to code like:

```
template  <class T>
T f (T x[5])
{
     auto y1 =  x [2]  *  (5.0  *  x [0]  +  x [1]);
     auto y2 =  log (y1);
     auto y = (y1 + x[3] * y2) * (y1 + y2);
     return y;
}
```

which consists in eight operations (not counting the constant 5), would be contained in a single node for the whole expression:

$$y = (y_1 + x_3\, y_2)(y_1 + y_2),\, y_1 = x_2\, (5x_0 + x_1),\, y_2 = \log\, (y_1)$$

with five ancestors (the 5 entries to the array x, of which *x[4]* is inactive, something a good implementation should figure and optimize). This technique, sometimes called 'tape compression' [11], reduces the size of the graph in memory and stores the nodes with a better coalescence for the execution of the adjoint algorithm, resulting in a sharp increase in the speed of execution (up to five times faster). The implementation is rather advanced, because we no longer have a closed, limited set of node types for which the number of arguments, and how to compute values and derivatives from them, are all known in advance. The framework must figure all of this on its own. This is achieved by *template meta-programming*, more precisely *expression templates*, a powerful paradigm in programming languages supporting it, like C++, which, not only figures on its own, the number of arguments to the expression, and how to evaluate and differentiate it, but also, delegates much of this work to compile time, saving precious run time and achieving efficiency close to optimal manual AD code.

The cutting edge implementation of AAD with expression templates is explained and discussed in detail in the chapter 15 of [10]. The implementation is freely available in its companion code on GitHub.

It is important to understand, however, that this more complicated, more efficient, implementation, bears the exact same mathematical complexity than the simplistic implementation explained here. The number of calculations involved is the same. The speed-up is only computational. It minimizes *administrative* work (like moving from node to node in the graph) delegating part of it to compile time, by contrast with *mathematical* work (performing the actual calculations), which remains identical. It also leverages the ability of modern hardware to optimize computations when data is better localized in memory.

Any implementation of AD, however naive, produces an *order of magnitude* speed improvement, compared to conventional differentiation, in the sense that differentiation time is independent of the number of differentials. A smarter, more complicated, more efficient implementation of AD, may offer an additional improvement, but this additional improvement can be only linear (x times faster, whatever the number of differentials).

One node per tensor operation Finally, modern Machine Learning frameworks like TensorFlow are not implemented in terms of scalar operations (functions of real numbers that produce real numbers) but in terms of *tensors*. Tensors are vectors, matrices, and higher-dimensional friends. Tensor operations are functions of tensors that return tensors. Examples of tensor operations include matrix products or element-wise application of scalar functions. There are many good reasons why ML frameworks are based on tensors rather than numbers:

1. Machine Learning, especially Deep Learning, consists in sequences of matrix operations. Deep Learning equations (known as *feed-forward equations*) are naturally expressed in matrix terms, and are most naturally coded in the same terms.
2. Modern CPUs and more so GPUs and TPUs are highly optimized for tensor operations. Concretely, a modern computer may leverage different forms of parallelism to compute a tensor operation vastly faster than the sequence of its scalar operations.
3. ML frameworks are typically coded in a fast language like C++ with main APIs in Python, which is a vastly slower language. Tensor operations minimize slow computations in Python and costly communications between Python and C++. This technique is not unique to ML frameworks, it is also implemented, for example, in NumPy, Python's de-facto standard math library, and is largely responsible for the success of Python in the data science community, despite the slow performance of the language.

It is highly inadvisable to apply scalar AAD (where nodes are scalar operations) to tensor code (where operations manipulate tensors). For example, the product of a 2,000 by 1,000 matrix by a 1,000 vector (reasonable by Deep Learning standards) would be represented by 4 billion nodes on the graph, one for each scalar multiplication and one for each scalar addition. Each one of these nodes may store two pointers to its arguments (16 bytes on a x64 platform) and the corresponding two derivatives (another 16 bytes if double precision). This is 128GB for this operation alone! In addition to insane memory consumption, to traverse the 4 billion nodes at run time would slow the computation down to a crawl. By contrast, a tensor based framework would register the whole operation in a single node, storing the vector and the matrix as the right and left hand side 'derivatives' in the multivariate sense. This is 2,001,000 numbers, occupying about 16MB of memory in double precision (by contrast with 128GB; in addition, TensorFlow standard is single precision, halving memory requirement to 8MB). To visit this node for adjoint differentiation takes a matrix by vector product, optimized on modern CPU, GPU or TPU. As formerly men-

Computation Graphs can be automatically extracted from calculation code with the operator overloading technique when the programming language supports it, like modern C++

tioned, Deep Learning, and, more generally, Machine Learning code is mostly tensor based, and, therefore, well suited to tensor based differentiation and not suitable for differentiation with a scalar framework.

Conversely, it would be clumsy to apply a tensor framework to (mostly) scalar code. Most financial Derivatives libraries, for instance, are coded in terms of scalars, with real numbers, generally represented by the *double* type, as computation unit. Present financial libraries are coded in terms of (large) sequences of scalar operations with occasional (generally small) matrix operations. A tensor framework would represent all those scalar operations as operations on tensors of size 1, with the unnecessary associated complexity and overhead. Financial calculations *may* be converted to tensor operations. For instance, a Monte-Carlo simulation may be coded in terms of tensors, where one dimension is the path index. Such an implementation is called a *vectorized simulation*. It allows us to simulate batches of paths simultaneously and may run faster than conventional path-wise simulations on modern hardware, leveraging SIMD (Same Instruction, Multiple Data) parallelism built in modern CPUs, GPUs and TPUs. It also makes the code considerably harder to read, write and maintain. Despite an active current research and development trend to combine ML with classic financial models, which may result in combined tensor based libraries, programming practice in Derivatives finance and many other fields outside ML is set to remain scalar in the near future. This results in two somewhat redundant types of adjoint differentiation frameworks, one for scalar code like financial simulations and one for tensor code like neural networks.

We note again that the type of the framework (scalar or tensor) does not affect mathematical complexity in any way. The number of mathematical operations involved in the evaluation and the differentiation of the code remains strictly identical, for a tensor operation as a whole, or its decomposition in scalar operations. The 'only' difference is in computational considerations, the storage of the graph in memory, the administrative cost of traversal of the graph and the speed of execution of mathematical operations on modern hardware.

>

In this article, we only cover scalar frameworks of the simplest form, where every mathematical operation is stored in its own node. The notions we develop naturally extend to expression based and tensor-based frameworks. The naive implementation we present still demonstrates an order of magnitude increase in speed, compared to conventional differentiation, and bears rather spectacular results in many contexts, as we shall see with an example of practical relevance in Derivatives finance in part III. Of course, it cannot compete with a more complicated implementation, built for maximum efficiency, like the one in chapter 15 of [10].

1.3.2 Extracting computation graphs

Frameworks also apply different techniques to extract computation graphs from calculation code. The computation graph is not only a somewhat abstract object that can be drawn on a white board to help reason about code. Adjoint differentiation needs a concrete computation graph represented in computer memory. Useful applications of computation graphs besides AD include lazy evaluation, the ability to execute a graph on demand with different inputs, sequentially or in parallel, on different hardware (like GPU) without the need to rewrite or recompile the code. For this purpose, as for AD, it is necessary to build, in computer memory, the computation graph of the calculation. Note that the calculation may be simple and self contained, like Black and Scholes formula on page 7, or it may consist in a complicated procedure, involving many layers of nested function calls, the creation and manipulation of objects, data structures and algorithms.

At first sight, it may look like we need a dedicated program to inspect and analyze calculation code in order to extract its graph, in other words, we need a compiler. Indeed, attempts were made to write compiler-like programs to extract and exploit computation graphs. This is tremendous, specialized work, and the resulting programs are generally not able to handle calculation code written in a fully flexible, arbitrary manner. We shall not explore this option any further, referring interested readers to a specialized publication like [9].

A natural solution is for the platform to expose functions in its API to explicitly create and manipulate nodes in the graph. This is the solution implemented in TensorFlow, where, for instance, the instruction $Z = tf.matmul(X, Y)$ puts a matrix product node on the graph, with (previously created) ancestors nodes referenced by X and Y, and stores a reference to the newly created node in the variable Z. This is how TensorFlow can build graphs in C++ or CUDA (GPU) memory (for efficient evaluation and differentiation) from various client platforms like Python. But this is not an *automatic* differentiation. It cannot take arbitrary calculation code and extract its graph. Calculation code must be written with TensorFlow objects and functions to work with TensorFlow. In order to move it to a different framework, the code would have to be rewritten entirely.

The option we are exploring in the rest of this paper applies *operator overloading* as a lighter and simpler alternative. Overloading extracts computation graphs *at run time* in a portable, non-invasive manner, in standard C++ (or any other language supporting overloading) without the need for a specialized compiler or library.

1.4 Non-invasive graph extraction with operator overloading

Non-invasive means that the framework is able to extract graphs from calculation code and differentiate it, without modifying it. There are a few minor caveats. First, calculations must be templated for the real number representation type, so our code on page 7 must be modified as follows, to represent real numbers with a template class 'T' in place of *double*:

```cpp
template <class T>
T blackScholes(
const T spot,
const T rate,
const T yield,
const T vol,
const T strike,
const T mat)
{
    auto df = exp(- rate * mat),
        fwd = spot * exp ((rate - yield) * mat),
        std = vol * sqrt (mat);
    auto d = log (fwd / strike) / std;
    auto d1 = d + 0.5 * std, d2 = d - 0.5 * std;
    auto p1 = normalCdf (d1), p2 = normalCdf(d2);
    return df * (fwd * p1 - strike * p2);
}
```

This is a minor invasion, and templating on the real number representation type quickly becomes a natural habit.[6] But it is an invasion nonetheless,[7] and it means, in particular, that the entire source for the calculation must be available. Chapter 8, section 8.2 of [10] briefly discusses how to handle calls to third party libraries, for which the source may not be available.

Provided a fully templated source, we can extract the graph at run time without further invasion, with operator overloading. Operator overloading refers to the ability to modify the behaviour of standard mathematical functions (like log) and operators (like +, -, *, /) when they are applied to custom types. When the following lines of code are executed:

```
    double x = 2.0, y = 3.0, z = 4.0;
    double r = x * y + log(z));
```

the compiler knows exactly what to do, and in what order: take the logarithm of *z*, add *y*, multiply by *x*, store the result in *r*. This is because the variables *x*, *y* and *z* are declared with the *double* type, and the compiler knows exactly how to apply functions and operator like log, + and * to doubles.

Now, suppose we use our own custom class to represent real numbers, call it *Number*. Then, the following lines of code:

```
    Number x = 2.0, y = 3.0, z = 4.0;
    Number r = x * (y + log(z));
```

would not compile, because the compiler doesn't know how to apply mathematical functions and operators log, + or * to objects of type *Number*. We must tell it explicitly how to do that. For instance,

```
struct Number
{
    double v;
    Number(const double& value = 0.0) : v (value) {}
};

Number operator +(const Number lhs, const Number rhs)
{
    return lhs.v + rhs. v;
}

Number operator*(const Number lhs, const Number rhs)
{
    return lhs. v * rhs. v;
}

Number log (const Number arg)
{
    return log (arg. v);
}

ostream& operator << (ostream& ost, const Number& n)
{
    ost << n.v ;
    return ost ;
}

int main()
{
    Number x = 2.0, y = 3.0, z = 4.0;
    Number r = x * (y + log(z));
    cout << "Result = " << r << endl;  //  8.77259
}
```

compiles and does the right thing, because we spelled it out for the compiler, what to do when functions and operators are applied to the *Number* type, in the definition of the overloaded *operator+()*, *operator*()* and *log()*. When the compiler applies these operators to *Number* types, it executes *exactly what we wrote in the definition of the overloads*. It doesn't have to be a mathematical operation. For example,

```
Number operator+(const Number lhs, const Number rhs)
{
    cout << "Adding " << lhs.v << " to " <<  rhs. v << endl;
    return lhs. v + rhs. v;
}

Number operator*(const Number lhs, const Number rhs)
{
    cout << "Multiplying " << lhs. v <<  " by " <<  rhs. v << endl;
    return lhs. v * rhs . v;
}

 Number log (const Number arg)
{
     cout << "Taking the logarithm of "  << arg. v << endl;
     return log(arg. v);
}
```

still performs the mathematical operations, but also logs them on the console as they execute. The compiler applies operator precedence so operations are executed in the same order as with doubles, but what is executed is what the developer defined in the overloads.

This feature gives us the means to construct the calculation graph at run time, for any calculation code templated on the real number representation type. In the same way we just easily logged the sequence of operations on the console, we can build nodes on the graph every time an operation executes. For this purpose, we overload all mathematical operators and functions for our *Number* type to build a node in the graph and correctly connect successors to ancestors. Then, we execute the code with *Number* as a template argument. As long as the source is entirely templated, the execution constructs the graph, including nested function calls, data structures and algorithms of any sort.

This methodology effectively produces calculation graphs in a reliable manner, without recourse to complicated, specialized tools or libraries. The downside of producing graphs at run time is run time overhead. The demonstration code in this article does not concern itself with maximum efficiency. But even the more complicated, way more efficient code in the chapter 10 of [10] is subject to *some* node creation overhead, every time a mathematical operation is invoked. Meta-programming with expression templates, covered in chapter 15, effectively delegates much of the overhead to compile time, achieving run time speed similar to manual adjoint code, with the convenience of automatic, overload based graph extraction.

2 Working with computation graphs

In this section, we temporarily set aside AAD, BP and differentiation is general, to explore calculation graphs on their own right. We will revert to differentiation in the next part. The code is available at https://github.com/asavine/WilmottArticle. It is incomplete, since we only overload a few operators and functions. Readers can easily complete the code as an exercise if they wish.

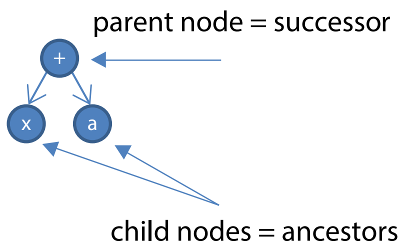
2.1 Definition and examples

Every calculation defines a calculation graph. The *nodes* are all the elementary mathematical calculations involved: +, −, *, /, *pow*, *exp*, *log*, *sqrt* and so forth.[8] The *edges* that join two nodes represent operation (successor) to argument (ancestor) relationships: *child* nodes are the arguments (*ancestors*) to the operation in the *parent* (*successor*) node. The *leaves* (nodes without ancestors) are constants and inputs to the calculation.[9]

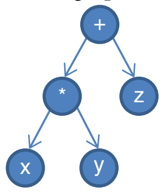
It may be confusing that, in the context of computation graphs, contrarily to real life, parents are successors and childs are ancestors. Parent nodes represent operations, child nodes represent arguments. Arguments are computed before operations. They are ancestors to the operation. Operations are computed after their arguments. They are successors to their arguments. For avoidance of doubt, see table below.

	graph	*computation*
operation	parent	successor
argument	child	ancestor

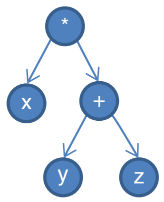
For example, in the expression $x + y$, + is the parent node on the graph and the successor of the child nodes x and y, who are the ancestors of the parent + node.



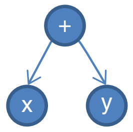
The structure of the graph reflects precedence among operators: for instance, the graph for $xy + z$ is:



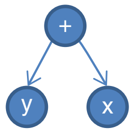
because multiplication has precedence, unless explicitly overwritten with parentheses. The graph for $x(y + z)$ is:



The graph also reflects the order of calculation selected by the compiler among equivalent ones. For instance, the graph for $x + y$ could be:



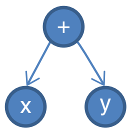
or:



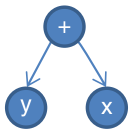
The compiler selects one of the possible calculation orders. Hence, the graph for the calculation *code* is not unique, but the graph for a given execution of the code is, the compiler having made a choice among the equivalent orders.[10] We work with the *unique* graphs that reflect the compiler's choice of the calculation order, and always

>

represent calculations left to right. Hence, the graph:



represents $x + y$ and:



represents $y + x$.

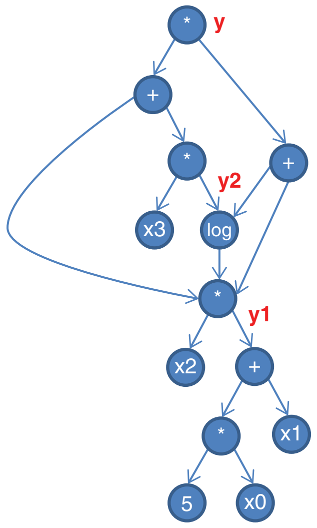
The following example computes a scalar y out of 5 scalars $(x_i)_{0 \le i \le 4}$ (where x_4 is *purposely* left out of the calculation) with the formula:

$$y = (y_1 + x_3 y_2)(y_1 + y_2),\ y_1 = x_2 (5x_0 + x_1),\ y_2 = \log (y_1)$$

This is a particularly simple and irrelevant example, which we retained nonetheless for demonstration purposes, since anything more complicated would make it difficult to explicitly draw the calculation graph on the page.

```
double f(double x[5])
{
    double y1 = x [2] * (5.0 * x[0] + x[1]);
    double y2 = log(y1);
    double y = (y1 + x[3] * y2) * (y1 + y2);
    return y ;
}
```

Its graph (assuming the compiler respects the left to right order in the code) is:



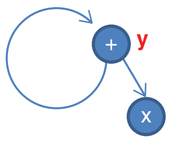
It is important to understand that complicated calculations, with nested function calls and object-oriented logic, like the ones that compute the value of an exotic option with Monte-Carlo simulations or Finite Difference Methods, or even the value of the CVA of a large netting set, or any kind of valuation or other mathematical or numerical calculation, all define a similar graph, perhaps with billions of nodes, but a calculation graph all the same, where nodes are elementary operations and edges represent operations (successors) to arguments (ancestors) relationships.

2.2 Calculation graphs and computer programs

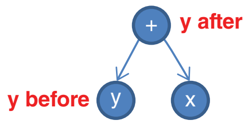
It is also important to understand that calculation graphs refer to *mathematical operations*, irrespective of how computer programs store them in *variables*. For instance, the calculation graph for:

```
y = y + x;
```

is most definitely *not*:



but:

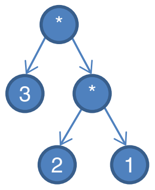


Although the variable y is reused and overwritten in the calculation of the sum, the sum defines a new calculation of its own right in the graph. Graph nodes are calculations, not variables.

Similarly, recursive function calls define new calculations in the associated graph: the following recursive implementation for the factorial:

```
unsigned fact (const unsigned n)
{
    return n <= 1 ? 1 : n * fact (n - 1);
}
```

defines the following graph when called with 3 as argument, where we can see that graphs automatically *unroll* recursive calls:



The fact that the function is defined recursively is irrelevant, the graph is only concerned with mathematical operations and their order of execution.

Loops are unrolled in the graph the same as recursive calls. This implies, in particular, that loops, like other control flow, are taken into account at the time when the graph is extracted, but do not themselves constitute part of the graph. Our graphs have no 'for', 'while' or 'if' nodes, which means that the graph represents, not the entire code, but one branch of its execution. The control flow itself is not part of the graph. This doesn't matter for differentiation: we compute differentials over one branch of execution, since control flow itself isn't differentiable -or continuous. But it may well matter for other applications, like sending extracted graphs for execution on different hardware. For this reason, TensorFlow has instructions to insert 'conditional' or 'while' nodes on its graphs. We will briefly revert to this rather technical problem of control flow, and refer to chapter 11, section 11.3 of [10] for a more complete discussion.

2.3 Directed Acrylic Graphs (DAGs)

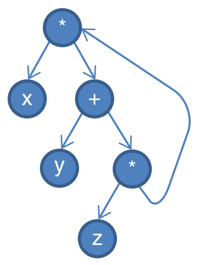
Computer programs do not define just any type of graph.

First, notice that a calculation graph is not necessarily a *tree*, a particularly simple type of graph, where every child node (ancestor) has exactly one parent (successor).[11] In the example page 56, the node y_2 has two parents and the node y_1 has three. We have commented earlier that an ancestor may have any number of successors in a computation graph. Computation graphs are not trees, so reasoning and algorithms that apply to trees don't necessarily apply to them.

Secondly, the edges are *directed*, successors to ancestors. An edge from y to x means that the result of x is an argument to operation y, like in $y = \log(x)$, which is of course not the same as $x = \log(y)$. Graphs where all edges are directed are called directed graphs. Computation graphs are always directed.

>

Finally, in a computation graph, a node cannot refer to itself, either directly or indirectly. The following graph, for instance:



is *not* a *computation* graph, and no code could ever generate a graph like this. The reason is that the arguments to an operation must be evaluated prior to the operation. The evaluation of ancestors always precedes the evaluation of successors. Calculation graphs are evaluated bottom-up.

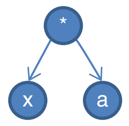
Graphs that satisfy such property are called *acyclic* graphs. Therefore, computation graphs are *directed acyclic graphs* or DAGs in short.

2.4 DAGs and control flow

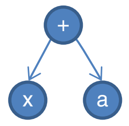
The DAG of a calculation only represents the mathematical operations involved, not the control flow. There are no nodes for *if*, *for*, *while* and friends. A DAG represents the chain of calculations under one particular control branch. For instance, the code

```
y = x > 0 ? x * a : x + a;
```

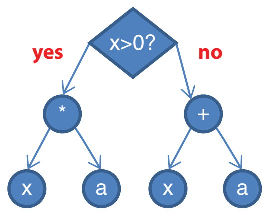
produces the DAG:



when executed with a positive input *x* and the different DAG:



when executed with a negative input. It doesn't have a composite DAG of the type:



This is *not* a type of DAG we will be dealing with. Control flow is irrelevant to differentiation. Our methodology applies control flow for the extraction of the DAG, but does not incorporate it in the DAG. Other applications of DAGs may need control flow in the DAG, but this cannot be achieved with operator overloading without invasion of the source code, because the operators *if*, *for* or *while* cannot be overloaded.

For example, in TensorFlow, the following code (in Python):

```
a = tf.constant (2.0)
b = tf.constant (5.0)
x = tf.placeholder (dtype = tf.float32)
```

```
y = tf.placeholder (dtype = tf.float32)
z = tf.multiply (a, b)

result = tf.cond(x < y, lambda: tf.add(x, z), lambda: tf.square (y))

sess = tf . Session ()
sess.run (result, feed_dict ={x: 3.0, y: 4.0}) # 13.0, x < y so add is executed
sess.run (result feed_dict ={x: 4.0, y: 3.0}) # 9.0, x > y so square is executed
```

puts a *conditional* node on the graph, where the condition is checked and the appropriate operation is executed *when the graph is evaluated, after it has been built*. By contrast, this code:

```
add_or_square = True
result2 = tf.add(x, z) if add_or_square else tf.square(y)

sess.run(result2, feed_dict ={x: 3.0, y: 4.0}) # 13.0
add_or_square = False
sess.run(result2, feed_dict={x: 3.0, y: 4.0}) # still 13.0
# the graph was build with add_or_square = True and remains unchanged
```

checks the condition when the Python code, which builds the graph, is executed, and constructs different nodes depending on the value of the Boolean variable *add_or_square* at that time. At evaluation time, the graph will always execute the same node.

For the purpose of differentiation alone, we don't need control flow on the graph. We work with *calculation* DAGs, not *code* DAGs. Calculation DAGs refer to one particular execution of the code, with a specific set of inputs, and a unique control branch, resulting in a specific output. They don't refer to the *code* that produced the calculation. AAD frameworks generally differentiate calculations, not code. It follows that AAD does not attempt to differentiate through the control flow itself: since there are no *if* nodes in the DAG, the AAD 'derivative' of an 'if' statement is 0.

This may look at first sight like a flaw with AAD, but how is any type of differentiation supposed to differentiate through control flow, something that is, by nature, discontinuous and not differentiable? To differentiate code with control flow involves 'smoothing' the calculation, that is, approximating it with a continuous one. We refer to [13] for details of smoothing control flow, and how it relates to fuzzy logic.

The problem of control flow with AAD is also discussed in more detail in section 11.3 of [10].

For avoidance of doubt, the presence of control flow in instrumented code does not prevent its correct differentiation with AAD frameworks: it is only the control flow itself that is not differentiated, the branch of execution responsible for the result is always differentiated correctly.

2.5 DAGs and control flow

A DAG is not just a theoretical representation of some calculation. *Operator overloading* can be applied to explicitly build a computation DAG in memory, at run time. The following code instantiates the +, * and *log* operators, leaving it as a (very easy) exercise for readers to instantiate other mathematical operators and functions. The code is also *extremely* inefficient because every operation allocates memory for its node on the DAG. The problem of memory is so fundamental for efficient AAD that chapter 10 of [10] is almost entirely devoted to it. In the second part of this article, we will also build a simple AAD framework in a much more efficient manner. Here, we focus on functionality alone. We also don't implement private encapsulation, const or exception correctness. This code is for demonstration only. We suggest readers take the time to fully understand it. This inefficient code nonetheless demonstrates the DNA of AAD.

First, we implement classes for the various types of nodes: +, *, *log* and *leaf*, in an object oriented hierarchy:

```
# include <memory>
# include <string>
# include <vector>

using namespace std;

class Node
```

```
{
protected:

    vector<shared_ptr<Node>> myArguments;

public:

    virtual ~Node () {}
};

class PlusNode : public Node
{
public:

    PlusNode(shared_ptr <Node> lhs, shared_ptr<Node> rhs)
    {
        myArguments.resize (2);
        myArguments[0] = lhs;
        myArguments[1] = rhs;
    }
};

class TimesNode : public Node
{
public:

    TimesNode (shared_ptr <Node> lhs, shared_ptr <Node> rhs)
    {
        myArguments.resize(2;
        myArguments [0] = lhs;
        myArguments [1] = rhs;
    }
};

class LogNode : public Node
{
public:

    LogNode(shared_ptr<Node> arg)
    {
        myArguments.resize(1);
        myArguments[0] = arg;
    }
};

class Leaf: public Node
{

    double myValue;

public:

    Leaf(double val)
        : myValue (val) {}

    double getVal ()
    {
        return myValue;
    }

    void  setVal (double val)
    {
        myValue = val;
    }
};
```

We apply a classical composite pattern (see the 'Gang of Four' book, or GOF, who pioneered the systematic application of design patterns in software engineering [4]) for the nodes in the graph. Concrete nodes are derived for each operation type and hold their arguments by base class pointers. The systematic use of smart pointers guarantees an automatic (if inefficient) memory management. It has to be *shared* pointers because multiple successors may share common ancestors. We can't have a successor release an ancestor still referenced by another successor. When all its successors are gone, the ancestor is released automatically.

Next, we design a custom number type. This type holds a reference to its node on the graph.[12] When a Number is initialized, it creates a leaf on the graph. We use this custom type in place of *doubles* in our calculation code.

```
class Number
{
    shared_ptr<Node> myNode;

public:

    Number(double val)
        : myNode (new Leaf (val)) {}

     Number (shared_ptr<Node> node)
         :  myNode (node) {}

     shared_ptr <Node> node ()
     {
         return myNode ;
     }

     void setVal(double val)
     {
    //  Cast to leaf, only leaves can be changed
    dynamic_pointer_cast<Leaf>(myNode)->setVal(val);
  }

  double getVal()
  {
    //  Same comment here, only leaves can be read
    return dynamic_pointer_cast<Leaf>(myNode)->getVal();
  }
};
```

dynamic_pointer_cast() is a C++11 standard function associated with smart pointers that returns a copy of the smart pointer of the proper type with its managed pointer casted dynamically.[13] This is necessary because only leaf nodes store a value.

Next, we *overload* the mathematical operators and functions for the custom number type so that when these operators and functions are executed with this type, the program does not evaluate the operator as normal, but, instead, constructs the corresponding node on the graph.

```
shared_ptr<Node> operator+(Number lhs, Number rhs)
{
    return shared_ptr <Node>(new PlusNode(lhs.node(), rhs.node()));
}

shared_ptr<Node> operator*(Number lhs, Number rhs)
{
    return shared_ptr<Node>(new TimesNode (lhs.node(), rhs.node()));
}

shared_ptr<Node> log (Number arg)
{
    return shared_ptr <Node>(new LogNode (arg.node()));
}
```

This completes our simplistic framework.[14] We can now build graphs *automatically*. We don't build the DAG by hand. The code does it for us when we instantiate it with our *Number* type. We could hard replace 'double' by 'Number' in the calculation code, or make two copies, one with doubles, one with Numbers. The best practice is *template* the calculation code on its number type, like this:

```
template <class T>
inline T f (T x[5])
{
    auto y1 = x [2] * (5.0 * x [0] + x[1]);
    auto y2 = log (y1);
    auto y = (y1 + x[3] * y2) * (y1 + y2);
    return y;
}
```

and call it with our custom type like that:

```
1 int main ()
2 {
3     Number x [5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
4     Number y = f (x);      \ label { line : aadPG1 }
5 }
```

Because we called *f*() with *Numbers* in place of *doubles*, line 4 does *not* calculate anything. All mathematical operations within *f*() execute our overloaded operators and functions. So, in the end, what this line of code does is construct in memory the DAG for the calculation of *f*(). To template calculation code in view of running of it with AAD, or extracting its graph for any other purpose, is called *instrumentation*.

2.6 Traversing the DAG

Once we have built the graph in memory, we may *traverse* it in many different ways, for many different purposes.

2.6.1 Lazy evaluation

First, we can evaluate the graph. This means, evaluate the DAG itself, irrespective to the function *f*(), which execution built it in the first place. In order to evaluate the graph, we *visit* its nodes in a specific order.

First, we note that arguments to an operation must be computed before the operation. This means that the processing of a node starts with the processing of its ancestors.

Secondly, we know that nodes may have more than one successor in a DAG. Those nodes will be processed multiple times if we are not careful. For this reason, we start the processing of a node with a check that this node was not processed previously. If this is verified, then, we process its ancestors, and then, only then, we evaluate the operation on the node itself, mark it as processed, and cache its result for the benefit of its successors.

In this algorithm, the only one step that is specific to evaluation is the execution of the operation on the node. All the rest is graph traversal logic, and applies to evaluation as well as other forms of *visits* we explore next. If we follow these rules, starting with the top node, we are guaranteed to visit (evaluate) each node exactly once, and in the correct order for the evaluation of the DAG.

Hence, we have the following graph *traversal* and *visit* algorithm: starting with the top node,

1. Check if the node was already processed. If so, exit.
2. Process the child nodes.
3. Visit the node: conduct the calculation, cache the result.
4. Mark the node as processed.

The third line of the algorithm is the only one specific to evaluation. The rest only relates to the order of the visits. More precisely, the traversal strategy implemented here is well known to the graph theory. It is called *depth-first postorder* or simply postorder. Depth-first because it follow every branch down to a leaf before moving on to the next branch. Postorder because the ancestors (arguments) of a node are always visited (evaluated) before the the node (operation). It follows that visits start on the leaves and end on the top node. Postorder implements a bottom-up traversal strategy and guarantees that every node in a DAG is processed exactly once, where ancestors are always visited before successors. It is the natural order for an evaluation. When we move on to differentiation, we will see that other forms of visits may need a different traversal strategy.

We implement our evaluation algorithm, separating traversal and visit logic into different pieces of code. The traversal logic goes to the base class, which also stores a flag that tells if a node was processed, and caches the result of its evaluation. For our information only, the order of calculation is also stored on the (base) node. Finally, we implement a simple recursive method to reset the processed flags on all nodes.

```
class Node
{
protected :

    vector<shared_ptr<Node>> myArguments ;

    bool         myProcessed = false ;
    unsigned     myOrder = 0;
    double       myResult;

 public :

     virtual  ~ Node () {}

     //  visitFunc:
   //  a function of Node& that conducts a particular form of visit
   //  templated so we can pass a lambda
  template <class V>
  void postorder(V& visitFunc)
  {
     //  Already processed -> do nothing
    if (myProcessed == false)
    {
      // Process ancestors first
      for (auto argument : myArguments)
        argument->postorder(visitFunc);

      // Visit the node
      visitFunc(*this);

      // Mark as processed
      myProcessed = true;
    }
  }

  // Access result
  double result()
  {
    return myResult;
  }

  // Reset processed flags
  void resetProcessed()
  {
    for (auto argument : myArguments) argument->resetProcessed();
    myProcessed = false;
  }
};
```

Note that the algorithm in *resetProcessed*() isn't optimal, as most of the code in this chapter. Next, we code the specific evaluation visitor as a virtual method: evaluation is different for the different concrete nodes (sum in a + node, product in a * node, and so on):

```
// On the base Node
virtual void evaluate() = 0;

// On the + Node
void evaluate() override
{
  myResult = myArguments[0]->result() + myArguments[1]->result();
}

// On the * Node
void evaluate() override
{
  myResult = myArguments[0]->result() * myArguments[1]->result();
}

// On the log Node
void evaluate() override
{
  myResult = log(myArguments[0]->result());
```

```
 }

//  On the Leaf
 void evaluate() override
 {
   myResult = myValue;
 }
```

We start the evaluation of the DAG from the Number that holds the result, and refers to the top node of the DAG. Recall that Numbers contain a shared pointer to the last assignment made to them.

```
//  On the Number class
 double evaluate()
 {
   myNode->resetProcessed();
   auto visit = [](Node& n) {n.evaluate(); };
   myNode->postorder(visit);
   return myNode->result();
 }
```

Now we can evaluate the DAG:

```
1 int main()
2 {
3     Number x [5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
4     Number y = f (x);      \ label { line : aadPg2 }
5     cout << y. evaluate() << endl;    // 797.751
6}
```

The evaluation of the mathematical operations in the calculation did not happen on line 4. That just constructed the DAG in memory. Nothing was calculated. The calculation happened on line 5 in the call to *evaluate*(), directly from the DAG, after it was built. That it returns the exact same result than a direct function call (with double as the number type) indicates that our code is correct.

To see this more clearly, we can change the inputs on the DAG and evaluate it again, without any further call to the function that built the DAG in the first place. The new result correctly reflects the change of inputs, as readers may easily check:

```
int main()
{
    Number x [5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
    Number y = f (x);
    cout << y.evaluate () << endl ;    // 797.751

  //  Change x0 on the DAG
  x[0].setVal(2.5);
  //  Evaluate the DAG again
  cout << y.evaluate() << endl;    // 2769.76
}
```

This pattern to store a calculation in the form a DAG, instead of conducting its evaluation straight away (or *eagerly*), and perform the evaluation at a later time, directly from the DAG, possibly repeatedly, possibly with different inputs set directly on the DAG, and possibly on a different device or machine, is called *lazy evaluation*. Lazy

>

evaluation is what makes AAD automatic, it is also the main principle behind, for example, expression templates (see chapter 15 of [10]). In financial Derivatives, lazy evaluation forms the backbone of cash-flow scripting (see [1]).

2.6.2 Calculation order

There is more we can do with a DAG than evaluate it lazily. For instance, we can store the calculation order number on the node, which will be useful later in this article. We identify the nodes by their calculation order and store that order on the node. We reuse our postorder code, we just need a new visit function. In this case, the visits don't depend on the concrete type of the node, we don't need a virtual visit function, so we can code the visitor directly on the base node.

```
//  On the base Node

 void setOrder(unsigned order)
 {
   myOrder = order;
 }

 unsigned order()
 {
    return myOrder;
  }

//  On the Number class

  void setOrder()
  {
    myNode->resetProcessed();
    auto visit = [order = 0](Node& n) mutable {n.setOrder(++order); };
    myNode->postorder(visit);
  }
```

The lambda defines a data member *order* that starts at 0. Since C++14, lambdas not only capture environment variables, but can also define other internal variables in the capture clause. The evaluation of the lambda modifies its *order* member, so the lambda must be marked mutable.

```
int main()
{
    //  Inputs
 Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
  //  Build DAG
 Number y = f(x);
  //  Number the nodes
 y.setOrder();
}
```

Our DAG is numbered after the execution of the code above, and each node is identified by its postorder. We can log the intermediate results in the evaluation as follows:

```
//  On the Number class

 void logResults()
 {
   myNode->resetProcessed();
```

```
  myNode->postorder([] (Node& n)
  {
    cout << "Processed node "
      << n.order() << " result = "
       << n.result() << endl;
   });
 }
```

after evaluation of the DAG. When we evaluate the DAG repeatedly with different inputs, we get different logs accordingly:

```
int main()
{
    //  Set inputs
Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

  //  Build the dag
 Number y = f(x);

  //  Set order on the dag
  y.setOrder();

   //  Evaluate on the dag
  cout << y.evaluate() << endl;   // 797.751

   //  Log all results
  y.logResults();

   //  Change x0 on the dag
  x[0].setVal(2.5);

   //  Evaluate the dag again
  cout << y.evaluate() << endl;   // 2769.76

   //  Log results again
  y.logResults();
}
```

For the first evaluation we get the log:

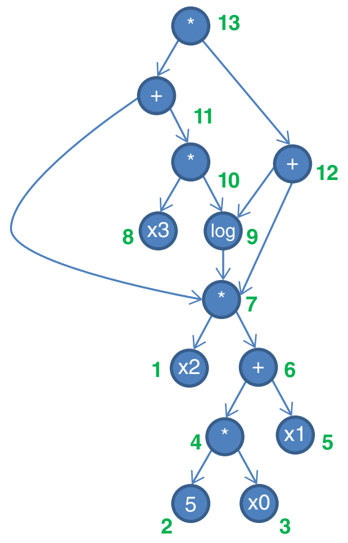
```
Processed node 1 result = 3
Processed node 2 result = 5
Processed node 3 result = 1
Processed node 4 result = 5
Processed node 5 result = 2
Processed node 6 result = 7
Processed node 7 result = 21
Processed node 8 result = 4
Processed node 9 result = 3.04452
Processed node 10 result = 12.1781
Processed node 11 result = 33.1781
```

>

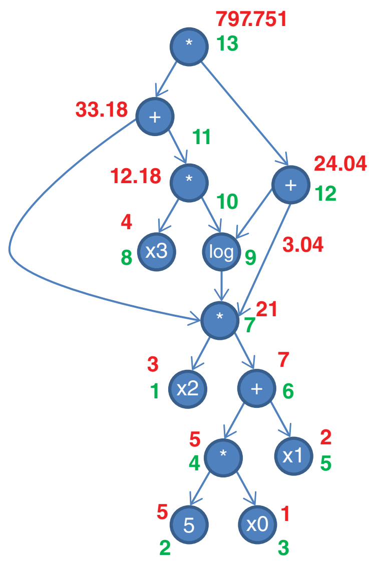
```
Processed node 12 result = 24.0445
Processed node 13 result = 797.751
```

For the second evaluation, we get different logs reflecting $x_0 = 2.5$.

We can display the DAG, this time identifying nodes by their evaluation order:



and intermediate results (for the first evaluation) as follows.



2.6.3 Reverse engineering

There is even more we can do with the DAG. For instance, we can reverse engineer an equivalent calculation program:

```
//  On the base Node
 virtual void logInstruction() = 0;

//  On the + Node
 void logInstruction() override
 {
   cout << "y" << order()
     << " = y" << myArguments[0]->order()
     << " + y" << myArguments[1]->order()
      << endl;
  }

 //  On the * Node
  void logInstruction() override
  {
    cout << "y" << order()
      << " = y" << myArguments[0]->order()
      << " * y" << myArguments[1]->order()
      << endl;
  }

 //  On the log Node
  void logInstruction() override
  {
    cout << "y" << order() << " = log("
      << "y" << myArguments[0]->order() << ")" <<
      endl;
  }

// On the Leaf
  void logInstruction() override
  {
    cout << "y" << order() << " = " << myValue << endl;
  }

// On the number class
  void logProgram()
  {
    myNode->resetProcessed();
    auto visit = [](Node& n) {n.logInstruction(); };
    myNode->postorder(visit);
  }

int main()
{
   // Inputs
  Number x[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

   // Build the DAG
  Number y = f(x);

   // Set order
  y.setOrder();

   // Log program
  y.logProgram();
}
```

We get the log:

```
y1 = 3
y2 = 5
y3 = 1
y4 = y2 * y3
y5 = 2
y6 = y4 + y5
y7 = y1 * y6
y8 = 4
y9 = log(y7)
y10 = y8 * y9
y11 = y7 + y10
y12 = y7 + y9
y13 = y11 * y12
```

>

We reconstructed an equivalent computer program from the DAG. Of course, we can change the inputs and generate a new computer program that reflects the new values of the leaves and produces different results.

Note that we managed to make visits (evaluation, instruction logging) depend on both the visitor and the concrete type of the visited node. Different things happen according to the visitor (evaluator, instruction logger) and the node (+, – and so forth). We effectively implemented GOF's *visitor* pattern, see [4]. This pattern is explained in more detail and systematically applied to scripts in our publication [1] (where we study trees, not DAGs, but the logic is the same).

We learned to automatically generate DAGs from templated calculation code with operator overloading. We learned to traverse DAGs in a specific order called postorder, and visit its nodes in different ways: lazily evaluate the DAG or reverse engineer an equivalent sequence of instructions.

Building calculation DAGs in memory has many useful applications besides adjoint differentiation. Suppose for example that we want to execute code on GPU. In principle, we must re-write a version of the code in CUDA, the general purpose code compiler for nVidia GPUs. This takes time, effort and specialized skills, repeatedly, for every calculation we may want to code for execution on GPU. Instead, we could develop a single CUDA program, once and for all, a 'graph executor', taking a graph as an argument to execute it on GPU. With this graph executor, combined with a graph extraction framework along the lines we just implemented, we could execute any present or future calculation on GPU without having to write any CUDA code: write our calculation code in standard C++, extract its graph and execute it on the CUDA executor. This is roughly how TensorFlow operates, and how it is able to transparently execute code on CPU or GPU. Of course, matters are more complicated in practice: large scalar graphs cannot be efficiently communicated back and forth between CPU and GPU memory, control flow must be somehow incorporated in the graph and memory must be carefully managed. But this is the core idea.

This section covered the 'Automatic' in 'Automatic Adjoint Differentiation'. In the next part, we move on to 'Adjoint Differentiation' and traverse the DAG extracted from calculation code, to compute its *differentials*, all of them, in a single traversal, in constant time. We will introduce the notion of *tape* to organize the nodes of a graph in a sequence, explore in detail how AAD differentiates calculations an order of magnitude faster by reversing the order of its operations and bring all the pieces together and develop a simple, yet fully functional AAD framework.

ENDNOTES

1. Although a framework like TensorFlow implements both BP and AAD.
2. Thankfully, a more sophisticated implementation of AAD can deal with individual risk reports rather efficiently, as covered in the chapter 14 of [10], titled "Multiple Differentiation in Almost Constant Time".
3. We have only a limited number of *elementary* mathematical operations: addition, multiplication, logarithm, square root, etc. All those operations are either unary or binary. Functions of three or more arguments may be split into sequences of unary or binary operations. In order to unify notations, we consider inputs and constants as operations with zero arguments.
4. As well as all the differentials to all intermediate results, which may or may not be useful information: see for instance [8] for an application to better train deep neural networks in Derivatives finance.
5. In the literature, it stands for either Automatic or Algorithmic. Algorithmic sounds better but Automatic better describes the method.
6. To systematically use the 'auto' type in place of 'T' in the body of functions and methods simplifies the process and offers efficiency benefits with more advanced AAD frameworks explained in the chapter 15 of [10].
7. So we should have advertised AAD as 'minimally invasive' instead of 'non-invasive'.
8. We define elementary operations as the building blocks of any calculation: scalar numbers, unary functions like *exp* or *sqrt* and binary arithmetic operators and functions like *pow*. In addition to the operators and mathematical functions available in standard C++, we consider as building blocks all the fundamental unary or binary functions, repeatedly called in calculation code, and which derivatives are known analytically, for example, the Gaussian density *n*() and cumulative distribution *N*(), particularly useful in financial calculations.
9. Not all leaf nodes are inputs. Leaf nodes may also be constants. But that does not matter for now. We will compute the partial derivatives of the final result to all the leaves here, chapter 10 of [10] explains how to optimize the constants away.
10. In all that follows, we neglect rounding errors that could, for example, make different from $xy + xz$ and only consider mathematical equivalence.
11. We are not talking about recombining trees here.
12. More precisely, a variable of type Number holds a reference to the operation last assigned to it.
13. See http://www.cplusplus.com/reference/memory/dynamic_pointer_cast/.
14. Although we only instantiated addition, multiplication and logarithm nodes, readers may easily instantiate all other functions and operators in the same manner.

REFERENCES

[1] Andreasen, J. and Savine, A. 2018. *Modern Computational Finance: Scripting for Derivatives* and xVA. Wiley Finance.
[2] Black, F. and Scholes, M. 1973. The pricing of options and corporate liabilities. *Journal of Political Economy* 81(3), 637–654.
[3] Dupire, B. Pricing with a smile. 1994. *Risk* 7(1), 18–20.
[4] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing.
[5] Giles, M. and Glasserman, P. 2006. Smoking adjoints: Fast evaluation of greeks in monte carlo calculations. Risk. Available at https://www.risk.net/landmarks-in-xva/5540761/smoking-adjoints-fast-evaluation-of-monte-carlo-greeks.
[6] Griewank, A. 2012. Who invented the reverse mode of differentiation. *Documenta Mathematica* Extra Volume ISMP, 389–400.
[7] Hogan, R. J. 2014. Fast r*everse-mode automatic differentiation using expression templates in C++. ACM Trans. Math. Softw* 40(4).
[8] Huge, B. and Savine, A. 2019. Deep Analytics. Vienna: QuantMinds.
[9] Naumann, U. 2012. *The art of differentiating computer programs: An introduction to algorithmic differentiation*. SIAM. https://epubs.siam.org/doi/book/10.1137/1.9781611972078.
[10] Savine, A. 2018. *Modern Computational Finance*, Volume 1: AAD and Parallel Simulations. Wiley.
[11] Sokol, A. Retrofitting AAD to your existing C++ library using tape compression. QuantsHub Webinar, 2015.
[12] Trefethen, N. 2015. Who invented the great numerical algorithms? https://people.maths.ox.ac.uk/trefethen/inventorstalk.pdf.
[13] Zadeh, L. A. 1996. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A. Zadeh*. World Scientific.