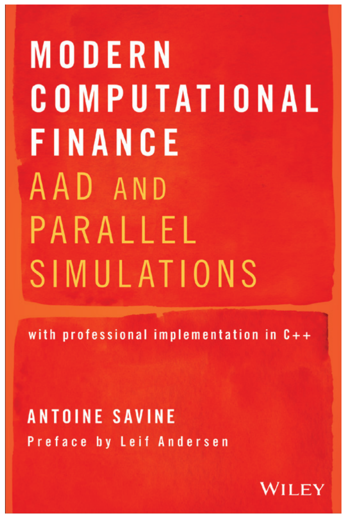
Antoine Savine

Computation Graphs for AAD and Machine Learning Part II: Adjoint Differentiation and AAD

Second 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

Abstract

The first part of this article in a series of three introduced the main ideas of Adjoint Differentiation (AD) and back-propagation (BP), and covered in detail the key notion of computation graph and how one can be extracted from calculation code. This part covers adjoint differentiation, and its implementation with *tapes*, which organize the nodes of the computation graph in a sequence, well suited for both evaluation (in the direct order) and differentiation (in the reverse order). We also discuss in depth how reversing the order of operations allows to reduce the complexity of differentiation by an order of magnitude, and bring the pieces together to develop a simplistic, yet fully functional AAD framework.
The next and final part applies the framework to the fast computation of a large number of differentials in Dupire's [2] 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 Adjoint differentiation

Recall the adjoint differentiation algorithm, described in the first part of the series. In what follows, we identify the nodes i in the DAG by their postorder traversal number. As before, we call y_i the result of the operation on node i. We denote $y = y_N$ the final result on the top node N. We denote A_i the set of ancestors (child nodes, arguments) of i and S_j the set of successors (parent nodes) to node j.

Mathematical operations are either unary (they take one argument like *log* or *sqrt* or the unary - that changes a number into its opposite) or binary (two arguments, like *pow* or the operators +, -, *, /). Hence A_i is a set of either 0, 1 or 2 nodes. Besides, because of postorder, the indices of the nodes in A_i are always less than i. The only nodes i for which A_i is empty are, by definition, the leaves.

The leave nodes are processed first in postorder traversal, so their node number i is typically low. Leaves represent the inputs to the calculation.

We define the adjoint of the result y_i on the node i, and denote a_i, the partial derivative of the final result $y = y_N$ to y_i:

$$a_i \equiv \frac{\partial y}{\partial y_i} = \frac{\partial y_N}{\partial y_j}$$

Then, we recall the adjoint equation:

$$a_N = \frac{\partial y_N}{\partial y_N} = 1$$

and, directly from the derivative chain rule:

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

The adjoint of a node is the sum of the adjoints of its successors, weighted by the partial derivatives of successor to ancestors. The operations on the nodes are elementary operations, which number is closed and limited, and which derivatives are known analytically: the derivative of a product to its right hand side is its left hand side, its derivative to its left hand side is its right hand side, and so on.

Note the reversal of the order, compared to evaluation: in an evaluation, the result of an operation depends on the values of the arguments. In the adjoint equation, it is the adjoint of the argument that depends on the adjoints of the operation. Hence, just like evaluation computes results bottom-up, ancestors first, adjoint computation propagates differentials top down, successors first. The adjoint of the top node, the final result, is know to be 1, and from there, adjoints propagate parent to childs, successor to ancestors, top down through the DAG, towards the leaves, whose adjoints are, by definition, the partial derivatives of the final result to the corresponding inputs.

We have seen that a direct implementation of the adjoint equation would be clumsy and inefficient, and that the same results could be efficiently produced by accumulating the adjoint of ancestors from their successors. The adjoints of every node are first seeded to 0, except the top node, which is seeded to 1. Visits to node i propagate its adjoint a_i to its ancestors, for every (zero, one or two) $j \in A_i$:

$$a_j += \frac{\partial y_i}{\partial y_j} a_i \qquad \text{(adjoint formula)}$$

Repeating this procedure in the reverse order $j = N, N - 1, N - 2, \ldots, 1$ ensures that every node is processed exactly once, and that the adjoints are correctly accumulated in accordance with the adjoint equation.

The derivatives of successors to direct ancestors, $\frac{\partial y_i}{\partial y_j}$ are known analytically, but depend on the values of the ancestors $(y_i)_{j \in A_i}$. This means that we must know all the intermediate results before we compute adjoints. Adjoint propagation is a *post-evalu-*

ation step. It is only after we built the DAG and evaluated it that we can propagate adjoints through it. At this point, all the intermediate results are cached on the corresponding nodes, so the necessary partial derivatives can be computed.[1]

We may now code adjoint propagation: set $a_N = 1$ and start with all other adjoints zeroed: for $i < N$, $a_i = 0$. We process all the nodes in the DAG, top to leaves, N to 1. When visiting a node, we must know its adjoint a_i, so we can implement its propagation to ancestors, in accordance with the algorithm (adjoint formula). This effectively computes the adjoints of all the nodes in the DAG, hence all the partial derivatives of the final result, with a single DAG traversal, hence, in constant time.

This point is worth repeating, since the purpose of AAD is constant time differentiation. All the adjoints of the calculation are accumulated from *one* top down traversal of its DAG. In this traversal, every node is visited once, and its adjoints are propagated to its ancestors, weighted by known local derivatives. Hence, the adjoint accumulation, like the evaluation, consists in one visit to every node (operation) on the calculation DAG. Adjoint accumulation, which effectively computes all derivative sensitivities, is of the same order of complexity as the evaluation, and constant in the number of inputs or differentials.

It is easy enough to code the visit routines. We store adjoints on the nodes and write methods to set and access them, like we did for evaluation results when exploring DAGs. We also need a method to reset all adjoints to 0, like we did for the processed flag. Finally, we log the visits so we can see what is going on.

```
// On the base Node
double myAdjoint = 0.0;

//...

public:

//...

// Note: return by reference
// used to get and set
double& adjoint()
{
    return myAdjoint;
}

void resetAdjoints()
{
    for (auto argument: myArguments) argument->resetAdjoints();
  myAdjoint = 0.0;
}

virtual void propagateAdjoint() = 0;

// On the + Node
void propagateAdjoint() override
{
    cout << "Propagating node " << myOrder
           << " adjoint = " << myAdjoint << endl;

myArguments[0]->adjoint() += myAdjoint;
myArguments[1]->adjoint() += myAdjoint;
}

// On the * Node
void propagateAdjoint() override
{
    cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

myArguments[0]->adjoint() += myAdjoint * myArguments[1]->result();
myArguments[1]->adjoint() += myAdjoint * myArguments[0]->result();
}

// On the log Node
void propagateAdjoint() override
{
    cout << "Propagating node " << myOrder
            << " adjoint = " << myAdjoint << endl;

myArguments[0]->adjoint() += myAdjoint / myArguments[0]->result();
}

// On the Leaf
void propagateAdjoint() override
{
    cout << "Accumulating leaf " << myOrder
            << " adjoint = " << myAdjoint << endl;
}

// On the number class

// Accessor/setter, from the inputs
double& adjoint()
{
    return myNode->adjoint();
}

// Propagator, from the result
void propagateAdjoints()
{
    myNode->resetAdjoints();
    myNode->adjoint() = 1.0;
// At this point, we must propagate sensitivities,
// but how exactly do we do that?}
}
```

Notice how the propagation code uses the intermediate results stored on the ancestor nodes to compute partial derivatives, as commented earlier. The propagation of adjoints on the unary and binary operator nodes implements the adjoint equation. Leaf nodes don't propagate anything, by the time propagation hits them, their adjoints are entirely accumulated, so their override of *propagateAdjoints()* simply logs the results.

We could easily code the visits but not start the propagation. It is clear that doing the same as for evaluation, something like:

```
myNode->postorder([](Node& n) {n.propagateAdjoint();});
```

is not going to fly. Postorder traverses the DAG arguments first, but for adjoint propagation, it is the other way around. Adjoints are propagated *successor to ancestor*. We must implement the correct traversal order.

2 Back-propagation and differentiation in the reverse order

Postorder is evidently not the correct traversal strategy for adjoint propagation, nor are other classic traversal strategies like preorder or breadth-first, discussed in the chapter 9 of [5] and not reproduced here. We want to traverse every node exactly once. We need a traversal strategy with guarantee of a complete and correct accumulation of adjoints after a single visit to every node. Visiting a node propagates its adjoints to its ancestors. Hence, every node must have completed accumulating its own adjoint *before* it is visited. In other terms, *all* the successors of a node must be visited *before* the node. By contrast, postorder is a traversal strategy where the *ancestors* are visited first.

The particular traversal order we need, where parent nodes (successors) are visited before child nodes (ancestors), is well known to graph theory. It is called the *topological order*. In general, it takes specialized algorithms to sort a graph topologically, and such algorithms consume precious run time. But, for DAGs, we have a simple result:

The topological order for a DAG is the reverse of its postorder.

This is simple enough, and the demonstration is intuitive: postorder corresponds to the evaluation order, where arguments to operations (ancestors) are visited before operations (successors). Hence, in its reverse order, all the successors (operations) of a node (argument) are visited before the node. In addition, postorder visits each node once, hence, so does the reverse. Therefore, the reverse postorder *is* the topological order.[2]

The correct order of traversal in our simple example from the first part is simply: 13, 12, 11,..., 1. More generally, the adjoint algorithm is executed on the exact sequence of

The particular traversal order we need, where parent nodes (successors) are visited before child nodes (ancestors), is well known to graph theory. It is called the *topological order*

operations in the calculation, in the reverse order. This realization affects design considerations explored in the next section. For now, let us step back and reflect on what we realized.

A computation can be differentiated in constant time by reversing the order of its operations.

This may sound like magic.[3] Let us try to make intuitive and mathematical sense of how this is possible.

First, notice that, although AAD is different from any form of symbolic differentiation, it is still analytical. Even though the original calculation may be a numerical method like a Monte-Carlo simulation or a Finite Difference Method, the extraction of its graph turns it into a series of mathematical operations, hence, an *analytic calculation*. To perform adjoint propagation over the graph is similar to an explicit differentiation of the sequence of operations through the chain rule:

$$y = f(g(h(x))) \Longrightarrow\Leftarrow \frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

To evaluate y, we must first evaluate h, then g, then f. But, to differentiate it analytically, we would generally first differentiate f, then g, then h. Think of a concrete example, like the following one, suggested by Peter Caspers of Quaternion Risk Management in `https://linkedin.com/feed/update/urn:li:activity:6488818005642866689`:

$$z = e^{x^2+y^2}$$

To write the gradient of z, one would first compute $\frac{\partial z_i}{\partial x^2+y^2}$ (which is $e^{x^2+y^2}$), then multiply by $2x$ to find $\frac{\partial z}{\partial x}$ and by $2y$ to find $\frac{\partial z}{\partial y}$. This is nothing else than an intuitive invocation of the adjoint model, in an order reversed from evaluation (where one would first compute x^2 and y^2, then the sum, then the exponential). To differentiate z in the direct order would be both awkward and inefficient, with the exponential evaluated twice.

Another intuitive explanation of how reversing the order of operations cuts complexity by an order of magnitude, due to Brian Huge of Danske Bank [3], makes an analogy with matrix calculus. Consider the matrix product:

$$M_1M_2 \dots M_n\,V$$

where M_i are matrices and V is a vector. The product results in a vector of dimension the number of rows in M_1. To perform the product left to right results in a series of matrix by matrix products, of cubic complexity, each resulting in a matrix, so that the next product is also matrix by matrix. Only the last operation, the product of the matrix $M_1M_2 \dots M_n$ resulting from the n first products, by the vector V, is a matrix by vector product of quadratic complexity, producing the final vector result.

When the same product is performed *right to left*, we conduct a series of matrix *by vector* products, of *quadratic* complexity, each resulting in a vector, ensuring that the next operation is also a matrix by vector product, also of quadratic complexity. Hence, to compute the product right to left produces the same result, *an order of magnitude faster*. Now, consider a calculation consisting in a sequence of vector to vector functions. Provided the calculation is scalar, its final result is a real number, so the last operation in the sequence must be a vector to scalar function. The differentials of the final result to the initial (first) vector are given by the product of the Jacobian matrices of the sequence of vector to vector functions. The Jacobian matrix of the last calculation is, in fact, a vector, since the calculation is scalar. Hence, the computation of the differentials boils down

The relationship of back-prop to AD directly flows from the fact that *any* calculation may be represented as a sequence of vector to vector functions, hence, a neural network

to a matrix product of the form above. We can evaluate it left to right, in the sequence of the calculation, or right to left, in the reverse order, an order of magnitude faster. We effectively decrease complexity by an order just by reversing the order of operations for differentiation. Note that this only works when the rightmost term in the matrix product is a vector, and that this is only the case when the calculation is scalar. This is why only scalar functions are differentiated in constant time.

3 Case study: back-propagation and Deep Learning

We now develop the ideas of the previous section, in the important context of artificial neural networks (ANNs), and re-derive a well known, highly efficient differentiation algorithm for the cost function of neural nets, called *back-propagation*. The relationship of back-prop to AD directly flows from the fact that *any* calculation may be represented as a sequence of vector to vector functions, hence, a neural network. When the calculation is scalar, the last operation in its sequence is a vector to scalar function.

Let us briefly recall the equations that define a neural net. Readers will find details in a reference textbook like [4]. A neural net consists in a series of L vectors (called *layers*) computed in a sequence 1 to L with the *feed-forward* equations:

$$a_{l+1} = \varphi_{l+1}(a_l; \vartheta_{l+1})$$

where $a_0 \in \mathbb{R}^{n_0}$ are the inputs, $a_L \in \mathbb{R}^{n_L}$ are the outputs and the other $a_l \in \mathbb{R}^{n_l}$ are the *hidden layers*. The feed-forward functions $\varphi_{l+1}(a_l; \vartheta_{l+1})$ are known from the onset, as part of the *architecture* of the network. Their purpose is to compute a layer as a function of the previous layer, subject to parameters ϑ_l. Those parameters are ultimately optimized to *train* the network to best fit a *training dataset*.

In most applications, the feed-forward equations are of the form:

$$a_{l+1} = g_{l+1}(W_{l+1} a_l + b_{l+1})$$

where the g_l are scalar, non-linear *activation* functions, applied element-wise,[4] W_{l+1} is a matrix in dimension n_{l+1} by n_l and b_{l+1} is a vector in dimension n_{l+1}. Hence, every layer is an activated linear combination of the previous layer. This form, known as a *multi-layer perceptron* or MLP, has mathematical and computational benefits. The main mathematical benefit is known as the *Universal Representation Theorem*, which states that (asymptotically in the number n_l of units) the network can represent *any* function (with mild technical conditions) and encode it in its weights W_l and b_l. The major computational benefit is that the network consists in a sequence of matrix operations (matrix by vector product, followed by vector addition, followed by the element-wise application of a scalar function, repeated for the sequence of layers), which are performed optimally on modern hardware, particularly GPUs and TPUs.

The weights ϑ_l (for perceptrons, W_l and b_l) are set when the network is trained, but the shape (definition) of the feed-forward functions φ_l are known, together with their *Jacobian matrices*:

$$\frac{\partial \varphi_l + 1}{\partial a_l}$$

in dimension n_l by n_{l+1}, applying the convention where Jacobian matrices have their derivatives in rows and functions in columns, and:

$$\frac{\partial \varphi_l + 1}{\partial \vartheta_l + 1}$$

in dimension the number of parameters for layer $l + 1$, $dim\,(\vartheta_{l+1})$ by n_{l+1}. In the case of a perceptron, we have:

The derivative of the loss to the output layer is (twice) the prediction error, which is why the back-prop algorithm is sometimes called 'error back-propagation'

$$\frac{\partial \varphi_{l+1}}{\partial a_l} = W_{l+1}^T \otimes g_l'(W_{l+1} a_l + b_{l+1})$$

where $\otimes$ denotes the multiplication of the n_{l+1} columns of the matrix W_{l+1}^T by the n_{l+1} entries of the vector $g_l'(W_{l+1} a_l + b_{l+1})$ of the derivatives of the activation function;

$$\frac{\partial \varphi_{l+1}}{\partial W_{l+1}} = a_l$$

meaning that, element-wise:

$$\frac{\partial \varphi_{l+1}[i]}{\partial W_{l+1}[i,j]} = a_l[j]$$

and:

$$\frac{\partial \varphi_{l+1}}{\partial b_{l+1}} = g_l'(W_{l+1} a_l + b_{l+1})$$

The performance of a neural network is measured by comparison of its output a_L, given a training input a_0, with the corresponding training *label* y. This is often called the *loss* function:

$$c(a_L, y)$$

The most commonly used loss functions are the *categorical cross-entropy* for classification problems, or simply the (squared) distance of a_L to y for regression problems. In all cases, the loss function c is a known, scalar function of a_L and the loss on a training dataset, or a subset of it, is simply the sum of the losses of individual examples. Its gradient $\frac{\partial c}{\partial a_L}$ is also a known vector function in dimension n_L. In simple regression problems where a_L is a real number ($n_L = 1$), $c(a_L, y) = (a_L, y)^2$ and $\frac{\partial c}{\partial a_L} = 2(a_L - y)$. The derivative of the loss to the output layer is (twice) the prediction error, which is why the back-prop algorithm is sometimes called 'error back-propagation'.

To train the network means find its parameters ϑ_l that minimize the loss of the training dataset. This is a non-convex problem. There is no analytic solution, and no known iterative algorithm guaranteed to converge to a global minimum. However, its has been repeatedly demonstrated (empirically) that variations of the simple gradient descent algorithm do converge to a satisfactory minimum in a vast number of contexts. The gradient descent algorithm repeatedly modifies parameters in the direction of steepest descent of the cost function, given its gradient to the weights of the network:

$$\vartheta_l \leftarrow \vartheta_l - \alpha \frac{\partial c}{\partial \vartheta_l}$$

on every iteration of the descent, for all layers l, where α is (in the simplest implementation of gradient descent) a constant called *learning rate*.

Therefore, the key to training neural nets in reasonable time is a fast computation of the gradients $\frac{\partial c}{\partial \vartheta_l}$. Immediately from the chain rule, we have:

$$\frac{\partial c}{\partial \vartheta_l} = \frac{\partial a_l}{\partial \vartheta_l} \frac{\partial a_{l+1}}{\partial a_l} \cdots \frac{\partial a_{L-1}}{\partial a_{L-2}} \frac{\partial a_L}{\partial a_{L-1}} \frac{\vartheta c}{\partial a_L}$$

where we recall that we write Jacobians with derivatives in rows, just like gradients. Looking attentively into this equation, we see that:

1. The gradient $\frac{\partial c}{\partial \vartheta_l}$ is the result of the product of multiple terms, matrices and vectors, and we know exactly how to calculate each of these terms, as previously discussed. The computation of the desired gradient therefore boils down to performing a matrix product.
2. The leftmost term $\frac{\partial a_l}{\partial \vartheta_l}$ is the Jacobian matrix of activations on layer l to the parameters for that layer. The rightmost term is the gradient *vector* of the *scalar* loss to the out-

>

put layer. The inner terms are the Jacobian matrices of successive layers to the previous layer, written left to right, layer $l+1$ to layer l, following the sequence of feed-forward equations.

3. The inner terms in the product are therefore Jacobian *matrices* of vector to vector functions φ_l. The rightmost term is the gradient *vector* of the *vector to scalar* function c.

We have seen that a computation of this matrix product, left to right, in the order of layers 1 to L, is of cubic complexity, whereas a computation right to left, in the reverse order of layers L to 1, produces the same result with quadratic complexity.

This is why the differentials of the cost function in a neural network are always computed in the reverse order to the feed-forward equations, an order of magnitude faster. This way of computing gradients of the cost function is called *back-propagation*, it constitutes the backbone of Deep Learning, and the key ingredient in its recent and spectacular successes in the fields of computer vision or natural language processing. To train the gigantic neural nets involved in these fields without back-prop would take years on the most advanced computers available today.

4 Automatic differentiation with tapes

We have discussed in length the main ideas and techniques involved in AAD: the extraction of the computation graph at run time with operator overloading, and its traversal in the reverse order to implement adjoint propagation and differentiate the calculation analytically, in constant time. At this point, we can bring the pieces together and build a simplistic, yet fully functional AAD framework. Its code is found in `https://github.com/asavine/WilmottArticle/aad.h`

4.1 Nodes and tape

Our graph design from part I is not ideal for the purpose of differentiation. In that design, child nodes were held by shared pointers on the parent nodes. It was convenient enough to build the graph with overloading, and traverse it in postorder, for example, for the purpose of evaluation or reverse engineering. But it could not easily or efficiently be traversed in the topological order, for the purpose of adjoint differentiation.

We have seen that the topological order is the reverse postorder, which means that the sequence of operations involved in the calculation is differentiated in the reverse order of its evaluation. We will build a sequence of nodes while executing operations, and run adjoint propagation in the reverse order on this sequence. A natural data structure to hold a sequence of objects for left to right or right to left iteration is the vector:

```
// The tape, declared as a global variable
vector<Node> tape;
```

Note that a vector guarantees memory coalescence and offers convenient and efficient traversal left to right or right to left. But it is not the most efficient data structure for computation graphs. We don't know in advance the number of nodes in a computation graph. As we build the sequence of nodes at run time, the vector fills out. A new, bigger block of memory must be allocated and the contents of the vector must be copied there, before the old, insufficient memory is released. In the chapter 10 of [5], we build custom data structures, which offer the convenience and coalescence of vectors, while optimizing memory management and prevent copies of existing data. In this simple implementation, we stick with a vector.

To make the tape a global variable is also a questionable choice. Among other problems, this means that the tape cannot be accessed by multiple threads concurrently without a complicated and costly locking system. Chapters 10 and 12 of [5] explain in length the concurrent implementation of AAD.

In part I, nodes were represented in a polymorphic class. In general, the nodes of a computation graph must contain information about the operation they represent (addition, multiplication, logarithm...) and the location of its arguments. For the purpose of AAD, the identification of the operation is only needed to compute local derivatives at back-propagation time. Alternatively, we may compute these derivatives *eagerly* as we build the graph, and store them on the node. In this case, we no longer need to identify the operation on the node. This simplifies the code and saves virtual function calls.

Since an operation may have zero, one or two arguments, we provide space for two indices to locate the arguments on tape, two *doubles* to hold the derivatives of the operation to its arguments, and an *int* giving the effective number of arguments. A professional implementation, such as the companion code to [5], would implement a mechanism to avoid wasting memory this way, but this is good enough for our purpose:

```
struct Node
{
    int      numArg;         // number of arguments: 0, 1 or 2
    int      idx1;          // index of first argument on tape
    int      idx2;          // index of second argument on tape
    double   der1;          // partial derivative to first argument
    double   der2;          // partial derivative to second argument
};
```

4.2 Custom number type

Next, we redefine our custom *Number* type to match our new graph design. Our *Number* holds its value, and the index of its node on tape. It also builds a leaf on tape on initialization:

```
struct Number
{
    double value;
    int    idx;

    // default constructor does nothing
    Number() {}

    // constructs with a value and record
    Number(const double& x): value(x)
    {
        // create a new record on tape
        tape.push_back(Node());
        Node& node = tape.back();

        // reference record on tape
        idx = tape.size() - 1;

        // populate record on tape
        node.numArg = 0;
    }
};
```

4.3 Operator overloading

As before, we overload all mathematical operations and functions for our number type, to build the tape during the evaluation of a calculation. Our overloads evaluate the operation and store its result, just like the standard overloads, but, in addition, they create and populate the corresponding node on tape, including the indices of the arguments, and, crucially, the derivatives of the operation to the arguments.

This is best seen through a couple of examples, like the operator + below:

```
Number operator+(const Number& lhs, const Number& rhs)
{
    // create a new record on tape
    tape.push_back(Node());
    Node& node = tape.back();

    // compute result
    Number result;
    result.value = lhs.value + rhs.value; // calling double overload

    // reference record on tape
    result.idx = tape.size() - 1;

    // populate record on tape
    node.numArg = 2;
    node.idx1 = lhs.idx;
```

>

```
17     node.idx2 = rhs.idx;
18
19     // compute derivatives
20     node.der1 = 1;
21     node.der2 = 1;
22
23     return result;
24 }
```

The overloads of all the other binary operators are identical, safe for the computation of the value on line 9 and derivatives on line 20. For instance, the operator * is defined as follows:

```
1 Number operator*(const Number& lhs, const Number& rhs)
2 {
3     // create a new record on tape
4     tape.push_back(Node());
5     Node& node = tape.back();
6
7     // compute result
8     Number result;
9     result.value = lhs.value * rhs.value; // Different value here
10
11     // reference record on tape
12     result.idx = tape.size() - 1;
13
14     // populate record on tape
15     node.numArg = 2;
16     node.idx1 = lhs.idx;
17     node.idx2 = rhs.idx;
18
19     // compute derivatives
20     node.der1 = rhs.value; // Different derivative here
21     node.der2 = lhs.value; // Different derivative here
22
23     return result;
24  }
```

The unary operator and function overloads are defined similarly:

```
1 Number log(const Number& arg)
2 {
3     // create a new record on tape
4     tape.push_back(Node());
5     Node& node = tape.back();
6
7     // compute result
8     Number result;
9     result.value = log(arg.value);
10
```

```
    // reference record on tape
    result.idx = tape.size() - 1;

    // populate record on tape
    node.numArg = 1;
    node.idx1 = arg.idx;

    // compute derivative
    node.der1 = 1.0 / arg.value;

    return result;
}
```

The code in `https://github.com/asavine/WilmottArticle/aad.h` defines overloads for the most useful operators and functions, including Gaussian density and cumulative distribution defined in https://github.com/asavine/WilmottArticle/gaussians.h. For instance, the overload for the cumulative Gaussian, which derivative is (trivially) the Gaussian density, is given below:

```
Number normalCdf(const Number& arg)
{
    // create a new record on tape
    tape.push_back(Node());
    Node& node = tape.back();

    // compute result
    Number result;
    result.value = normalCdf(arg.value);

    // reference record on tape
    result.idx = tape.size() - 1;

    // populate record on tape
    node.numArg = 1;
    node.idx1 = arg.idx;

    // compute derivative
    node.der1 = normalDens(arg.value);

    return result;
 }
```

The logic and code are identically repeated in the definition of all the overloads, safe for the computation of value and derivatives. This is poor coding practice. The professional AAD library developed in the chapter 15 of [5] applies policy-based design, a powerful paradigm promoted by Alexandrescu in [1], to avoid code duplication without run time penalty.

We must also provide overloads for comparison operators to avoid compile errors when *Numbers* are compared:

```
bool operator==(const Number& lhs, const Number& rhs)
    { return lhs.value == rhs.value;}
bool operator!=(const Number& lhs, const Number& rhs)
    { return lhs.value != rhs.value;}
bool operator>(const Number& lhs, const Number& rhs)
```

>

```
    { return lhs.value > rhs.value;}
bool operator>=(const Number& lhs, const Number& rhs)
    { return lhs.value >= rhs.value;}
bool operator<(const Number& lhs, const Number& rhs)
    { return lhs.value < rhs.value;}
bool operator<=(const Number& lhs, const Number& rhs)
    { return lhs.value <= rhs.value;}
```

Finally, calculation code often applies shortcuts like + =, * = and friends, who must be overloaded on class. The complete definition *Number* class is therefore:

```
struct Number
{
    double value;
    int    idx;

    // default constructor does nothing
    Number() {}

    // constructs with a value and record
    Number(const double& x): value(x)
 {
    // create a new record on tape
     tape.push_back(Record());
    Node& node = tape.back();

    // reference record on tape
    idx = tape.size() - 1;

    // populate record on tape
    node.numArg = 0;}
 }

    Number operator +() const { return *this;}
    Number operator -() const { return Number(0.0) - *this;}

    Number& operator +=(const Number& rhs)
           { *this = *this + rhs; return *this;}
    Number& operator -=(const Number& rhs)
           { *this = *this - rhs; return *this;}
    Number& operator *=(const Number& rhs)
           { *this = *this * rhs; return *this;}
    Number& operator /=(const Number& rhs)
           { *this = *this / rhs; return *this;}
 };
```

Depending on taste, it may also be more convenient to implement all the overloaded operators and functions as *friends* on the *Number* class (which is how we wrote the file aad.h on the GitHub repo for the article).

Our code does not implement *all* the standard operators and functions, which is an exhausting task: we should implement overloads for all functions, down to hyperbolic sine, cosine, tangent and the like. Thankfully, when instrumented code uses functions without an overload in the framework, the code will not compile, and additional overloads may be implemented when this happens. The absence of overload will *not* cause silent bugs.

4.4 Adjoint back-propagation

What we have built so far permits to record the execution of calculation code on tape. More precisely, the tape captures the sequence of operations (without explicitly identifying them), together with their ancestors and local derivatives.

In ord–––er to record calculation code, it is necessary to template the code on its number representation type, and call it with our custom *Number* as the template argument, as previously explained.

After the execution of the instrumented calculation code is recorded, we may execute the adjoint propagation algorithm, in the reverse order through the tape:

```
vector<double> calculateAdjoints(Number& result)
{
    // initialization
    vector<double> adjoints(tape.size(), 0.0);     // initialize all to 0
    int N = result.idx;                            // find N
    adjoints[N] = 1.0;                             // seed aN = 1

    // backward propagation
    for(int j=N; j>0; --j) // iterate backwards over tape
    {
        if (tape[j].numArg > 0)
        {
            // propagate first argument
            adjoints[tape[j].idx1] += adjoints[j] * tape[j].der1;
            if (tape[j].numArg > 1)
            {
                // propagate second argument
                adjoints[tape[j].idx2] += adjoints[j] * tape[j].der2;
            }
        }
    }

    return adjoints;
}
```

This code is a literal implementation of the adjoint algorithm, and should be self explanatory. Instead of assuming that the final result is the last entry on tape, we pass a reference to the result and extract its index on tape from there.

4.5 Applying the framework

The first step to differentiate a calculation code with AAD is template it. As previously explained, code like this one:

```
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 = normalCdf(d2);
    return df * (fwd * p1 - strike * p2);
}
```

would become:

```
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 code is also available on `https://github.com/asavine/WilmottArticle/` in the file `BlackScholes.h`. It is particularly simple and self contained. Actual calculation code may involve many nested function calls, data structures and algorithms. All of these must be templated for the real number representation type. Code that is not templated (not *instrumented* in AAD lingo) would not be recorded on tape and its adjoints would not be propagated. Intentional selective instrumentation may be a strong optimization, as explained in the chapter 12 of [5]. But unintentional failure to instrument part of the code leads to wrong derivatives.

The rest is straightforward: call instrumented code to record the tape, propagate adjoints, return desired derivatives and clear memory:

```
int main()
{
    // initializes and records inputs
    Number spot = 100,
           rate = 0.02,
           yield = 0.05,
           vol = 0.2,
           strike = 110,
           mat = 2;
    // evaluates and records operations
    auto result = blackScholes(spot, rate, yield, vol, strike, mat);
    cout << "Value = " << result.value << endl; // 5.03705

    // propagate adjoints
    vector<double> adjoints = calculateAdjoints(result);

    // show derivatives
    cout << "Derivative to spot (delta) = "
           << adjoints[spot.idx] << endl;
    // 0.309
    cout << "Derivative to rate (rho) = "
           << adjoints[rate.idx] << endl;
    // 51.772
    cout << "Derivative to dividend yield = "
           << adjoints[yield.idx] << endl;
     // -61.846
    cout << "Derivative to volatility (vega) = "
           << adjoints[vol.idx] << endl;
     // 46.980
    cout << "Derivative to strike (-digital) = "
           << adjoints[strike.idx] << endl;
    // -0.235
    cout << "Derivative to maturity (-theta) = "
           << adjoints[mat.idx] << endl;
    // 1.321

    // clear
    tape.clear();
}
```

Conclusion

The preceding example is obviously not so interesting as it differentiates the Black and Scholes formula, a self-contained, analytical code, executing in a matter of microseconds and subject to only six inputs.

What is probably more interesting is that the simplistic framework we just developed is able to differentiate much more complicated code, like the value of a barrier option in a local volatility model a la Dupire [2] implemented in terms of Monte-Carlo simulations. In this case, one evaluation of the pricing code takes considerable time and hundreds to thousands of derivatives to local volatilities would take minutes to hours with conventional differentiation. Despite its simplicity and inefficiency, our framework produces all differentials in around six times the time of one pricing. With 100,000 paths, 156 time steps, we computed 1,081 differentials in around 7 seconds on a laptop. This is a remarkable achievement, although the professional code of [5] differentiates this example in a fraction of a second on a quad-code laptop. This is all explained in detail in the next, and last part of this series dedicated to AAD.

About the Author

Antoine Savine is a French mathematician and risk management practitioner with Superfly Analytics in Danske Bank, winner of the RiskMinds 2019 award for Excellence in Risk Management and Modelling. He has held multiple leading roles in the derivatives industry in the past 20 years, and presently also teaches Volatility and Computational Finance at Copenhagen University.

ENDNOTES

1. An alternative solution is to compute the partial derivatives eagerly during evaluation and store them on the parent node. Once the parent node caches all its partial derivatives, the intermediate results are no longer needed for back-propagation. This solution is more efficient, and the one we eventually implement.
2. Actually, this proves that reverse postorder is a topological order but let us put uniqueness considerations aside.
3. Indeed, the spectacular speed of AAD, without loss of accuracy, often looks like magic when witnessed for the first time
4. The most common activation functions are the sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ and relu . $relu(x) = \max(x,0)$

REFERENCES

1. Alexandrescu, A. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.
2. Dupire, B. 1994. Pricing with a smile. *Risk* 7(1):18–20.
3. Flyger, H.-J., Huge, B, and Savine, A. 2015. Practical implementation of AAD for derivatives risk management, xVA and RWA. Global Derivatives.
4. Goodfellow, I., Bengio, Y., and Courville, A. 2016. A. Deep Learning. MIT Press.
5. Savine, A. 2018. *Modern Computational Finance, Volume 1: AAD and Parallel Simulations*. Wiley.