ESAIM: PROCEEDINGS AND SURVEYS, September 2017, Vol. 59, p. 56-75
B. Bouchard, E. Gobet and B. Jourdain, Editors

MINI-SYMPOSIUM ON AUTOMATIC DIFFERENTIATION AND ITS APPLICATIONS IN THE FINANCIAL INDUSTRY

Sébastien Geeraert[1], Charles-Albert Lehalle[2], Barak A. Pearlmutter[3], Olivier Pironneau[4] and Adil Reghai[5]

Abstract. Automatic differentiation has been involved for long in applied mathematics as an alternative to finite difference to improve the accuracy of numerical computation of derivatives. Each time a numerical minimization is involved, automatic differentiation can be used. In between formal derivation and standard numerical schemes, this approach is based on software solutions applying mechanically the chain rule formula to obtain an exact value for the desired derivative. It has a cost in memory and cpu consumption.

For participants of financial markets (banks, insurances, financial intermediaries, etc), computing derivatives is needed to obtain the sensitivity of their exposure to well-defined potential market moves. It is a way to understand variations of their balance sheets in specific cases. Since the 2008 crisis, regulation demands to compute this kind of exposure to many different cases, to be sure that market participants are aware and ready to face a wide spectrum of market configurations.

This paper shows how automatic differentiation provides a partial answer to this recent explosion of computations to be performed. One part of the answer is a straightforward application of Adjoint Algorithmic Differentiation (AAD), but it is not enough. Since financial sensitivities involve specific functions and mix differentiation with Monte-Carlo simulations, dedicated tools and associated theoretical results are needed. We give here short introductions to typical cases arising when one uses AAD on financial markets.

1. Introduction

Numerous discussions with professionals underlined the recent importance of automatic differentiation in financial institutions. It hence appeared natural to include a mini-symposium on this topic in the "International Conference on Monte Carlo techniques".

One of the roots of the recent interest of market participants for automatic differentiation is the ever demanding regulations on *sensitivity* computations. For instance, Section B.2 and B.3 of Basel Committee on Banking Supervision's *Minimum capital requirements for market risk* [2] contain a long list of sensitivities to be computed by investment banks on every product they own. Sensitivities have then to be aggregated according to a regulatory formula to obtain the capital requirement the bank has to put in front of its risks. Banks own financial products in their books, and the sensitivity of a financial product is the variation of its value with

[1] Murex (Paris, France).

[2] Capital Fund Management (Paris, France) and Imperial College (London, U.K.).

[3] Department of Computer Science, Maynooth University, Co. Kildare, Ireland.

[4] Sorbonne Universités, LJLL, UMR 7598, Case 187, 4 pl. de Jussieu, F-75252 Paris Cedex 5, France.

[5] Natixis (Paris, France).

respect to a well defined variation of the market context. For instance, [2, Section B.2.iii.] demands to compute *delta*, *vega* and *curvature* for "*each financial product with optionality*" owned by a bank. These sensitivities are partial derivatives of the value of the instrument with respect to different *risk factors*, grouped into *risk classes*. Seven risk classes are defined. As an example, the risk factors of the *General Interest Rate Risk* class are [2, pp20-22] :

- for the *delta*: "delta risk factors are defined along two dimensions: a risk-free yield curve for each currency in which interest rate-sensitive instruments are denominated and" ten maturities.
- For the *vega*: "within each currency, the vega risk factors are the implied volatilities of options that reference general interest rate risk-sensitive underlyings; further defined along two dimensions: the *Maturity of the option at the expiry date of the option* and the *Residual maturity of the underlying of the option at the expiry date of the option*." Each of them over 5 maturities.

We just listed more than 30 sensitivities to compute, belonging to one of the seven risk classes, to be applied on each financial instrument owned by the bank. Each sensitivity is a partial derivative.

Before this increase in regulatory demand, banks restricted computed sensitivities (i.e. partial derivatives) needed to hedge their portfolios, and almost no more. They used finite differences methods in the "few" needed directions.

The sudden increase of partial derivatives to be computed motivated the exploration of other ways to compute them. Moreover, since regulatory updates continue to increase the list of the partial derivatives to compute, banks tried to identify methods allowing to compute new partial derivatives without changing their software code too much.

With automatic differentiation (or adjoint algorithmic differentiation: AAD), you have not to develop a dedicated code each time you have a new product on which you need to compute sensitivities. It is "enough" to embed your overall numerical libraries in a framework supporting AAD (see Section 3). Sessions of this mini-symposium (and hence sections of this paper) show that it has an overall cost, but once this cost is paid, the marginal cost of computing more sensitivities is small. ADD has hence been considered as one of the solutions to current regulatory demand and to be prepared to answer to future ones at a reasonable cost. It explains why banks are carefully looking at ADD or are already using it to some extent.

If AAD exists for long and has numerous applications (see [4] for a survey on interactions of AAD and statistical learning), the specificities of financial payoffs need some dedicated developments, as in [27] (see Section 3). Moreover, the reformulation of some financial problems (like Credit Value Adjustement, see Section 5) helps their inclusion in an AAD framework.

The sections of this paper reflect the sessions of the mini-symposium: each Section has been written by a speaker and contains a "take home message" about a specific aspect of AAD useful to use it on financial markets. The list of references at the end of the paper will help the reader to go deeper in this field.

In the first section, Barak Pearlmutter gives an overview of the most important aspects of AAD, in general. The reader can refer to [4] for details. This section underlines the role of AAD in applied mathematics: progresses in numerical optimization, as a balance between manual derivation of specific functions involved in the criterion to minimize, and a pure numerical scheme via finite differences. Hence domains like non linear statistics, and more recently machine learning[1], took profit of progresses in AAD.

In Section 3, Sébastien Geeraert emphasizes some practical aspects he explored at Murex. He underlines how the specific mix of deterministic and stochastic processes involved in the payoffs of financial contracts can be exploited thanks to ad hoc choices to improve the efficiency of AAD. Typically ADD has to be used inside Monte-Carlo simulations, giving birth to specific constraints in terms of memory and cpu consumption.

In the next section, Olivier Pironneau put the emphasis on ways to apply AAD on non differentiable functions. This section follows the idea of applying the Vibrato method to a non differentiable function via smoothing. It allows to obtain accurate estimates for partial derivatives of order two in some cases.

[1]Well-known machine learning frameworks, like Theano or TensorFlow, use AAD.

Adil Reghai exposes in the last section how Natixis uses AAD in production to compute sensitivities of its exposure to potential market moves. It demonstrates the advantage of AAD in the scope of the specific case of sensitivity to the counterparty risk.

2. What Automatic Differentiation Means to Me

by *Barak A. Pearlmutter*

We briefly (and from a self-serving biased and revisionist perspective) survey the basic ideas and intuitions of Automatic Differentiation: the efficient mechanical accurate calculation of derivatives of functions specified by computer programs.

2.1. Introduction

Automatic Differentiation or AD [20] is a subdiscipline of numeric computation dedicated to the study of an enticing idea: the mechanical transformation of a computer program that computes a numeric function into an augmented computer program that also computes some desired derivatives accurately and efficiently. The intuition for how this might be accomplished can be formed by considering a computer program implementing the function f which maps an input vector to an output vector

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

as consisting of a *flow graph*: a directed acyclic graph leading from inputs to outputs whose edges carry $\mathbb{R}$ values and whose nodes represent primitive arithmetic operations or fan-out. This flow graph is the natural “split” of the implementation of a computation in the usual atomic operators provided by the considered programming language. Automatic differentiation operates at the level of this graph, extending it to simultaneously computing the desired value, and the first term of its Taylor expansion.

2.2. Forward AD

Using modern terminology, [40] proposed that each real number r in the flow graph could be replaced by a “dual number” [12], a pair of real numbers $\langle r, r'\rangle$ formally representing a truncated power series $r + r'\epsilon + O(\epsilon^2)$. It will formally allow to carry not only operations on r (and thus provide the standard operation), but on r' too (providing the derivative in the direction of r). Each operation in the dual space is made of a pair of operations: the standard one and its derivative. This means replacing each primitive arithmetic operation g:

$$(v_1, \ldots, v_m) = g(u_1, \ldots, u_n)$$

by its modification Fg, where the operator F is a way to formalize we replaced u by $\langle v, v'\rangle$ in the expression of g:

$$(\langle v_1, v_1'\rangle, \ldots, \langle v_m, v_m'\rangle) = Fg(\langle u_1, u_1'\rangle, \ldots, \langle u_n, u_n'\rangle)$$

where

$$\langle v_i, v_i'\rangle = Fg_i(\langle u_1, u_1'\rangle, \ldots, \langle u_n, u_n'\rangle) = \langle g_i(u_1, \ldots, u_n), \sum_j u_j'(d/du_j)g_i(u_1, \ldots, u_n)\rangle.$$

Figure 1 shows the transformation of two basic operations (the upper box figures the flow graph for a multiplication and sin or cos) into their dual representation.

Or, gathering values into vectors $\mathbf{u} = (u_1, \ldots, u_n)$, etc, and letting $J_g(\mathbf{u})$ be the Jacobian matrix of g at $\mathbf{u}$, and gathering the primed values into vectors as well,

$$\mathbf{v} = g(\mathbf{u}) \qquad \mathbf{v}' = J_g(\mathbf{u})\mathbf{u}' \qquad \langle \mathbf{v}, \mathbf{v}'\rangle = Fg(\langle \mathbf{u}, \mathbf{u}'\rangle) = \langle g(\mathbf{u}), J_g(\mathbf{u})\mathbf{u}'\rangle$$

It is now clear the dual space representation allows to simultaneously carry an operation and its derivative.

Assuming we have replaced each primitive arithmetic operation by an operation which does what the original operation did but also performs a Jacobian-vector product, as above, the entire computation will now perform a Jacobian-vector product. We exhibit a small example of this transformation in Figure 1, which graphically represents a procedure:

```
function fig(double a, double b, double u) {
    double c = a * b;
    double[] (v,w) = sincos(u);
    return (c,v,w);
}
```

which is transformed into:

```
function Ffig(double a, double da, double b, double db, double u, double du) {
    double c = a * b;
    double dc = a * db + da * b;
    double[] (v,w) = sincos(u);
    double dv = w * du;
    double dw = - v * du;
    return (c,dc,v,dv,w,dw);
}
```

or, if we bundle each primal variable `v` with `dv` into a structure `(v,dv)` stored in `Fv`, and for each primal subroutine or operator `fig` we use `Ffig` which calculates the primal but also propagated derivatives:

```
function Ffig(double[] Fa, double[] Fb, double[] Fu) {
    double[] Fc = fTimes(Fa,Fb);
    double[][] (Fv,Fw) = Fsincos(Fu);
    return (Fc,Fv,Fw);
}.
```

Thanks to the implementation `Ffig` of the representation of routine `fig` in the dual space, it is now straightforward to calculate the full Jacobian by calling this $n = 3$ times, each time calculating a single column of the Jacobian. It is enough to replace the second element of the dual variables by 0 or 1:

```
((c,j11),(v,j12),(w,j13)) = Ffig((a,1),(b,0),(u,0));
((c,j21),(v,j22),(w,j23)) = Ffig((a,0),(b,1),(u,0));
((c,j31),(v,j32),(w,j33)) = Ffig((a,0),(b,0),(u,1));
```

This is "Forward Accumulation Mode Automatic Differentiation", or "Forward AD". Note that since n and m are very small for primitive operations (one or two in general), the extra arithmetic added is a small constant factor overhead. And note that we can allow the new "prime" values to flow through the computation in parallel with the original ones, meaning that the storage overhead is bounded by a factor of two.

The same mathematical transformation as Forward AD has many names in many fields: propagation of perturbations in machine learning, the pushforward in differential geometry, a directional derivative in multivariate calculus, forward error analysis in computer science, etc. And many software tools exist that perform the Forward AD transformation, using a variety of implementation strategies ranging from source-to-source transformation (generally the fastest, but the least flexible or convenient) to overloading of operators in an object-oriented system (easy to implement and use but usually quite slow and sometimes not very robust due to semantic limitations of the overloading mechanisms.)

2.3. Reverse AD

Consider an attempt to calculate the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$ using Forward AD. This would require running the transformed function $\overrightarrow{\mathcal{J}}f : \langle \mathbf{x}, \mathbf{x}' \rangle \mapsto \langle \mathbf{y}, \mathbf{y}' \rangle$ repeatedly, with $\mathbf{x}'$ being cycled through the n basis vectors $(1, 0, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 0, 1)$. When n is large (say, 10^6) this overhead is unacceptable.

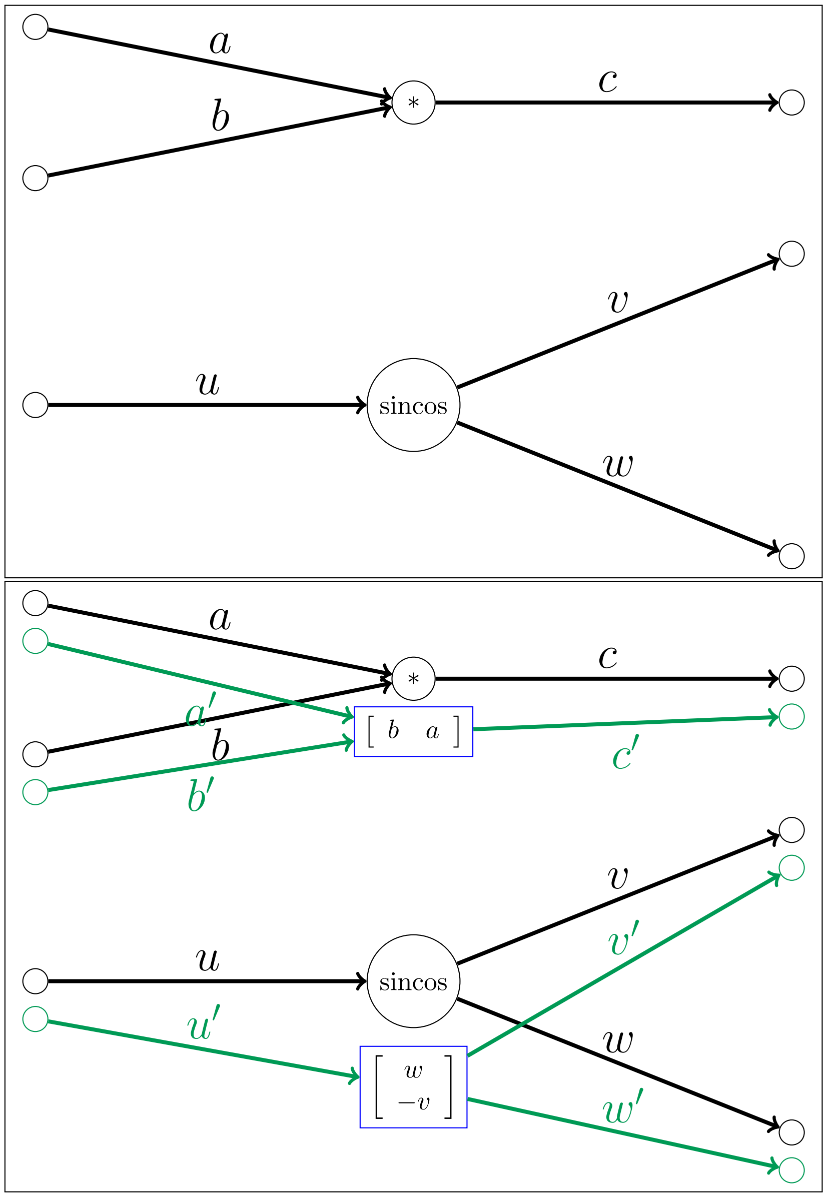

Figure 1. Forward AD augments all primitive arithmetic operations in the original flow graph (top) to also propagate derivatives forward through the computation (bottom).

Fortunately there is a different AD transformation which can calculate gradients more efficiently. Each value v in the original flow graph is augmented with an adjoint value $\bar{v}$, which are propagated backwards through the flow graph. Each primitive arithmetic operation is modified (see Figure 2) to also multiply the vector of sensitivities of its *output* by the *transpose* of its Jacobian matrix, yielding the vector of sensitivities of its inputs. Transforming the entire graph in this fashion allows the efficient computation of a Jacobian-transpose-vector product with only a small constant factor increase in operation count. This allows the gradient to be calculated with only a small constant factor overhead!

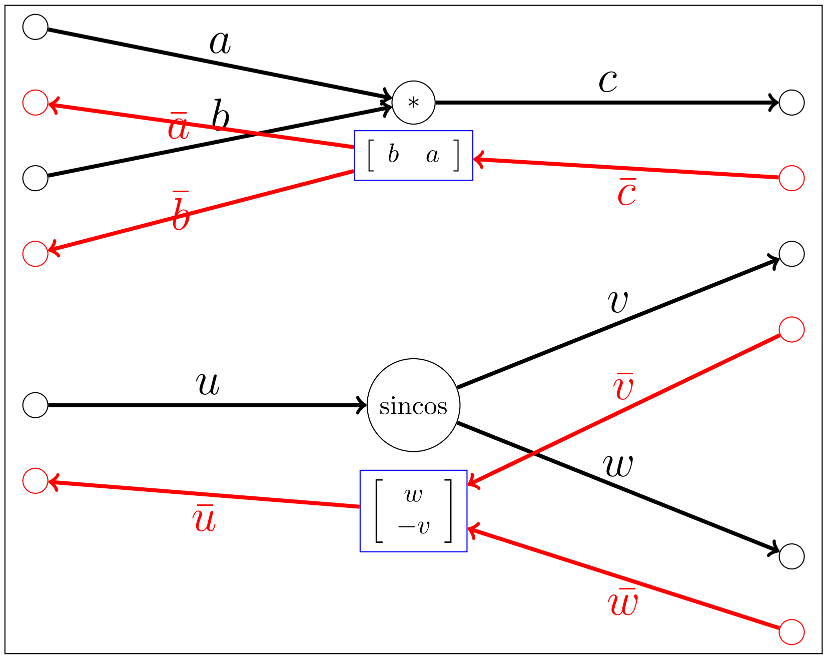

FIGURE 2. Reverse AD augments all primitive arithmetic operations in the original flow graph to also propagate derivatives backwards through the computation.

We can view the computation of Figure 2 as code instead of a graph. Here, the primal must save some state for use in the reverse computation. We do this explicitly, in a data structure.

```
  function Rfig(double a, double b, double u) {
1:    double c = a * b;
2:    double[] (v,w) = sincos(u);
3:    double[] RfigState = (a, b, u, c, v, w);
4:    return (c,v,w, RfigState);
  }
```

The returned variable `RfigState` has to contain all that will be needed to compute the derivative of the function, because during the back propagations of derivatives none of its outputs will be known. The function `Rfig` has to return, as usual, its three-dimensional result: (c, v, w). That is the purpose of the reverse phase: it just needs this "state variable" `RfigState` as first argument, and back propagated derivatives (i.e. derivatives of the outputs of the function, using the notation $\bar{c}, \bar{v}, \bar{w}$) as complementary arguments:

```
  function R2fig(double[] (a,b,u,c,v,w), double cbar, double vbar, double wbar) {
1:    double ubar = w * vbar - v * wbar;
2:    double bbar = a * cbar;
3:    double abar = b * cbar;
4:    return (abar, bbar, ubar);
  }
```

To understand these expressions, focus on $c = a \times b$: the first line of function `Rfig`. It is clear that if a function ϕ is applied to c, we have (with the notation $\bar{c} = \phi'(c)$, corresponding to a back propagation of the derivative coming from an application of functions to c):

$$\bar{a} := \frac{\partial \phi(a \times b)}{\partial a} = b \times \phi'(c) = b \times \bar{c},$$

which is exactly the expression written in the third line of function R2fig.

This could be used to calculate the $m = 3$ rows of the Jacobian by calling R2fig three times, each time calculating one row.

```
(RfigState, c, v, w) = Rfig(a,b,u);
(j11, j21, j31) = R2fig(RfigState, 1, 0, 0);
(j12, j22, j32) = R2fig(RfigState, 0, 1, 0);
(j13, j23, j33) = R2fig(RfigState, 0, 0, 1);
```

Although low overhead for computing a gradient, in terms of operation count[2], makes this technique extremely useful, there are a variety of technical difficulties associated with Reverse AD. Since the derivatives are propagated backwards relative to the original computation, the original values might need to be preserved, increasing the storage burden. Fanout in the original computation corresponds to a linear primitive operation $\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} [u]$, implying that the adjoint computation must multiply by the transpose of this tiny matrix, $\bar{u} = [1\ 1] \begin{bmatrix} \bar{v}_1 \\ \bar{v}_2 \end{bmatrix} = \bar{v}_1 + \bar{v}_2$, corresponding to a simple addition. In other words, fanout in the original computation graph results in addition in the reverse-direction adjoint computation graph. These complexities of the transformation makes efficient implementation, via a source-to-source transformation of the code, quite involved. The most popular implementation strategy is to actually store the dataflow graph and later retrace it in reverse order, which imposes considerable constant-factor overhead.

Although this transformation was formulated in the AD community, and automated, by [39] and others [19], it was known, albeit not automated, in the field of control theory where it was known as the Pontryagin Maximum Principle [32], in Mathematical Physics [14] where it was known as an adjoint calculation, in Machine Learning where it is known as backpropagation [36], and in differential geometry where it is known as the pull-back. The history of the discovery of both Forward AD and Reverse AD is fascinating, with many brilliant and colorful players. This former is discussed by [24], and the latter, at some length, by [19].

2.4. Practical AD Systems and Applications

Most useful complicated numeric routines are not so simple as a static dataflow graph, and part of the complexity of AD is handling control. Other "modes" of AD are also available, to compute higher-order derivatives [25, 30], to vectorize multiple similar derivative calculations with stacked tangents or cotangents sometimes called vector mode, to conserve storage while still calculating gradients efficiently [18, 37], and many many others. Another substantial body of work has arisen as a consequence of the fact that taking exact derivatives and numeric approximation for implementation do not commute—a particularly difficult issue with loops that iterate to convergence. And yet more work has gone into aggressive implementation technologies, with enormous efforts put into aggressive systems like TAPENADE [21, 28, 29], ADIFOR [7], and others. Currently, AD is enjoying substantial applications not just in finance [6] and scientific computation like climate science [22], but also in machine learning [3, 13, 15]. The field of AD is active, and welcoming to newcomers. A catalog of AD implementations and literature is maintained at a community web portal, http://www.autodiff.org/.

2.5. An AD Dream

In considering AD, it may be helpful to distinguish a hierarchy of notions, spanning a spectrum of automation and generality.

AD Type I: A calculus for efficiently calculating derivatives of functions specified by a set of equations.

AD Type II: Rules applied manually which mechanically transform a computer program implementing a numeric function to also efficiently calculate some derivatives.

[2]For our example the gain is not huge, since the forward computation needs $3 + 1 + 2 = 6$ (see lines 2, 4 and 5 of Ffig) versus $3 + 2$ for the backward version (i.e. all computations in function R2fig), at the price of the storage in RfigState of all six variables involved in the primal expressions). But for more complex expressions the gain is more clearly at the advantage of the backward implementation, at a cost of more storage.

AD Type III: A computer program which automatically transforms an input computer program specifying a numeric function into one that also efficiently calculates derivatives.

AD Type IV: AD-taking operators included in language in which numeric algorithms are written, and they can be applied to procedure which use them.

The present author's AD research program, done in collaboration with Jeffrey Mark Siskind, has been to bring Type IV into existence, and to make it fast, robust, general and convenient [31]. These efforts have therefore focused not just on formalizations of the AD process suitable for mathematical analysis and integration into compilers, but also in making AD more general and robust, allowing the AD operators to be used as first-class operators in a nested fashion, thus expanding the scope of programs involving derivatives that can be written succinctly and conveniently [4,5,33,34]. We have also explored the limits of speed that Type IV AD integrated into an aggressive compiler can attain [38].

It is our hope that AD will become as ubiquitous for all numeric programmers as libraries of linear algebraic routines are now, and that optimizing compilers will in the future support AD with the same nonchalance with which they currently support trigonometric functions or loops.

Acknowledgments. This work was funded by Science Foundation Ireland grant 09/IN.1/I2637. Thanks to collaborators Jeffrey Mark Siskind and Atilim Gunes Baydin for many helpful discussions.

3. Practical implementation of adjoint algorithmic differentiation (AAD)

by *Sébastien Geeraert*

3.1. Aim of the study

In this study, we show a way of implementing adjoint algorithmic differentiation (AAD). AAD is a method producing the exact derivatives of a computation, in an automatic and efficient way. By using pathwise differentiation with AAD, we can compute Monte Carlo sensitivities of a price: if the value of the payoff of a financial contract is given by $p(\theta) = \mathbb{E}\left[f(\theta, W)\right]$ where θ is a parameter and W is a random variable we can estimate its derivative by

$$p'(\theta) = \mathbb{E}\left[\frac{\partial f}{\partial \theta}(\theta, W)\right] \approx \frac{1}{M}\sum_{i=1}^{M} \frac{\partial f}{\partial \theta}(\theta, W_i)$$

by swapping the differentiation and the expectation. The advantage of AAD is that the computation time does not depend on the number of sensitivities: by differentiating the operations in reverse (starting with the final computations and going back to the initial computations), we can progressively compute the influence of each intermediate result on the final output. Thus, we can compute every sensitivity at once. In the existing automatic differentiation libraries, there are two methods used: operator overloading (e.g. `ADOL-C`, `Adept`) and source code transformation (e.g. `Tapenade`).

Source code transformation consists in automatically adding in the source code, before compilation, the instructions to compute the derivatives. It requires a tool able to read an existing source code and to understand it (like a compiler does). Therefore, this method is quite hard to implement. It is often unable to handle advanced features of a language (such as object-oriented programming).

Using operator overloading is much easier. Operator overloading is a feature of some programming languages, allowing to redefine basic operators (such as `+`, `*`, `/`). For example, if there is an addition `a+b` in the source code, it will call a customized implementation of `+` (written by the programmer) on the arguments `a` and `b`, instead of performing a traditional addition. In the case of AAD, we redefine the basic operators to also handle the computation of derivatives. It allows to quickly apply AAD to an existing code, with very few modifications to the code. In comparison with source code transformation, operator overloading cannot be as well optimized by the compiler, and may add some overhead during execution. Thus, it is considered as less efficient.

For the sake of simplicity, we choose to use operator overloading in `C++`. The specificities of our implementation are that it focuses on Monte Carlo computations, and that it is efficiently parallelized, whether on CPU (using `OpenMP`) or on GPU (using `CUDA`).

3.2. Principle of AAD and basic implementation

We model a computation as a sequence of n intermediate results x_i produced by elementary functions f_i: $x_i = f_i(x_1, \dots, x_{i-1})$ for $i = 1, \dots, n$. To compute the derivative of the output x_n with respect to the input x_1, there are two possible modes of differentiation:

- The *tangent mode*: we define the tangent of a variable x as $\dot{x} = \frac{\partial x}{\partial x_1}$. We know that $\dot{x}_1 = \frac{\partial x_1}{\partial x_1} = 1$, so we can compute the tangents from $i = 1$ to $i = n$ using the chain rule:
$$\dot{x}_i = \frac{\partial x_i}{\partial x_1} = \frac{\partial}{\partial x_1} f_i(x_1, \dots, x_{i-1}) = \sum_{j=1}^{i-1} \frac{\partial f_i}{\partial x_j} \frac{\partial x_j}{\partial x_1} = \sum_{j=1}^{i-1} \frac{\partial f_i}{\partial x_j} \dot{x}_j.$$
Finally, the sensitivity is given by the tangent $\dot{x}_n = \frac{\partial x_n}{\partial x_1}$.
- The *adjoint mode*: we define the adjoint of a variable x as $\overline{x} = \frac{\partial x_n}{\partial x}$. We know that $\overline{x_n} = \frac{\partial x_n}{\partial x_n} = 1$, so we can compute the adjoints from $i = n$ to $i = 1$ using the chain rule:
$$\overline{x_i} = \frac{\partial x_n}{\partial x_i} = \sum_{j=i+1}^{n} \frac{\partial x_n}{\partial x_j} \frac{\partial f_j}{\partial x_i} = \sum_{j=i+1}^{n} \overline{x_j} \frac{\partial f_j}{\partial x_i}.$$
Finally, the sensitivity is given by the adjoint $\overline{x_1} = \frac{\partial x_n}{\partial x_1}$.

AAD uses adjoint mode: that way, if there are several inputs, all the sentitivities are computed at once (they are given by the corresponding adjoints). That is why it is particularly interesting to use AAD in situations where we want to compute sensitivities to many inputs. The downside of AAD is that, since the derivatives are computed in reverse, the values of the x_i must be available before we can start the differentiation.

Therefore, to apply AAD in practice, we must first do a forward sweep, where we compute and store every intermediate result x_i from the beginning to the end. We can then do the reverse sweep, where we compute the adjoints $\overline{x_i}$ from the end to the beginning. Here is the resulting algorithm:

```
for $i$ from 1 to $n$
    $x_i = f_i(x_1, \dots, x_{i-1})$
$\overline{x_i} = 0$ for $i < n$, $\overline{x_n} = 1$
for $i$ from $n$ to 1
    for $j$ from 1 to $i-1$
        $\overline{x_j} \mathrel{+}= \overline{x_i} \cdot \frac{\partial f_i}{\partial x_j}$
```

This is very easy to implement using operator overloading: the functions f_i are elementary operators which are overloaded to compute and store the relevant quantities during the forward and reverse sweep. To apply AAD on a Monte Carlo computation, we must differentiate the same computation a large number of times with different inputs (the random variables). Therefore, we apply the algorithm above to vectors: each component of the vectors corresponds to one path of the Monte Carlo. At the end of the computation, we can average the components of any adjoint to get the corresponding sensitivity.

We propose two versions of the algorithm that are very close to each other:

- A *CPU version*, where the overloaded operators use `for` loops. To parallelize this version, we use the OpenMP framework, which allows to easily divide the Monte Carlo paths of the `for` loops among the available CPU cores.

- A *GPU version*, where the overloaded operators call GPU functions. These functions, also known as kernels, are implemented in CUDA, a proprietary framework for Nvidia graphic cards.

We need to be careful to avoid useless computations. Some computations are identical across all the Monte Carlo paths: if a computation does not depend on random variables (directly or indirectly), it always has the same inputs and outputs. Therefore, to be efficient, we need to do these computations in a scalar way.

3.3. **Optimisations**

Problems of efficiency arise when a big part of the original computation is deterministic. In that case, the forward sweep is cheap, because almost all the computations are scalar. However, the reverse sweep is very expensive, because all the adjoints computations are vectorial (all the intermediate adjoints indirectly depend on the random variables). For example, this can happen for a call on a log-normal underlying, as shown in figure 3 (left). If the deterministic drift and volatility involve a lot of computations, there will be a lot of corresponding adjoint computations (which are expensive because they are vectorial).

We can solve this problem by computing intermediate means in the middle of the reverse sweep. If we divide the computation as $f(\theta, W) = g(h(\theta), W)$ where $h = (h_1, \dots, h_q)$ is the deterministic part and g the non-deterministic part, the sensitivity is given by

$$\frac{1}{M}\sum_{i=1}^{M}\frac{\partial f}{\partial \theta}(\theta, W_i) = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{q}\frac{\partial g}{\partial h_j}(h(\theta), W_i)\frac{\partial h_j}{\partial \theta}(\theta) = \sum_{j=1}^{q}\left(\frac{1}{M}\sum_{i=1}^{M}\frac{\partial g}{\partial h_j}(h(\theta), W_i)\right)\frac{\partial h_j}{\partial \theta}(\theta).$$

In practice, it means that during the reverse sweep, we can pause the computation between the deterministic and non-deterministic parts, and replace every (vectorial) adjoint by its (scalar) mean. Thus, when we resume the reverse sweep, the subsequent computations are scalar. The effects of the optimisation for the call are shown on figure 3 (right).

Note however that this trick works only because taking the mean of the derivatives is a linear operation. It cannot be applied to a variance computation. But if we want to estimate the variance in addition to the sensitivities, we can divide the Monte Carlo paths into a small number of groups and apply the trick on each group to compute an estimate of the sensitivity; the variance can then be estimated using the different samples of sensitivity obtained on the different groups.

To increase the parallelization, we do independent computations in parallel. Two computations are independent if the inputs of one do not depend on the output of the other. Thus, the computations are divided into layers, which are chronologically ordered: each layer can only begin when the previous layer is finished. Inside a layer, everything can be computed in parallel.

When an intermediate result is used as input by many computations, the corresponding adjoint has to be updated many times during the reverse sweep. But since adjoint updates read and modify the adjoints, two adjoint updates on the same variable cannot happen in parallel (they have to be in different layers): if the adjoint updates are done naively (i.e. by adding the updates successively one by one), the number of layers will grow linearly with respect to the number of adjoint updates, which breaks the parallelization. But the resulting adjoint is just a sum of many independent contributions. We can therefore use a divide and conquer method: the big sum can be computed as the addition of two smaller independent sub-sums. Thus, by aggregating the adjoint updates by powers of two, the number of layers grows only logarithmically. This process of aggregation is shown in figure 4.

Finally, we can also take advantage of the specificities of the model and the payoff. For example, for a CVA computation, the payoff can be seen as a sum of much simpler payoffs. Therefore, we can replace one big AAD with many small ones, which improves parallelization. Furthermore, we can in the GPU version make the work on CPU (the analysis of dependencies to divide the computational graph into layers) overlap with the work on GPU (the actual computation of intermediate results and adjoints): while the GPU computes the results on one AAD, the CPU works on the next AAD.

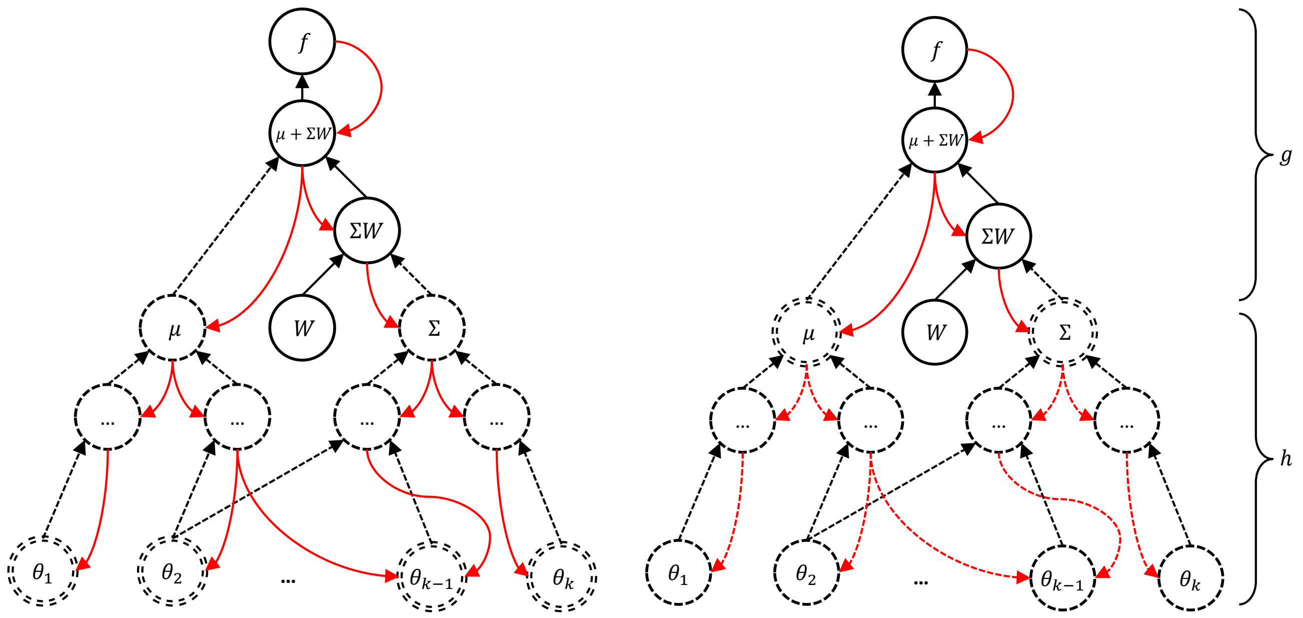

Figure 3. Computational graph for a call on a log-normal underlying, with deterministic drift $\mu(\theta)$ and deterministic volatility $\Sigma(\theta)$. The payoff has the form $f(\theta, W) = (\exp(\mu(\theta) + \Sigma(\theta)W) - K)_+$ where θ represents the inputs and W is a standard gaussian. Each node corresponds to an intermediate result in the forward sweep. Each downward arrow (in red) corresponds to an adjoint update. Solid lines represent vectorial computations and dashed lines represent scalar computations. Means of the adjoints are computed at the doubly circled nodes. On the left (without optimisation), means are computed at the end of the reverse sweep, therefore the reverse sweep is entirely vectorial. On the right (with optimisation), means are computed in the middle of the reverse sweep, therefore the reverse sweep is mostly scalar (which is much cheaper). Here the non-deterministic part is $g(h, W) = (\exp(h_1 + h_2 W) - K)_+$ and the deterministic part is $h(\theta) = (\mu(\theta), \Sigma(\theta))$.

3.4. **Results**

On a CVA example where we compute 200 sensitivities with 5000 paths, we get the following timings:

	CPU	GPU
Finite differences	271 s	29 s
AAD	9.1 s	2.4 s

We see that when there are many sensitivities to compute, AAD allows interesting gains in comparison with finite differences: a factor of 30 on CPU and of 12 on GPU. It shows that the method can be efficiently parallelized, both on CPU and on GPU, especially if we take advantage of the specificities of the model and payoff. However, AAD cannot be directly applied to non-smooth payoffs. That is why [16] proposed the Vibrato Monte Carlo method, which consists in applying the Likelihood Ratio Method to the conditional expectation at the last step of an Euler scheme.

4. Second Sensitivities in Quantitative Finance

By *Olivier Pironneau*, based on a joint work with Gilles Pagès and Guillaume Sall

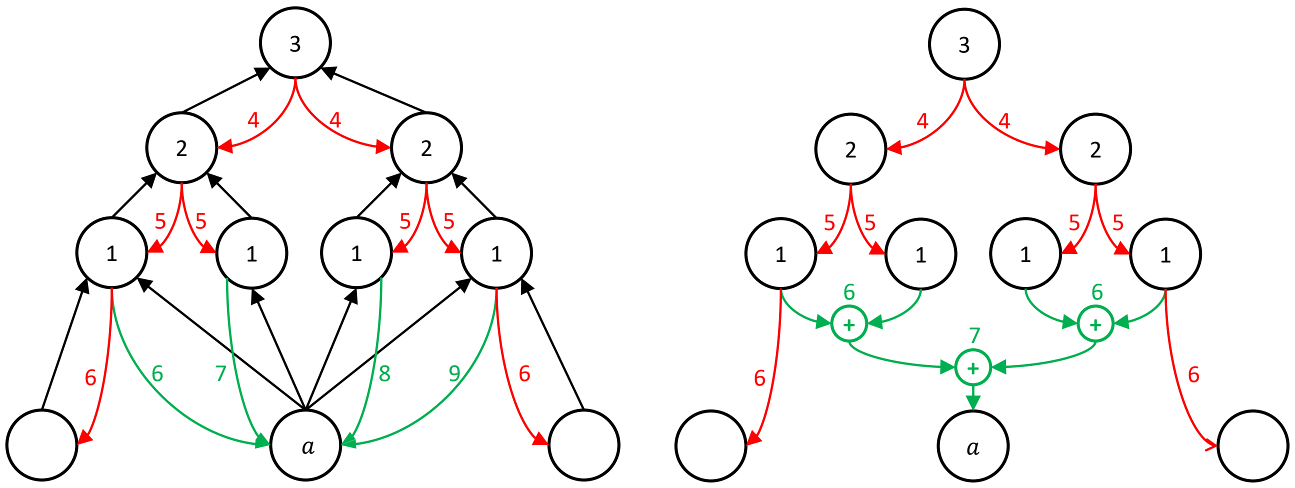

FIGURE 4. Example where an intermediate result (called a) is used by many computations. For each node or adjoint update, we have indicated the layer to which the computation belongs. Layers 1 to 3 correspond to the forward sweep. Layers 4 and beyond correspond to the reverse sweep. Without aggregation (on the left), adjoint updates for a (in green) occupy four layers (6 to 9). With aggregation (on the right, where forward arrows have been omitted for the sake of clarity), adjoint updates for a are done with only two layers (6 and 7).

This short summary presents our work aimed at the computation of higher order sensitivities in quantitative finance; in particular Vibrato and automatic differentiation are compared with other methods. We show that this combined technique is more stable than automatic differentiation of second order derivatives by finite differences and more general than Malliavin Calculus. We also extend automatic differentiation for second order derivatives of options with non-twice differentiable payoff.

Sensitivities of large portfolios are time consuming to compute. Vibrato [16], [17] is an efficient method for the differentiation of mean value of functions involving the solution of a stochastic ODE. In many cases, double sensitivities, i.e. second derivatives with respect to parameters, are needed (e.g. gamma hedging). Then one can either apply an automatic differentiation module to Vibrato or try Vibrato of Vibrato, or even do a second order automatic differentiation of the computer program. But in finance the payoff is never twice differentiable and so generalized derivatives have to be used requiring approximations of Dirac functions of which the precision is also doubtful.

Consider a financial asset X_t modeled by

$$\mathrm{d}X_t = X_t\Big(r(t)\mathrm{d}t + \sigma(X_t,t)\mathrm{d}W_t\Big), \quad X_0 \text{ given, } W_t \text{ Brownian}$$

A European put option $V_t = e^{-r(T-t)}\mathbb{E}(K - X_T)^+$. How can we compute the 1st and 2nd derivatives of V_0 w.r. to K,T,X_0, r,σ. The most straightforward approach is to use the following approximation

$$\Gamma = \frac{\partial^2 V_0}{\partial X_0^2} \approx \frac{1}{h^2}\Big(V_0|_{X_0+h} - 2V_0|_{X_0} + V_0|_{X_0-h}\Big)$$

It costs 3 times the computation of the option but it is unstable with respect to h: if h is too small round-off errors pollute the result because both the numerator and the denominator are small while the result of the division is not.

One trick is to use a complex h. With $\mathbf{i} := \sqrt{-1}$,

$$Re[\frac{f(a+\mathbf{i}\delta a)-f(a)}{\mathbf{i}\delta a}] = Im\frac{f(a+\mathbf{i}\delta a)}{\delta a} = f'(a) - f^{(3)}\frac{\delta a^2}{6} + o(\delta a^3)$$

and it is seen that the expression in the middle is not cursed by a 0 over 0 indeterminacy.

Does it work on non-differentiable functions? For example take $f(x) = (1-x)^+$ leading to $f'(x) = -\mathbf{1}_{x<1}$ and $f"(x) = \delta(1-x)$. Using Maple, Figure 5 shows clearly that complex finite difference to compute the second derivative is superior because it gives a correct result even with a very small complex increment. The conclusion

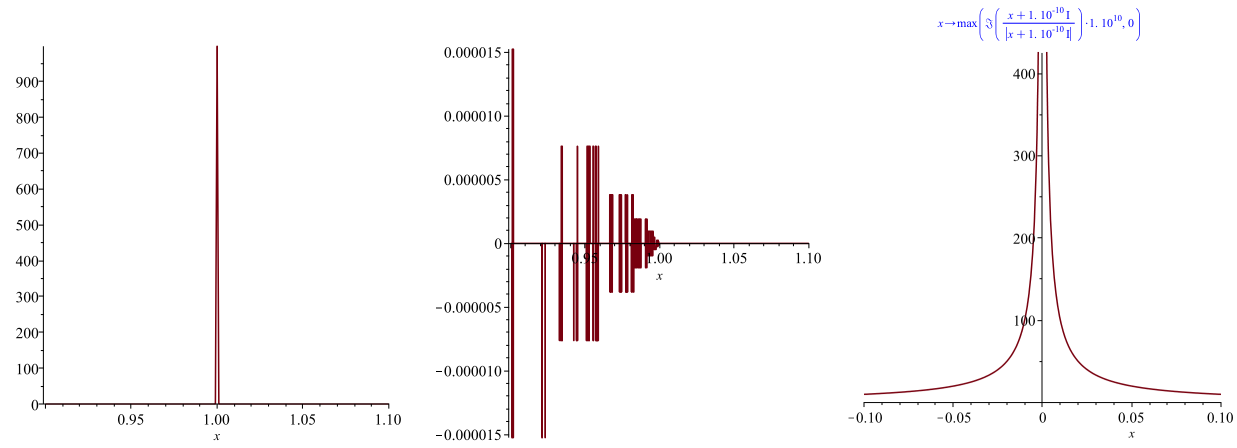

FIGURE 5. using Maple to plot the 2^{nd} derivative of $(1-x)^+$ with $\delta x = 10^{-3}$ (left), $\delta x = 3.510^{-6}$ (middle) and $\delta x = \mathbf{i}10^{-10}$ (right).

is that unless something is done the results will be unreliable.

Alternatively Malliavin calculus can be used in some but not all cases. For instance, if the payoff V of a vanilla future with spot price $X_t(x)$ where X_t satisfies the Black-Scholes SDE with constant σ and r, with $X_0 = x$, then $\Gamma = \mathbb{E}[\frac{e^{-rT}V(X_T)}{x^2T\sigma}(\frac{W_T^2}{\sigma T} - W_T - \frac{1}{\sigma})]$, It works so long as the weight functions are known and T not too small, but when applicable it is clearly faster than anything else.

4.1. **Vibrato**

Let us now turn to Vibrato, as introduced by Giles in [16]. Let θ be a parameter and let us focus on $\partial_\theta\mathbb{E}[V(X_T)]$ with $dX_t = b(\theta, X_t)\,dt + \sigma(\theta, X_t)dW_t, \quad X_0 = x$. With Z a normal vector valued random variable, an Euler explicit scheme leads to

$$\bar{X}_k = \bar{X}_{k-1} + b(\theta, \bar{X}_{k-1})h + \sigma(\theta, \bar{X}_{k-1})\sqrt{h}Z_k, \quad \bar{X}_0 = x, \quad k = 1, \ldots, n.$$

Now write $\mathbb{E}\left[V(\bar{X}_n)\right] = \mathbb{E}\left[\mathbb{E}\left[V(\bar{X}_n) \mid \bar{X}_{n-1}\right]\right]$ and note that, with $\sigma_{n-1}(\theta) = \sigma(\theta, \bar{X}_{n-1}(\theta))$,

$$\bar{X}_n = \mu_{n-1}(\theta) + \sigma_{n-1}(\theta)Z_n\sqrt{h} \text{ with } \mu_{n-1}(\theta) = \bar{X}_{n-1}(\theta) + b(\theta, \bar{X}_{n-1}(\theta))h,$$
$$\frac{\partial}{\partial\theta_i}\mathbb{E}[V(\bar{X}_n(\theta))] = \mathbb{E}_z\left[\frac{\partial}{\partial\theta_i}\left\{\mathbb{E}[V(\mu+\sigma Z\sqrt{h})]\right\}_{\substack{\mu\,=\,\mu_{n-1}(\theta)\\ \sigma\,=\,\sigma_{n-1}(\theta)}}\right] \tag{1}$$

The last time step sees constant coefficients, so we have an explicit solution. Some algebra (see [27]) around the identity

$$\frac{\partial}{\partial \theta_i}\Big[\mathbb{E}[V(X(\theta))]\Big]_{|\theta=\theta^0} = \int_{\mathbb{R}^d} V(y)\frac{\partial \log p}{\partial \theta_i}(\theta^0, y)p(\theta^0, y)dy = \mathbb{E}\left[V(X)\frac{\partial \log p}{\partial \theta_i}\right]_{|\theta^0}, \tag{2}$$

where p is the probability density of X, leads to the following result.

Proposition 1.

$$\frac{\partial}{\partial \theta}\mathbb{E}[V(\bar{X}_n(\theta))] = \mathbb{E}\left[\frac{1}{\sqrt{h}}\frac{\partial \mu}{\partial \theta}\cdot \mathbb{E}_z\left[V(\mu+\sigma Z\sqrt{h})\sigma^{-T}Z\right] + \frac{1}{2}\frac{\partial(\sigma\sigma^T)}{\partial \theta} : \mathbb{E}_z\left[V(\mu+\sigma Z\sqrt{h})\sigma^{-T}(ZZ^T-I)\sigma^{-1}\right]\right],$$

where $f\cdot v$ denotes the scalar product between f and v and $M : P$ denotes the trace of the product of matrices M and P.

Note that in the non constant case the tangent process $Y_t = \partial_\theta X_t$ is involved. Note also that V is not differentiated! The computing time in the non constant case is twice the evaluation of $V(X_0)$, similar to finite difference but much more precise. Note finally that antithetic variance reduction is easy to apply. More details can be found in [27].

4.2. Higher Derivatives with Automatic Differentiation

Once the computer program for the vibrato is written, automatic differentiation can be applied. It is exact because every line of the program is differentiated exactly (teaching the compiler for instance by adding in the AD library that the derivative of $\sin x$ is $\cos x$).

Automatic differentiation can be extended to handle the difficulty raised above for non-differentiable pay-off, to some degree, by approximating the Dirac mass at 0 by $\delta^a(x) = \frac{1}{a\sqrt{\pi}}e^{-\frac{x^2}{a^2}}$. Now, suppose f is discontinuous at $x = z$ and smooth elsewhere; then $f(x) = f^+(x)H(x-z) + f^-(x)(1-H(x-z))$; hence

$$f'_z(x) = (f^+)'_z(x)H(x-z) + (f^-)'_z(x)(1-H(x-z)) - (f^+(z) - f^-(z))\delta(x-z)$$

Unless this last term is added, the computation of the second order sensitivities will be wrong.

If in the Automatic library the ramp function x^+ is defined as $xH(x)$ with its derivative to be $H(x)$, if H is defined with its derivative equal to δ^a and if in the program which computes the financial asset these new functions are used explicitly, like $(x-K)^+ = \text{ramp}(x-K)$ and not $\max(x-K, 0)$, then the second derivative in K computed by the AD library will be $\delta^a(x-K)$. Moreover, it will also compute

$$\int_0^\infty f(x)(x-K)^+dx \approx \frac{1}{N}\sum_{i=1}^N f(\xi_i)\delta^a(\xi_i - K)$$

where ξ_i are the N quadrature points of the integral or the Monte$-$Carlo points used by the programmer to approximate the integral.

4.3. Conceptual algorithm for VAD

In Algorithm 1, we summarize here what needs to be done to compute a second order sensitivity by VAD with antithetic variance reduction in the general case. To generate M simulation paths with time step $h = \frac{T}{n}$ of the underlying asset X and its tangent process $Y = \dfrac{\partial X}{\partial \theta}$ we need Mn realisations of a normal random variable Z; we need also M_Z realisations of Z_n^m, called $\mathbb{Z}_i$, for the last time step for each m. Then given $\bar{X}_0, \bar{Y}_0$:

Algorithm 1 VAD for second derivatives using antithetic variance reduction

```
1: for $m = 1, .., M$ do
2:     for $k = 0, \dots, n-2$ do
3:         $\bar X^m_{k+1} = \bar X^m_k + rh\bar X^m_k + \bar X_k \sigma\sqrt{h} Z^m_{k+1}$
4:         $\bar Y^m_{k+1} = \bar Y^m_k + rh\bar Y^m_k + r'_\theta h \bar X^m_k + \left(\bar Y^m_k \sigma + \sigma'_\theta \bar X^m_k\right)\sqrt{h} Z^m_{k+1}$
5:     end for
6:     for $i = 1, .., M_Z$ do
7:         $X^m_{T_\pm} = \bar X^m_{n-1} + rh\bar X^m_{n-1} \pm \sigma^m \bar X_{n-1}\sqrt{h}\mathbb{Z}_i,\ \bar X_{T_o} = \bar X_{n-1} + rh\bar X_{n-1}$
8:         $V^m_{T_\pm,} = (\bar X^m_{T_\pm} - K)^+,\ V^m_{T_o} = (\bar X^m_{T_o} - K)^+.$
9:         $R = \bar Y^m_{n-1}(1+rh) + \bar X^m_{n-1} r'_\theta h,\quad S = \bar Y^m_{n-1}\sigma + \bar X^m_{n-1}\sigma'_\theta$
10:        $\left(\frac{\partial V_T}{\partial \theta}\right)^m_i = R\frac{(V^m_{T_+} - V^m_{T_-})\mathbb{Z}_i}{2\bar X^m_{n-1}\sigma\sqrt{h}} + S(V^m_{T_+} - 2V^m_{T_o} + V^m_{T_-})\frac{{\mathbb{Z}_i}^2 - 1}{2\bar X^m_{n-1}\sigma}$
11:    end for
12:    Compute $\left(\frac{\partial^2 V_T}{\partial \theta^2}\right)^m_i$ by AD on the program that implements $\left(\frac{\partial V_T}{\partial \theta}\right)^m_i$
13:    $\left(\frac{\partial^2 V_T}{\partial \theta^2}\right)^m = \frac{1}{M_Z}\sum_{i=1}^{M_Z}\left(\frac{\partial^2 V_T}{\partial \theta^2}\right)^m_i$
14: end for
15: $\frac{\partial^2 V_T}{\partial \theta^2} = \frac{1}{M}\sum_{m=1}^{M}\left(\frac{\partial^2 V_T}{\partial \theta^2}\right)^m.$
```

American Option

The value V_t of an American option requires the best exercise strategy. Let φ be the payoff; the dynamic programming approximation of the option is

$$\bar V_{t_n} = e^{-rT}\varphi(\bar X_T),\quad C_{t_k} = \mathbb{E}[e^{-rh}\bar V_{t_{k+1}} \mid \bar X_{t_k}],\quad \bar V_{t_k} = \max\left\{e^{-rt_k}\varphi(\bar X_{t_k}), C_{t_k}\right\}. \tag{3}$$

Following Longstaff et al. [26] the continuation value is approximated by a least square projection on I real valued basis functions, t_k, $\{\psi_i\}_{i=1}^I$:

$$C_k \simeq \sum_{i=1}^I \alpha_{k,i}\psi_i(\bar X_{t_k}),\ \alpha_{k,\cdot} = \text{argmin}_\alpha \mathbb{E}\left[\left(e^{-rh}\bar V_{t_{k+1}} - \sum_{i=1}^I \alpha_{k,i}\psi_i(\bar X_k)\right)^2\right]. \tag{4}$$

Once the optimal stopping time τ^* is known, the differentiation with respect to θ can be done as for a European contract. The dependency of the τ^* on θ is neglected; arguably this dependency is second order but this point needs to be validated.

We consider the following parameters : $\sigma = 20\%$ or $\sigma = 40\%$, X_0 varying from 36 to 44, $T = 1$ or $T = 2$, $K = 40$ and $r = 6\%$. The Monte Carlo parameters are: $M = 5\ 10^4$ simulation paths and $T/h = 50$ time steps. The basis in the Longstaff-Schwartz algorithm is $\psi_i(x) = x^{i-1}, i = 1, 2, 3$, i.e. $I = 3$.

We compare with the solution of the Black-Scholes partial differential inequation discretized by an implicit Euler scheme in time, finite element in space and semi-smooth Newton for the inequalities [1]. A large number of grid points are used, 10^4 to make it a reference solution.

A second order finite Difference approximation is also used to compute the Gamma for comparison.

In Table 1, the results are shown for different sets of parameters taken from Longstaff et al. [26]. The method provides a good precision when variance reduction is used, except when the underlying asset price is small with a small volatility. As for the computation time, the method is faster than Finite Difference.

Conclusion. Faced with the problem of writing a general software for the computations of sensitivites in connection with Basel III directives, we found that Automatic Differentiation alone is not up to the task, but applied to Vibrato it is dependable and fairly general.

Table 1. Results of the price, the Delta and the Gamma of an American option. The reference values are obtained via the Semi-Newton method plus Finite Difference, they are compared to Vibrato plus Automatic Differentiation on the Longstaff-Schwartz algorithm. We compute the standard error for each American Monte Carlo results. The settings of the American Monte Carlo are 50 time steps and $50,000$ simulation paths.

S	σ	T	Price (AMC)	Standard Error	Delta Vibrato (AMC)	Standard Error	Gamma Ref. Value	Gamma VAD (AMC)	Standard Error
36	0.2	1	4.46289	0.013	0.68123	1.820e−3	0.08732	0.06745	6.947e−5
36	0.2	2	4.81523	0.016	0.59934	1.813e−3	0.07381	0.06398	6.846e−5
36	0.4	1	7.07985	0.016	0.51187	1.674e−3	0.03305	0.03546	4.852e−5
36	0.4	2	8.45612	0.024	0.44102	1.488e−3	0.02510	0.02591	5.023e−5
38	0.2	1	3.23324	0.013	0.53063	1.821e−3	0.07349	0.07219	1.198e−4
38	0.2	2	3.72705	0.015	0.46732	1.669e−3	0.05907	0.05789	1.111e−4
38	0.4	1	6.11209	0.016	0.45079	1.453e−3	0.02989	0.03081	5.465e−5
38	0.4	2	7.61031	0.025	0.39503	1.922e−3	0.02233	0.02342	4.827e−5
40	0.2	1	2.30565	0.012	0.40780	1.880e−3	0.06014	0.05954	1.213e−4
40	0.2	2	2.86072	0.014	0.39266	1.747e−3	0.04717	0.04567	5.175e−4
40	0.4	1	5.28741	0.015	0.39485	1.629e−3	0.02689	0.02798	1.249e−5
40	0.4	2	6.85873	0.026	0.35446	1.416e−3	0.01987	0.02050	3.989e−5
42	0.2	1	1.60788	0.011	0.29712	1.734e−3	0.04764	0.04563	4.797e−5
42	0.2	2	2.19079	0.014	0.28175	1.601e−3	0.03749	0.03601	5.560e−5
42	0.4	1	4.57191	0.015	0.34385	1.517e−3	0.02391	0.02426	3.194e−5
42	0.4	2	6.18424	0.023	0.29943	1.347e−3	0.01768	0.01748	2.961e−5
44	0.2	1	1.09648	0.009	0.20571	1.503e−3	0.03653	0.03438	1.486e−4
44	0.2	2	1.66903	0.012	0.21972	1.487e−3	0.02960	0.02765	2.363e−4
44	0.4	1	3.90838	0.015	0.29764	1.403e−3	0.02116	0.02086	1.274e−4
44	0.4	2	5.58252	0.028	0.28447	1.325e−3	0.01574	0.01520	2.162e−4

5. CVA with Greeks and AAD

by *Adil Reghai*

5.1. **Framework of application**

In [35], we proposed a Monte Carlo approach for pricing CVA using the duality relationship between parameter and hedging sensitivities as well as Ito integral calculus.

CVA is the market-value of counterparty credit risk. It is computed as the difference between the risk-free value of the portfolio and its true value that takes into account counterparties default. When both parties to a bilateral contract may default, the price adjustment is called BCVA (Bilateral Credit Valuation Adjustment).

Under the Monte Carlo approach, CVA is rewritten as a weighted sum of expected discounted exposures from time of calculation t to the maturity of the portfolio T. A number of paths for portfolio underlying market variables are simulated and exposures distributions are obtained at each time between t and T. Expected exposure is then given by averaging exposures along simulated paths. Although, Monte Carlo simulations are quite easy to implement, for some complex portfolios this tool could easily lead to heavy computational requirements, thus quickly proving unworkable. Recently, [9] introduced a PDE representation for the value of financial derivatives under counterparty credit risk. Using the Feynman-Kac theorem, the total value of the derivative is decomposed into a default free value plus a second term corresponding to the CVA adjustment. However, for dimensions higher than 3, the PDE can no longer be solved with a finite-difference scheme. Building on this work, [23] introduces a new method based on branching diffusions using a Galton-Watson random tree. Although this method drastically reduces computational costs, it is still complex to implement.

Summarizing, we see that both existing approaches for CVA computation suffer from serious drawbacks. Hence we introduced a new approach for CVA valuation in a Monte Carlo setting. Our approach is innovative in that it combines both the martingale representation theorem and Adjoint Algorithmic Differentiation (AAD) techniques to retrieve future prices in a highly efficient manner.

To give a rough understanding of the methodology, that is more "trajectory by trajectory"-driven than the two previously existing ones (Monte-Carlo or PDE), it is needed to set few notations:

- The price of an equity follows here a geometric Brownian motion with a drift made of the risk free rate r_t and a financing cost ϕ_t:
$$\frac{dS_t}{S_t} = (r_t + \phi_t)\, dt + \sigma_t\, dW_t.$$
Assuming the risk free rate is zero for the sake of simplicity, any payoff $V(t, S_t)$ of a derivative with a maturity T written on such an equity hence follows the Back-Scholes backward PDE:
$$\frac{\partial V}{\partial t} + S_t\, \phi(t, S_t)\, \frac{\partial V}{\partial S} + \frac{1}{2}\, \sigma^2(t, S_t)\, S_t^2\, \frac{\partial^2 V}{\partial^2 S} = 0.$$
- In the scope of this focus on a use of AAD in finance, we will consider a case in which, seen from the issuer, only the default of the counterparty can occur, and take as given the expression of the Credit Valuation Adjustment (CVA) in such a context: a call option with zero strike on the net present value of a "portfolio" of one asset at the random time of default of the counterparty. Following [10], define CVA as
$$\text{CVA} = \text{LGD} \cdot \mathbb{E}\left(\mathbf{1}_{\tau \leq T}\, D(0, \tau)\, X(\tau)\right),$$
where: LGD is the "Loss Given Default" (i.e. the fraction of loss incurred by the investor upon default of his counterparty), τ is the default time, $D(u, v)$ is the discount factor between u and v (under our zero risk free rate assumption: $D(u, v) = 1$), and $X(t)$ is the "exposure[3] at time t". $X(t)$ can hence be written as the positive part of a solution V of (5).
- Last but not least, we assume here the hazard rate λ is constant and known, meaning the probability to default reads $\mathbb{Q}(\tau \leq t) = 1 - e^{-\lambda t}$.
- All this allows to write the CVA seen from time $t \leq \tau$ ([35, Section 3] for details) as
$$\text{CVA}_t = \mathbb{E}_t \int_{u=t}^{\tau} \text{LGD} \cdot V(u)^+ \lambda e^{-\lambda(u-t)}\, du. \tag{5}$$
A way to enable the use of AAD here is to consider a pathwise version of $\mathbb{E}_t$ (the expectation seen from t) using N_{paths} paths and to write this approximation of the CVA_t (with the notation v_i for the value along a given trajectory, i.e. the cost of this trajectory):
$$\text{CVA}_t = \frac{1}{N_{\text{paths}}} \sum_{i=1}^{N_{\text{paths}}} \underbrace{\int_{u=t}^{\tau} \text{LGD} \cdot v_i(u) \cdot \mathbf{1}_{v_i(u) \geq 0}\, \lambda\, e^{-\lambda(u-t)}\, du}_{\text{CVA}_t^i}. \tag{6}$$

In [35, Section 4], it is first established the following relationship
$$\left.\frac{\partial V}{\partial \phi(u, S_u^i)}\right|_{(t, S_t)} = \mathbb{E}_t \int_{z=t}^{T} \mathbf{1}_{(z=u, S_z = S_u^i)} S_u^i\, \frac{\partial V}{\partial S}(z, S_z)\, dz.$$

It means the partial derivative of the value function with respect to the financial costs at any time and price (u, S_u^i), can be expressed as the conditional expectation of the sum of the price S_u^i times the delta of the value function V sampled when $z = u$ and $S_z = S_u^i$. Keep in mind any point of the surface $(u, S_u^i) \mapsto \phi(u, S_u^i)$ can be chosen to compute the partial derivative of $V(t)$, and hence the "sampling" at $z = u$ and $S_z = S_u^i$

[3]We furthermore assume the exposure X and the default of the counterparty are independent.

means any trajectory crossing (u, S_u^i) at z in $[t, T]$ is considered. The mass of these events being $f_S(u, S_u^i) := \int_{z=t}^{T} \mathbf{1}_{(z=u, S_z=S_u^i)}\, dz$, we can then write

$$\underbrace{\frac{\partial V}{\partial S}(u, S_u^i)}_{\text{hedging sensitivity}} = \frac{1}{S_u^i\, f_S(u, S_u^i)} \cdot \underbrace{\frac{\partial V}{\partial \phi(u, S_u^i)}}_{\text{input sensitivity}} \quad . \tag{7}$$

Then, AAD techniques come into play in order to compute hedging sensitivities by applying the chain rule of instructions in a reversed order with regards to its original formulation. As shown in [11], the execution time of the AAD code is bounded by approximately four times the cost of execution of the original function.

Our approach to compute CVA could be summarized by the following four steps procedure:

1. **Solve the underlying value function V.**
   Generate a first Monte Carlo sampling consisting of $N_{\text{paths}}^{(1)}$ for the stock process. In this first sampling phase we determine the initial price $V(t)$ (price at time t of CVA valuation) as well as all input sensitivities $\frac{\partial V}{\partial \phi}(t, S)$ using the chain rule of AAD.
2. **Use the knowledge of V to sample the input sensitivities.**
   Generate a second Monte Carlo sample consisting of $N_{\text{paths}}^{(2)}$ for the stock process. At each node (path i, time u) of the stock price:
   - Get corresponding input sensitivities $\frac{\partial V}{\partial \phi}(u, S_u^i)$ by linear interpolation using results in step 1.
   - Calculate corresponding hedging sensitivities using (7).
   - Calculate $v_i(u)$ for each trajectory i at each time u (thanks to the martingale representation theorem allowing to express v_i as a perturbation of V).
3. **Pathwise CVA estimation.**
   Now it is possible to compute CVA_t^i along each path using its expression in formula (6).
4. **Averaging** these CVA_t^i over the $N_{\text{paths}}^{(2)}$ paths leads to an estimate of the CVA defined by (5).

Note the use of AAD unlocks step 1. Without such a systematic and automated method, it would be needed to go back to a finite difference approach, having no gain in using formula (7).

Moreover, it is possible to extend this AAD based approach to several equity payoffs and even to some path dependent ones [35].

5.2. Practical relevance of our research paper

The method we presented has potentially a considerable impact on the way banks calculate and manage their CVA. Indeed, not only AAD methods enhance the computational burden of CVA, but they also allow one to obtain sensitivities with regards to all input parameters (in a row). This turns out very crucial for all banks that need to risk manage their derivatives portfolios and to fulfil regulatory demand (see the Introduction of paper).

Method	Number of paths	Computing time
CVA with MC-MC	MC for prices: 10,000 MC for CVA: 10,000	5 hours
CVA with AAD	Step 1: 1,000,000 Step 2: 10,000	29 seconds

Table 2. Comparison of a standard approach (MC-MC: twice Monte-Carlo), with the proposed AAD one: two steps too, but automatic differentiation.

As far as CVA is concerned, we tested our approach for a number of equity payoffs and compared it to more standard approaches (basically MC of MC to calculate exposures at future times). As a matter of fact, we compared the computing time required to value CVA for a double barrier option (2 barriers: up and down) where no closed form solution for neither price nor CVA exists. For this option, the price is computed using Monte Carlo and CVA using a Monte Carlo of Monte Carlo (simulation of simulation). The main results are summarized in Table 2.

Results have been obtained with R code with a Processor Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz and 16 GB of installed RAM. So, we see that the speed up in computing time is over 620 times. Moreover, we have directly the surface of hedging sensitivities using the AAD method (with no other simulations required), whereas simulations would be needed to value hedging sensitivities in the standard MC-MC framework.

Acknowledgments. This mini-symposium and the sections of this paper have been coordinated and harmonized by Charles-Albert Lehalle.

Disclaimer. Views expressed by authors are the authors' own and do not necessarily reflect the view of the companies they work at.

References

[1] Y. Achdou and O. Pironneau. *Computation methods for option pricing.* Frontiers in Applied Mathematics. SIAM, Philadelphia, 2005. xviii+297 pp., ISBN 0-89871-573-3.

[2] Bank for International Settlements. *Minimum capital requirements for Market Risk*, 2015.

[3] A. G. Baydin and B. A. Pearlmutter. Automatic differentiation of algorithms for machine learning. In *Proceedings of the AutoML Workshop at the International Conference on Machine Learning (ICML), Beijing, China, June 21–26, 2014*, 2014.

[4] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*, 2015a.

[5] Atilim Gunes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. Diffsharp: Automatic differentiation library. Technical Report arXiv:1511.07727, arXiv, 2015b.

[6] C. H. Bischof, H. M. Bücker, and B. Lang. Automatic differentiation for computational finance. In E. J. Kontoghiorghes, B. Rustem, and S. Siokos, editors, *Computational Methods in Decision-Making, Economics and Finance*, volume 74 of *Applied Optimization*, chapter 15, pages 297–310. Kluwer Academic Publishers, Dordrecht, 2002. .

[7] Christian H. Bischof and Alan Carle. Users' experience with ADIFOR 2.0. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 385–392. SIAM, Philadelphia, PA, 1996. ISBN 0–89871–385–4.

[8] Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and Jean Utke, editors. *Advances in Automatic Differentiation*, volume 64 of *Lecture Notes in Computational Science and Engineering*. Springer, Berlin, 2008. ISBN 978-3-540-68935-5. .

[9] C. Burgard and M. Kjaer Partial Differential Equation Representations of Derivatives with Bilateral Counterparty Risk and Funding Costs. *Journal of Credit Risk*, 7:75–93, 2011.

[10] Damiano Brigo and Agostino Capponi Bilateral Counterparty Risk Valuation with Stochastic Dynamical Models and Application to Credit Default Swaps . *SSRN*, Dec. 2008.

[11] L. Capriotti and M. Giles Algorithmic Differentiation: Adjoint Greeks Made Easy. *Risk*; Sept: 92-98, 2012.

[12] William Kingdon Clifford. Preliminary sketch of bi-quaternions. *Proceedings of the London Mathematical Society*, 4:381–95, 1873.

[13] David Duvenaud and Ryan P. Adams. Black-box stochastic variational inference in five lines of Python. NIPS Workshop on Black-box Learning and Inference, 2015.

[14] Richard Phillips Feynman. Forces in molecules. *Physical Review*, 56(4):340–3, August 1939. .

[15] Andrew Gelman, Daniel Lee, and Jiqiang Guo. Stan: A probabilistic programming language for Bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530–543, 2015. .

[16] M. B. Giles. Vibrato Monte Carlo sensitivities. In P. L'Ecuyer and A. Owen, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2008*, pages 369–382. New York, Springer edition, 2009.

[17] M. B. Giles. Monte carlo evalutation of sensitivities in computational finance. September 20–22, 2007.

[18] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[19] Andreas Griewank. Who invented the reverse mode of differentiation? *Documenta Mathematica*, Extra Volume ISMP:389–400, 2012.

[20] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008. ISBN 978–0–898716–59–7.

[21] Laurent Hascoët and Valérie Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, 2004.

[22] Klaus Hasselmann, Susanne Hasselmann, Ralf Giering, Victor Ocaña, and Hans von Storch. Sensitivity Studdy of Optimal CO_2 Emission Paths using a Simplified structural Integrated Assesment Model (SIAM). *Climatic Change*, 37:345–386, 1997.

[23] P. Henry-Labordere Counterparty Risk Valuation: A Marked Branching Diffusion Approach. *SSRN Electronic Journal*, 1556-5068, 2012.

[24] Masao Iri. History of automatic differentiation and rounding error estimation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 1–16. SIAM, Philadelphia, PA, 1991. ISBN 0–89871–284–X.

[25] Jerzy Karczmarczuk. Functional differentiation of computer programs. *Higher-Order and Symbolic Computation*, 14:35–57, March 2001. ISSN 1388-3690.

[26] F. A. Longstaff and E. S. Schwartz. Valuing American options by simulation: a simple least squares approach. *Review of Financial Studies*, 14:113–148, 2001.

[27] G. Pagès, O. Pironneau and G. Sall Vibrato and Automatic Differentiation for High Order Derivatives and Sensitivities of Financial Options Submitted to J. Comp. Finance. 2017.

[28] Valérie Pascual and Laurent Hascoët. Extension of TAPENADE toward Fortran 95. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 171–179. Springer, New York, NY, 2005. .

[29] Valérie Pascual and Laurent Hascoët. TAPENADE for C. In [8], pages 199–209. ISBN 978-3-540-68935-5. .

[30] Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 2007 Symposium on Principles of Programming Languages*, pages 155–60, Nice, France, jan 2007.

[31] Barak A. Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make automatic differentiation sound and efficient. In [8], pages 79–90. ISBN 978-3-540-68935-5. .

[32] Lev Semenovich Pontryagin, V. G. Boltyanskii, R. V. Gamrelidze, and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes*. John Wiley & Sons, Inc., 1961.

[33] Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind. AD in Fortran, Part 1: Design. Technical Report arXiv:1203.1448, 2012a.

[34] Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind. AD in Fortran: Implementation via prepreprocessor. In *Advances in Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, Fort Collins, Colorado, USA, July 23–27 2012b. Springer. . Also available as arxiv:1203.1450.

[35] Adil Reghai, Othmane Kettani, and Marouen Messaoud CVA with Greeks and AAD In *Risk*, Nov., 2015.

[36] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error-propagation. In D. E. Rumelhart and R. J McClelland, editors, *Parallel Distributed Processing*, chapter 8. MIT Press, Cambridge, Mass., 1986.

[37] Jeffrey Mark Siskind and Barak A. Pearlmutter. Binomial checkpointing for arbitrary programs with no user annotation. Technical Report arXiv:1611.03410, September 2016a. Extended abstract presented at the AD 2016 Conference, Oxford UK.

[38] Jeffrey Mark Siskind and Barak A. Pearlmutter. Efficient implementation of a higher-order language with built-in AD. Technical Report arXiv:1611.03416, September 2016b. Extended abstract presented at the AD 2016 Conference, Oxford UK.

[39] Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1980.

[40] R. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.