15 Years of Adjoint Algorithmic Differentiation in Finance

Luca Capriotti [1*] and Mike Giles [2†]

[1] *Columbia University, New York, New York 10027, United States of America*
[2] *Oxford University Mathematical Institute, Woodstock Road, Oxford OX2 6GG, United Kingdom*

(Dated: January 22, 2024)

Following the seminal "Smoking Adjoint" paper by Giles and Glasserman, 2006, the development of Adjoint Algorithmic Differentiation (AAD) has revolutionized the way risk is computed in the financial industry. In this paper, we provide a tutorial of this technique, illustrate how it is immediately applicable for Monte Carlo and Partial Differential Equations applications, the two main numerical techniques used for option pricing, and review the most significant literature in quantitative finance of the past fifteen years.

In memory of Professor Sandro Sorella, extraordinary physicist, kindest teacher and friend to one of us (L.C.), who has recently left us, way too soon.

Keywords: Algorithmic Differentiation, Monte Carlo Simulations, Partial Differential Equations, Derivatives Pricing, Calibration of Stochastic Models

Introduction

Quantitatively sound risk management practices come with formidable computational challenges and a high operational cost. Risk management of complex portfolios requires the deployment of computationally intensive numerical schemes such as Monte Carlo (MC) simulations or Partial Differential Equations (PDE) solvers. Moreover, standard approaches for the calculation of risk require repeating the calculation of the Profit & Loss (P&L) of a portfolio under hundreds of market scenarios in order to form finite differences estimators of the required sensitivities. In many cases, even after deploying vast amounts of computer power, these calculations cannot be completed in a practical amount of time (Andreasen, 2023). Since the total cost of the through-the-life risk management can determine whether it is profitable to execute a new trade, solving this technology problem is critical to allow a securities firm to remain competitive in the market.

Several faster alternative methods for the calculation of price sensitivities, especially in the context of MC simulations, have been proposed in the literature (for a review see, e.g., Glasserman, 2004). Among these, the pathwise derivative method (Broadie and Glasserman, 1996) provides unbiased estimates at a computational cost that in some specific applications is smaller than the one of standard finite difference approaches. However, in the majority of the problems encountered in practice the standard pathwise derivative method provides limited computational gains.

A much more efficient implementation of the pathwise derivative method was proposed in Giles and Glasserman, 2006 in the context of the Libor Market Model for European payouts. Although they cited the relevance of computer science research on algorithmic differentiation (Griewank, 2000), which Giles had used in engineering design optimisation (Giles *et al.*, 2003; Giles and Pierce, 2000), the paper illustrated the adjoint methodology using an algebraic approach which was later generalized to Bermudan options by Leclerc *et al.*, 2009 and extended by Denson and Joshi, 2011.

However, the key to make adjoint methods generally applicable is the adoption of algorithmic differentiation (Giering and Kaminski, 2006; Griewank, 2000; Griewank and Walther, 2008; Naumann, 2011) or AAD[1] as explained in several papers (Capriotti, 2008, 2011; Capriotti and Giles, 2010, 2012). This gives a prescriptive approach to new applications, by operating at the level of the computer instructions in an easy-to-learn programmatic way.

Over the past fifteen years, AAD has emerged as tremendously effective for speeding up the calculation of sensitivities in MC for a variety of specific applications (Antonov, 2016; Capriotti, 2011; Capriotti and Giles, 2012; Capriotti *et al.*, 2017; Cesa, 2017; Geeraert *et al.*, 2017; Goloubentsev and Lakshtanov, 2019; Henrard, 2011, 2017; Huge and Savine, 2020; Naumann and Toit, 2018) including correlation risk (Capriotti and Giles, 2010), and counterparty credit risk management (Capriotti *et al.*, 2011; Huge and Savine, 2017; Silotto *et al.*, 2023). Although the focus of most researchers has been in the context of MC applications, AAD, being general in nature, can be used to compute sensitivities in

*Electronic address: lc3635@columbia.edu
†Electronic address: mike.giles@maths.ox.ac.uk

[1] To the best of our knowledge the acronym AAD was first used in (Capriotti, 2008).

the context of any numerical scheme, such as PDE (Bain *et al.*, 2019; Capriotti *et al.*, 2015; Denson and Joshi, 2010), and turns out to be especially beneficial when applied to numerically demanding calibration schemes (Capriotti and Lee, 2014; Goloubentsev and Lakshtanov, 2022; Goloubentsev *et al.*, 2021b; Henrard, 2011).

In this paper, we review the ideas underlying AAD, and discuss several examples that will illustrate how AAD can be used as a *design paradigm* to implement the pathwise derivative method, for virtually any model or payoff of interest in finance, including path-dependent and multi asset products. This makes the implementation of fast calculation of risk a straightforward exercise.

I. ALGORITHMIC DIFFERENTIATION

Algorithmic Differentiation (AD) is a set of programming techniques first introduced in the early 60's aimed at computing accurately and efficiently the derivatives of a function given in the form of a computer program. The main idea underlying AD is that any such computer program can be interpreted as the composition of functions each of which is in turn a composition of basic arithmetic (addition, multiplication etc.), and intrinsic operations (logarithm, exponential, etc.). Hence, it is possible to calculate the derivatives of the outputs of the program with respect to its inputs by applying mechanically the rules of differentiation.

What makes AD particularly attractive when compared to standard (e.g., finite difference) methods for the calculation of the derivatives, is its computational efficiency. In fact, AD aims at exploiting the information on the structure of the computer function, and on the dependencies between its various parts, in order to optimize the calculation of the sensitivities. In particular, in some important cases, the calculation of the derivatives can be highly optimized by applying the *chain rule* of differentiation through the instructions of the program in an appropriate fashion.

In the following, we will review in more detail these ideas. In particular we will describe the two basic approaches to AD, the so-called *tangent* (or *forward*) and *adjoint* (or *backward*) modes. These differ by how the chain rule is applied to the composition of instructions representing a given function, and are characterized by different computational costs for a given set of computed derivatives. The books Griewank, 2000, Griewank and Walther, 2008 and Naumann, 2011 contain a complete introduction to AD. For a discussion of the fundamental complexity results for AD, see also (Naumann, 2008a,b, 2009).

Here, we will only recall the main results in order to clarify how this technique is beneficial in the implementation of the calculation of sensitivities for financial products. We will begin by stating the results regarding the computational efficiency of the two modes of AD, and we will justify them by discussing in detail a toy example.

A. Tangent and Adjoint Mode

Let us consider a computer program with n inputs, $x = (x_1, \ldots, x_n)$ and and m outputs $y = (y_1, \ldots, y_m)$, that is defined by a composition of arithmetic and non-linear (intrinsic) operations. Such a program can be seen as a function of the form $F : \mathbb{R}^n \to \mathbb{R}^m$,

$$(y_1, \ldots, y_m)^t = F(x_1, \ldots, x_n) \ . \tag{1}$$

In its simplest form, AD aims at producing a code evaluating the sensitivities of the outputs of the original program with respect to its inputs, i.e., at calculating the Jacobian of the function F

$$J_{ij} = \frac{\partial F_i(x)}{\partial x_j} \ , \tag{2}$$

with $F_i(x) = y_i$.

The tangent mode of AD allows the calculation of the function F and of its Jacobian with a cost – relative to the one for F – which can be shown, under a standard computational complexity model (Griewank and Walther, 2008; Naumann, 2011), to be bounded by a small constant, ω_T, times the number of *independent* variables, namely

$$\frac{\text{Cost}[F\&J]}{\text{Cost}[F]} \leq \omega_T \, n \ . \tag{3}$$

The value of the constant ω_T can be also bounded using a model of the relative cost of algebric operations, non linear unary functions, and memory access. This analysis gives (Griewank and Walther, 2008; Naumann, 2011) $\omega_T \in [2, 5/2]$.

The form of the result (3) appears quite natural as it is the same computational complexity of evaluating the Jacobian by perturbing one input variable at a time, repeating the calculation of the function, and forming the appropriate finite difference estimators.

Consistently with Eq. (3), the tangent mode of AD provides the derivatives of all the m components of the output vector y with respect to a single input x_j, i.e., a single column of the Jacobian (2), at a cost which is independent of the number of dependent variables, and bounded by a small constant, ω_T. In fact, the same holds true for *any linear combination* of the columns of the Jacobian, $\mathcal{L}_c(J)$, namely

$$\frac{\text{Cost}[F \,\&\, \mathcal{L}_c(J)]}{\text{Cost}[F]} \leq \omega_T \ . \tag{4}$$

This makes the tangent mode particularly well suited for the calculation of (linear combinations of) the columns of the Jacobian matrix (2). Conversely, it is generally not the method of choice for the calculation of the gradients [i.e., the rows of the Jacobian (2)] of functions of a large number of variables.

On the other hand, the adjoint mode of AD, or AAD, is characterized by a computational cost of the form (Griewank and Walther, 2008; Naumann, 2011)

$$\frac{\text{Cost}[F \,\&\, J]}{\text{Cost}[F]} \leq \omega_A \, m \ , \tag{5}$$

with $\omega_A \in [3,4]$, i.e., AAD allows the calculation of the function F and of its Jacobian with a cost – relative to the one for F – which is bounded by a small constant times the number of *dependent* variables.

As a result, AAD provides the full gradient of a scalar ($m = 1$) function at a cost which is just a small constant times the cost of evaluating the function itself. Remarkably such relative cost is *independent* of the number of components of the gradient. This is to be compared with the cost of evaluating the same gradient by finite differences or by means of the tangent mode (3), scaling linearly with the number n of sensitivities.

For vector valued functions, AAD provides the gradient of arbitrary linear combinations of the rows of the Jacobian, $\mathcal{L}_r(J)$, at the same computational cost of a single row, namely

$$\frac{\text{Cost}[F \,\&\, \mathcal{L}_r(J)]}{\text{Cost}[F]} \leq \omega_A. \tag{6}$$

This clearly makes the adjoint mode particularly well-suited for the calculation of (linear combinations of) the rows of the Jacobian matrix (2). When the full Jacobian is required, the adjoint mode is likely to be more efficient than the tangent mode when the number of independent variables is significantly larger than the number of the dependent ones ($m \ll n$).

In the following Section, we will provide justification of these results by discussing in detail an explicit example. In particular, we will show that the remarkable difference in computational complexity of the tangent and adjoint mode arises from the different way of applying the chain rule of differentiation through the instructions of the function F.

B. How Algorithmic Differentiation Works: a Simple Example

Let us consider, as a specific example, the function $F : \mathbb{R}^2 \to \mathbb{R}^3$, $(y_1, y_2)^t = \big(F_1(x_1, x_2, x_3), F_2(x_1, x_2, x_3)\big)^t$ defined as

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2\log x_1 x_2 + 2\sin x_1 x_2 \\ 4\log^2 x_1 x_2 + \cos x_1 x_3 - 2x_3 - x_2 \end{pmatrix} . \tag{7}$$

1. Algorithmic Specification of Functions and Computational Graphs

Given a value of the input vector x, the output vector y is calculated by a computer code by means of a sequence of instructions. In particular, the execution of the program can be represented in terms of a set of scalar *internal variables*, $w_1, \ldots, w_N$, such that

$$w_i = x_i \ , \quad i = 1, \ldots, n \tag{8}$$
$$w_i = \Phi_i(\{w_j\}_{j \prec i}) \ , \quad i = n+1, \ldots, N \ . \tag{9}$$

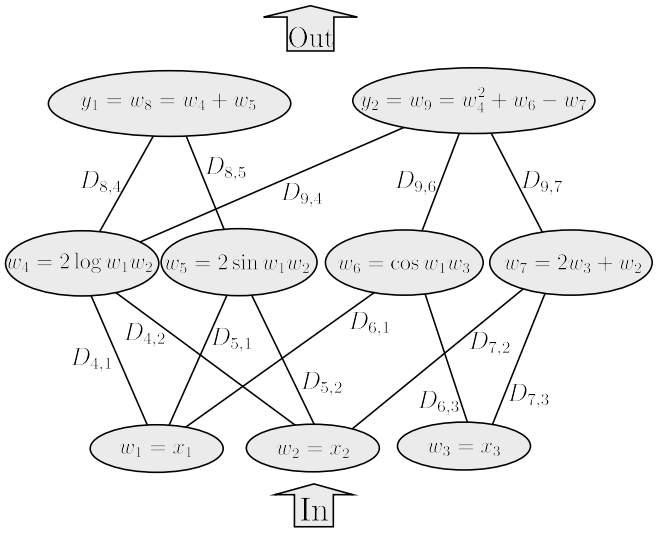

FIG. 1 Computational graph corresponding to the instructions (10) for the function in Eq. (7).

Here the first n variables are copies of the input ones, and the others are given by a sequence of consecutive assignments; the symbol $\{w_j\}_{j \prec i}$ indicates the set of internal variables w_j, with $j < i$, such that w_i depends *explicitly* on w_j; the functions Φ_i represent a composition of one or more elementary or intrinsic operations. In this representation, the last m internal variables are the output of the function, i.e., $y_{i-N+m} = w_i$, $i = N - m + 1, \ldots, N$. This representation is by no means unique, and can be constructed in a variety of ways. However, it is a useful abstraction in order to introduce the mechanism of AD. For instance, for the function (7), one can represent the internal calculations as follows:

$$\begin{gathered} w_1 = x_1 \ , \ w_2 = x_2 \ , \ w_3 = x_3 \ , \\ \downarrow \\ w_4 = 2\log w_1 w_2 \ , \\ w_5 = 2\sin w_1 w_2 \ , \\ w_6 = \cos w_1 w_3 \ , \\ w_7 = 2w_3 + w_2 \ , \\ \downarrow \\ y_1 = w_8 = w_4 + w_5 \ , \\ y_2 = w_9 = w_4^2 + w_6 - w_7 \ . \end{gathered} \tag{10}$$

In general, a computer program contains loops that may be executed a fixed or variable number of times, and internal controls that alter the calculations performed according to different criteria. Nevertheless, Eqs. (8) and (9) are an accurate representation on how the program is executed for a given value of the input vector x, i.e., for a given instance of the internal controls. In this respect, AD aims at performing a *piecewise* differentiation of the program, by reproducing the same controls in the differentiated code (Griewank and Walther, 2008; Naumann, 2011).

The sequence of instructions (8) and (9) can be effectively represented by means of a *computational graph*

with nodes given by the internal variables w_i, and connecting arcs between explicitly dependent variables. For instance, for the function in Eq. (7) the instructions (10) can be represented as in Fig. 1. Moreover, to each arc of the computational graph, say connecting node w_i and w_j with $j < i$, it is possible to associate the *arc derivative*

$$D_{i,j} = \frac{\partial \Phi_i(\{w_k\}_{k \prec i})}{\partial w_j} \ , \tag{11}$$

as illustrated in Fig. 1. Crucially, these derivatives can be calculated in an automatic fashion by applying mechanically the rules of differentiation instruction by instruction.

2. Tangent Mode

Once the program implementing $F(x)$ is represented in terms of the instructions Eqs. (8) and (9) (or with a computational graph like the one in Fig. 1) the calculation of the gradient of each of its m components,

$$\nabla F_i(x) = (\partial_{x_1} F_i(x), \partial_{x_2} F_i(x), \ldots, \partial_{x_n} F_i(x))^t \ , \tag{12}$$

simply involves the application of the chain rule of differentiation. In particular, by applying the rule starting from the independent variables, one obtains the *tangent mode* of AD

$$\nabla w_i = e_i \ , \quad i = 1, \ldots, n \tag{13}$$

$$\nabla w_i = \sum_{j \prec i} D_{i,j} \nabla w_j \ , \quad i = n+1, \ldots, N \tag{14}$$

where $e_1, e_2, \ldots, e_n$ are the vectors of the canonical basis in $\mathbb{R}^n$, and $D_{i,j}$ are the local derivatives (11). For the example in Eq. (7) this gives for instance:

$$\nabla w_1 = (1,0,0)^t \ , \ \nabla w_2 = (0,1,0)^t \ , \ \nabla w_3 = (0,0,1)^t \ ,$$
$$\downarrow$$
$$D_{4,1} = 2w_2/(w_1 w_2), \quad D_{4,2} = 2w_1/(w_1 w_2),$$
$$\nabla w_4 = D_{4,1} \nabla w_1 + D_{4,2} \nabla w_2 \ ,$$

$$D_{5,1} = 2w_2 \cos w_1 w_2, \quad D_{5,2} = 2w_1 \cos w_1 w_2,$$
$$\nabla w_5 = D_{5,1} \nabla w_1 + D_{5,2} \nabla w_2 \ ,$$

$$D_{6,1} = -w_3 \sin w_1 w_3, \quad D_{6,3} = -w_1 \sin w_1 w_3,$$
$$\nabla w_6 = D_{6,1} \nabla w_1 + D_{6,3} \nabla w_3 \ ,$$

$$D_{7,2} = 1, \quad D_{7,3} = 2,$$
$$\nabla w_7 = D_{7,2} \nabla w_2 + D_{7,3} \nabla w_3 \ ,$$

$$\downarrow$$
$$D_{8,4} = 1, \quad D_{8,5} = 1,$$
$$\nabla y_1 = \nabla w_8 = D_{8,4} \nabla w_4 + D_{8,5} \nabla w_5 \ ,$$

$$D_{9,4} = 2w_4, \quad D_{9,6} = 1, \quad D_{9,7} = -1,$$
$$\nabla y_2 = \nabla w_9 = D_{9,4} \nabla w_4 + D_{9,6} \nabla w_6 + D_{9,7} \nabla w_7 \ .$$

This leads to

$$\begin{aligned} \nabla y_1 &= (D_{8,4} D_{4,1} + D_{8,5} D_{5,1}, D_{8,4} D_{4,2} + D_{8,5} D_{5,2}, 0)^t \\ \nabla y_2 &= (D_{9,4} D_{4,1} + D_{9,6} D_{6,1}, \\ &\quad D_{9,4} D_{4,2} + D_{9,7} D_{7,2}, D_{9,6} D_{6,3} + D_{9,7} D_{7,3})^t \end{aligned}$$

which gives the correct result, as it can be immediately verified.

In the relations above each component of the gradient is computed independently. As a result, the computational cost of evaluating the Jacobian of the function F is approximately n times the cost of evaluating one of its columns, or any linear combination of them. For this reason, the computation in the tangent mode is more conveniently expressed by replacing the vectors ∇w_i with the scalars

$$\dot{w}_i = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} \ , \tag{15}$$

also known as *tangents*. Here λ is a vector in $\mathbb{R}^n$ specifying the chosen linear combination of columns of the Jacobian. Indeed, with this notation, the computation of the chain rule (13) and (14) becomes

$$\dot{w}_i = \lambda_i \ , \quad i = 1, \ldots, n \tag{16}$$

$$\dot{w}_i = \sum_{j \prec i} D_{i,j} \dot{w}_j \ , \quad i = n+1, \ldots, N \ . \tag{17}$$

At the end of the computation one finds therefore $\dot{w}_i$, $i = N - m + 1, \ldots, N$,

$$\dot{w}_i = \dot{y}_{i-N+m} = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} = \sum_{j=1}^{n} \lambda_j \frac{\partial y_{i-N+m}}{\partial x_j} \tag{18}$$

i.e., a linear combination of the *columns* of the Jacobian.

As illustrated in Fig. 2, the computation of the chain rule (16) and (17) allows one to associate to each node of the computational graph, the tangent of the corresponding internal variable, say $\dot{w}_i$. This can be calculated as a weighted average of the tangents of the variables preceding it on the graph (i.e., all the $\dot{w}_j$ such that $i \succ j$), with weights given by the arc derivatives associated with the connecting arcs. As a result, the tangents 'propagate' through the computational graph from the independent variables to the dependent ones, i.e., in the same direction followed in the evaluation of the original function, or *forward*. The computation of the tangents can in fact proceed instruction by instruction, at the same time when the function is evaluated.

It is easy to realize that the cost for the computation of the chain rule (16) and (17), for a given linear combination of the columns of the Jacobian is of the same order of the cost of evaluating the function F itself. Hence, for the simple example considered here, Eq. (4) represents an appropriate estimate of the computational cost of any linear combination of columns of the Jacobian. On the other hand, in order to get each column of the Jacobian

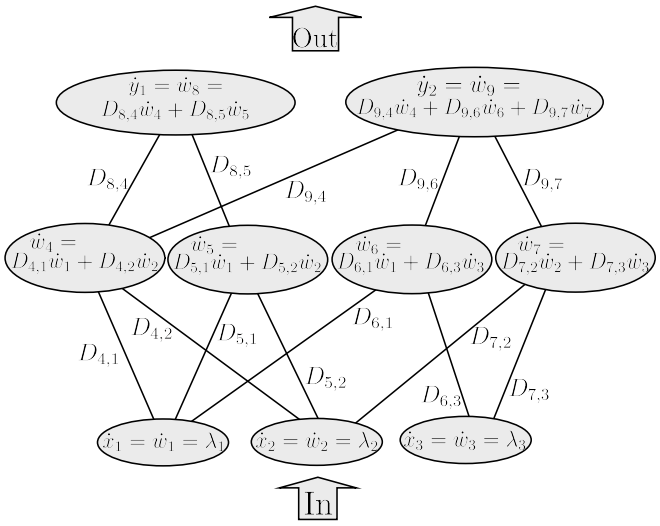

FIG. 2 Computational graph for the tangent mode differentiation of the function in Eq. (7).

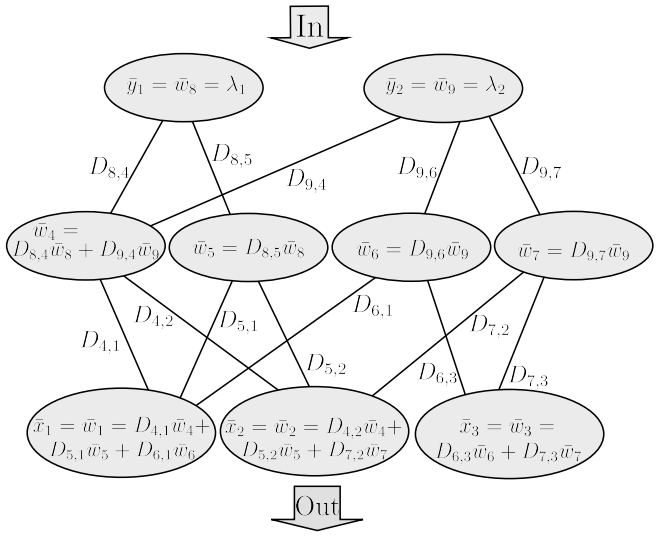

FIG. 3 Computational graph for the adjoint mode differentiation of the function in Eq. (7).

one has to repeat $n = 3$ times the calculation of the computational graph in Fig. 2, e.g., by setting λ equal to each vector of the canonical basis in $\mathbb{R}^3$. As a result, the computational cost of evaluating the Jacobian relative to the cost of evaluating the function F is proportional to the number of independent variables as predicted by Eq. (3).

We finally remark that, by performing simultaneously the calculation of all the components of the gradient (or, more in general, of a set of n linear combinations of columns of the Jacobian) one can optimize the calculation by reusing a certain amount of computations (for instance the arc derivatives). This leads to a more efficient implementation also known as *tangent multimode* (Griewank and Walther, 2008; Naumann, 2011). Although the computational cost for the tangent multimode remains of the form (3) and (4), the constant ω_T for these implementations is generally smaller than in the standard tangent mode see e.g., (Capriotti, 2011).

3. Adjoint Mode

The adjoint mode provides the Jacobian of a function in a mathematically equivalent way by means of a different sequence of operations. More precisely, the adjoint mode results from computing the derivatives of the final result with respect to all the intermediate variables – the so called *adjoints* – until the derivatives with respect to the independent variables are formed. Formally, the adjoint of any intermediate variable w_i is defined as

$$\bar{w}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} \ , \tag{19}$$

where λ is a vector in $\mathbb{R}^m$. In particular, for each of the dependent variables one has $\bar{y}_i = \lambda_i$, $i = 1, \ldots, m$, while, for the intermediate variables one has instead

$$\bar{w}_i = \frac{\partial y}{\partial w_i} = \sum_{j \succ i} \frac{\partial y}{\partial w_j} \frac{\partial w_j}{\partial w_i} = \sum_{j \succ i} D_{j,i} \bar{w}_j \ , \tag{20}$$

where the sum runs on the indices $j > i$ such that w_j depends explicitly on w_i. At the end of the computation one finds therefore $\bar{w}_i$, $i = 1, \ldots, n$,

$$\bar{w}_i = \bar{x}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial x_i} \ , \tag{21}$$

i.e., a given linear combination of the *rows* of the Jacobian (2).

For the example in Eq. (7) this gives in particular:

$$\begin{gathered}
\bar{w}_8 = \bar{y}_1 = \lambda_1 \ , \quad \bar{w}_9 = \bar{y}_2 = \lambda_2 \\
\downarrow \\
\bar{w}_4 = D_{8,4}\bar{w}_8 + D_{9,4}\bar{w}_9 \ , \\
\bar{w}_5 = D_{8,5}\bar{w}_8 \ , \bar{w}_6 = D_{9,6}\bar{w}_9 \ , \bar{w}_7 = D_{9,7}\bar{w}_9 \ , \\
\downarrow \\
\bar{w}_1 = \bar{x}_1 = D_{4,1}\bar{w}_4 + D_{5,1}\bar{w}_5 + D_{6,1}\bar{w}_6 \\
\bar{w}_2 = \bar{x}_2 = D_{4,2}\bar{w}_4 + D_{5,2}\bar{w}_5 + D_{7,2}\bar{w}_7 \\
\bar{w}_3 = \bar{x}_3 = D_{6,3}\bar{w}_6 + D_{7,3}\bar{w}_7 \ .
\end{gathered}$$

It is immediate to verify that by setting $\lambda = e_1$ and $\lambda = e_2$ (with e_1 and e_2 canonical vectors in $\mathbb{R}^2$), the adjoints $(\bar{w}_1, \bar{w}_2, \bar{w}_3)$ above give the components of the gradients of ∇y_1 and ∇y_2, respectively.

As illustrated in Fig. 3, Eq. (20) has a clear interpretation in terms of the computational graph: the adjoint of a quantity on a given node, $\bar{w}_i$, can be calculated as a weighted sum of the adjoints of the quantities that depend on it (i.e., all the $\bar{w}_j$ such that $j \succ i$), with weights given by the local derivatives associated with the respective arcs. As a result, the adjoints propagate through the computational graph from the dependent variables

to the independent ones, i.e., in the opposite direction with respect to the one of evaluation of the original function, or *backward*. The main consequence of this is that, in contrast to the tangent mode, the computation of the adjoints cannot be in general simultaneous with the execution of the function. Indeed, the adjoint of each node depends on variables that are yet to be determined on the computational graph. As a result, the computation of the adjoints can in general begin only after the construction of the computational graph has been completed, and the information on the value and dependences of the nodes on the graph, e.g., the arc derivatives, has been appropriately stored.

It is easy to realize that the cost for the computation of the chain rule (20) for a given linear combination of the rows of the Jacobian is of the same order of the cost of evaluating the function F itself, in agreement with Eq. (6). On the other hand, in order to get each row of the Jacobian, one has to repeat $m = 2$ times the calculation of the computational graph in Fig. 3, e.g., by setting λ equal to each vector of the canonical basis in $\mathbb{R}^2$. As a result, the computational cost of evaluating the Jacobian relative to the cost of evaluating the function F itself is proportional to the number of dependent variables, as predicted by Eq. (5).

C. Algorithmic Differentiation Tools

As illustrated in the previous examples, AD gives a clear set of prescriptions by which, given any computer function, one can develop the code implementing the tangent or adjoint mode for the calculation of its derivatives. This involves representing the computer function in terms of its computational graph, calculating the derivatives associated with each of the elementary arcs, and computing either the tangents or the adjoints in the appropriate direction. This procedure, being mechanical in nature, can be automated (Griewank and Walther, 2008; Naumann, 2011; Savine, 2018).

Several AD tools have been developed that allow the automatic implementation of the calculation of derivatives either in the tangent or in the adjoint mode. These tools falls in two main categories, namely *source code transformation* and *operator overloading*. An excellent source of information in the field can be found at `www.autodiff.org`.

Source code transformation tools are computer programs that take as an input the source code of a function, and return the source code implementing its derivatives. These tools rely on parsing the instructions of the input code and constructing a representation of the associated computational graph. In particular, an AD tool typically splits each instruction into the constituent unary or binary elementary operations for which the corresponding derivatives functions are known.

In the tangent mode, for each elementary instruction, the AD tool generates the code calculating the tangent of the output variable given the tangents of the input ones. This involves the mechanical application of a finite set of rules, and encoding the derivatives of the intrinsic unary and binary operations. For instance, for an elementary instruction of the form

$$w_1 = \sin w_2$$

the source code transformation tool will typically generate a code of the form

$$\begin{aligned} w_1 &= \sin w_2 \\ \dot{w}_1 &= \cos w_2 \ \dot{w}_2 \end{aligned}$$

evaluating simultaneously the original function, and computing the associated tangents.

In the adjoint mode, AD tools typically produce first a forward sweep that replicates the original function decorated with a record of the arc derivatives (or of the required information to calculate them on the fly in the backward sweep) in a data structure called the *tape*. Then, the tool produces the backward sweep by inverting the order of the instructions of the forward sweep. For each elementary instruction, the AD tool generates code for the calculation of the adjoint of the inputs given the adjoint of the output and the saved value of the arc derivatives. In the example above, the forward sweep could be for instance of the form

$$\begin{aligned} w_1 &= \sin w_2 \\ \text{save a record for} \quad & \cos w_2 \ . \end{aligned}$$

The corresponding instruction in the backward sweep would instead read

$$\begin{aligned} \text{retrieve the record for} \quad & \cos w_2 \\ \bar{w}_2 &= \cos w_2 \ \bar{w}_1 \ . \end{aligned}$$

On the other hand, the operator overloading approach exploits the flexibility of object oriented languages in order to introduce new abstract data types suitable to represent tangents and adjoints. Standard operations and intrinsic functions are then defined for the new types in order to allow the calculation of the tangents and the adjoints associated with any elementary instruction in a code. These tools operate by linking a suitable set of libraries to the source code of the function to be differentiated, and by redefining the type of the internal variables. Utility functions are generally provided to retrieve the value of the desired derivatives.

How this is possible is in fact very easy to understand for the tangent mode, as no information needs to be stored during the function evaluation for the calculation of the tangents. An operator overloading implementation is based on the definition of a new data type holding information on a certain variable w, and on its tangent $\dot{w} = \sum_{j=1}^{n} \lambda_j \partial w / \partial x_j$ for a given vector x on which w may depend on, and for a given set of weights λ, specifying a linear combination of derivatives. Given such a type, it is then easy to extend the algebra of real number in order to compute consistently both components of the pair

$\widetilde{w} = (w, \dot{w})$. This can be formally done by considering the algebra of the dual real numbers $\widetilde{w} = (w, \dot{w}) \equiv w + d\dot{w}$ defined by $d^2 = 0$. This way, for instance, the ordinary multiplication between real numbers is extended to the new types as

$$\begin{aligned} \widetilde{w}_1 \cdot \widetilde{w}_2 &= (w_1 + d\dot{w}_1) \cdot (w_2 + d\dot{w}_2) = \\ &= w_1 w_2 + d(w_1 \dot{w}_2 + \dot{w}_1 w_2) + d^2(\dot{w}_1 \dot{w}_2) \\ &= (w_1 w_2, w_1 \dot{w}_2 + \dot{w}_1 w_2) \ , \end{aligned}$$

while, for a generic function assignment $w_2 = f(w_1)$ the relation

$$\widetilde{w}_2 = f(\widetilde{w}_1) = f(w_1) + df'(w_1)\dot{w}_1 = (f(w_1), f'(w_1)\dot{w}_1) \ , \tag{22}$$

that can be formally derived from the Taylor expansion of $f(w_1 + d\dot{w}_1)$ in w_1, defines the corresponding assignment on the new types.

Source code transformation and operator overloading are both the subject of active research in the field of AD. Operator overloading is appealing for the simplicity of usage that boils down to linking some libraries, redefining the types of the variables, and calling some utility functions to access the derivatives. The main drawback is the lack of transparency, and the fact that the calculation of derivatives is generally slower than in the source code transformation approach, and more memory intensive. Source code transformation involves more work but it is generally more transparent as it provides the code implementing the calculation of the derivatives as a sequence of elementary instructions. This simplicity facilitates compiler optimization thus generally resulting in a faster execution.

In the context of computational finance, the application of several tools to financial software has been documented see, e.g., (Goloubentsev and Lakshtanov, 2019; Naumann and Toit, 2018).

In particular, recent developments in the runtime code generation approach (Goloubentsev and Lakshtanov, 2019) combine code transformation, operator overloading, and just-in-time compilation. Here, the operator overloading approach is used to extract the computation graph that represents the function at run-time, and then a new program is compiled to represent the function and its adjoint. This results in an optimized program for the *original* function because it is largely freed from high-level programming language abstractions, and only contains mathematical operations, and inlines specific parameters known at runtime. As a result, such optimized function can be much faster to execute than the original function even before applying the AAD transformations. Such function can be used for the forward sweep whether or not AAD sensitivities are required. The compilation time is a fixed run-time cost, which can be negligible if the function is computed a large number of times, e.g. as part of Monte Carlo sampling. As a result, in such situations the cost of computing sensitivities via AAD using the *optimized* function as a basis can be smaller than computing the *original* function (Goloubentsev and Lakshtanov, 2019).

The application of such automatic AD tools on large inhomogeneous computer codes, like the ones used in financial practice, is challenging. Indeed, pricing applications are rarely available as self contained packages, e.g., that can be easily parsed by an automatic AD tool. On the contrary, pricing applications generally consist of several independent components that are typically reusable in different contexts. Furthermore, they are possibly written in more than one programming language in an often heterogeneous programming environment, involving GPU, FPGA and cloud computing. In some cases the source code may not be even available as in the case when a third party library is used.

Fortunately, as we will discuss in the next Section, the principles of AD can be used as a programming paradigm for any algorithm and hand coding adjoints is a perfectly viable software development strategy when the analytics library is in a stable state, as it is usually the case in practice for mature, production standard, libraries. AD tools remain very useful nonetheless for the parts of the code that are self contained, especially if they are subject to frequent changes, like the implementation of the payoff functions, which are often tweaked to meet the demands of investors.

D. An introduction to the AAD Design Principles

An easy way to illustrate the adjoint design paradigm is to consider an arbitrary function

$$Y = \texttt{FUNCTION}(X) \tag{23}$$

mapping a vector X in $\mathbb{R}^n$ to a vector Y in $\mathbb{R}^m$ through a sequence of steps

$$X \ \rightarrow \ \ldots \ \rightarrow \ U \ \rightarrow \ V \ \rightarrow \ \ldots \ \rightarrow \ Y, \tag{24}$$

and to imagine that this represents a certain high level algorithm that we want to differentiate. Here, the real vectors U and V represent intermediate variables used in the calculation and each step can be a distinct high-level function or even an individual instruction.

The adjoint mode of AD results from computing the derivatives of the final output with respect to all the intermediate variables – the so called *adjoints* – until the derivatives with respect to the independent variables are formed. Using the standard AD notation, the adjoint of any intermediate variable V_k is defined as

$$\bar{V}_k = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial V_k}, \tag{25}$$

where $\bar{Y}$ is a vector in $\mathbb{R}^m$. In particular, for each of the intermediate variables U_i, using the chain rule we get,

$$\bar{U}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial U_i} = \sum_{j=1}^{m} \bar{Y}_j \sum_k \frac{\partial Y_j}{\partial V_k} \ \frac{\partial V_k}{\partial U_i},$$

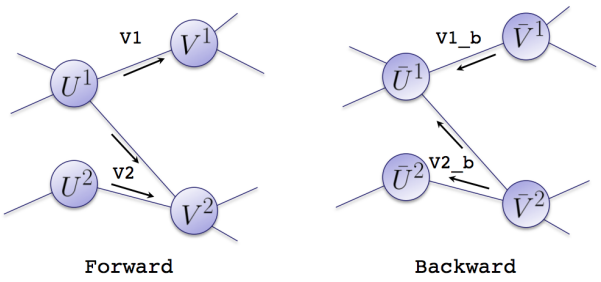

FIG. 4 Schematic illustration of the functions in Eqs. (29) and (30) (left) and their adjoint Eqs. (32) and (31) (right).

which corresponds to the adjoint mode equation for the intermediate step represented by the function $V = V(U)$

$$\bar{U}_i = \sum_k \bar{V}_k \frac{\partial V_k}{\partial U_i},$$

namely a function of the form $\bar{U} = \bar{V}(U, \bar{V})$. Starting from the adjoint of the outputs, $\bar{Y}$, we can apply this rule to each step in the calculation, working from right to left,

$$\bar{X} \leftarrow \ldots \leftarrow \bar{U} \leftarrow \bar{V} \leftarrow \ldots \leftarrow \bar{Y} \tag{26}$$

until we obtain $\bar{X}$, i.e., the following linear combination of the rows of the Jacobian of the function $X \to Y$:

$$\bar{X}_i = \sum_{j=1}^{m} \bar{Y}_j \frac{\partial Y_j}{\partial X_i}, \tag{27}$$

with $i = 1, \ldots, n$. This calculation can be seen as the implementation of the adjoint counterpart of the function (23), namely

$$\bar{X} = \texttt{FUNCTION\_b}(X, \bar{Y}) \ . \tag{28}$$

As noted before, in the adjoint mode, the cost does not increase with the number of inputs, but it is linear in the number of (linear combinations of the) rows of the Jacobian that need to be evaluated independently. In particular, if the full Jacobian is required, one needs to repeat the adjoint calculation m times, setting the vector $\bar{Y}$ equal to each of the elements of the canonical basis in $\mathbb{R}^m$. Furthermore, since the partial derivatives depend on the values of the intermediate variables, one generally first has to compute the original calculation storing the values of all of the intermediate variables such as U and V, before performing the adjoint mode sensitivity calculation.

By appropriately defining the intermediate variables, any algorithm can be abstracted in general as a composition of functions as in (24). However, the actual calculation graph might have a more complex structure. For instance the step $U \to V$ might be implemented in terms of two computer functions of the form

$$V^1 := \texttt{V1}(U^1) \ , \tag{29}$$
$$V^2 := \texttt{V2}(U^1, U^2) \ , \tag{30}$$

with $U = (U^1, U^2)^t$ and $V = (V^1, V^2)^t$. Here the notation $W = (W^1, W^2)^t$ simply indicates a specific partition of the components of the vector W in two sub-vectors. As illustrated in Fig. 4, a natural way to represent the step $\bar{U} \leftarrow \bar{V}$ in (26), i.e., the function $\bar{U} = \bar{V}(U, \bar{V})$, can be given in terms of an adjoint calculation graph. This adjoint graph has the same structure of the original graph with each node/variable representing the adjoint of the original node/variable, and it is executed in opposite direction with respect to the original one. The relation between the adjoint nodes is defined by the correspondence between Eqs. (23) and (28), e.g., in the specific example

$$(\bar{U}^1, \bar{U}^2)^t := \texttt{V2\_b}(U^1, U^2, \bar{V}^2) \ , \tag{31}$$
$$\bar{U}^1 := \bar{U}^1 + \texttt{V1\_b}(U^1, \bar{V}^1) \ . \tag{32}$$

Here, the above notation indicates that the adjoint $\bar{U}^1$ has two contributions, one for each node preceding it in the adjoint graph.

The structure of the adjoint calculation graph as given by equations (31) and (32) can be understood as follows. The variable U^1 is an input of two distinct functions in the instructions (29) and (30) so that, by applying the definition of adjoint (27) for the variable U^1 as an input of the function $V = V(U^1, U^2) = (V^1(U^1), V^2(U^1, U^2))^t$, we get

$$\bar{U}^1 = \sum_j \bar{V}_j \frac{\partial V_j}{\partial U^1} = \sum_k \bar{V}^1_k \frac{\partial V^1_k}{\partial U^1} + \sum_k \bar{V}^2_k \frac{\partial V^2_k}{\partial U^1} \tag{33}$$

where we have simply partitioned the components of the vector V as $(V^1, V^2)^t$ for the second equality. Similarly, one has for $\bar{U}^2$

$$\bar{U}^2 = \sum_j \bar{V}_j \frac{\partial V_j}{\partial U^2} = \sum_k \bar{V}^2_k \frac{\partial V^2_k}{\partial U^2}, \tag{34}$$

where we have used the fact that V^1 has no dependence on U^2. Therefore, one can realize that the adjoint calculation graph implementing the instructions in (31) and (32) indeed produces the adjoint $\bar{U} = (\bar{U}^1, \bar{U}^2)^t$.

The adjoint instructions (31) and (32) depend on the variables U^1 and U^2. As a result, as previously mentioned, the adjoint algorithm (26) can be executed only after the original instructions (24) have been executed and the necessary intermediate results have been computed and stored. This is the reason why the adjoint of a given algorithm generally contains a *forward sweep*, which reproduces the steps of the original algorithm, plus a *backward sweep*, which computes the adjoints.

The construction described above can be applied recursively for each of the functions involved in the calculation, see e.g., Fig. 5. Here it is clear that one needs to

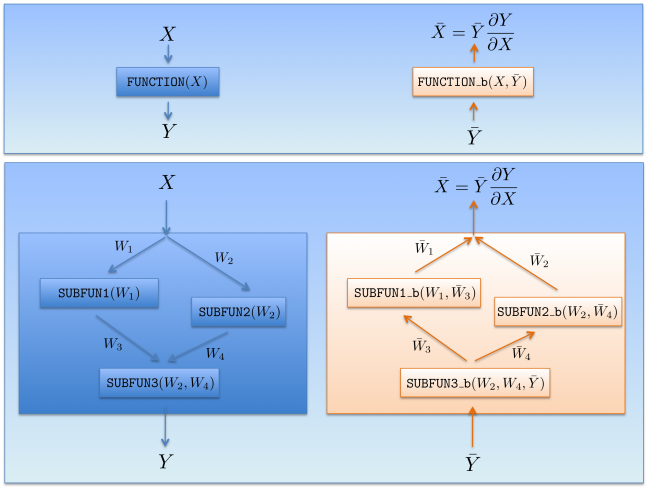

FIG. 5 Schematic illustration of recursive application of AAD design principles.

define an adjoint counterpart to each function of the original program with the dependencies among their inputs and outputs defined by the rules described above. Following this design paradigm it is therefore easy to write the blueprint of any numerical algorithm.

In particular, each adjoint function, taken in isolation, contains in turn a forward sweep recovering the information that is necessary for the computation of the adjoints. However, this is clearly suboptimal since all the information necessary to perform the adjoint of the algorithm is computed when performing the forward sweep of the algorithm as a whole. Hence, this information could be saved during this stage. This way, when the adjoint functions are invoked during the backward sweep there is no need to perform the functions' forward sweeps again. Strictly speaking, this is necessary to ensure that the computational cost of the overall algorithm remains within the expected bounds. However, there is a trade-off between the time and space necessary to store and retrieve this information and the time to recalculate it from scratch. Thus, in practice it is useful to store in the forward sweep only the results of relatively expensive calculations (for more details about this technique, known as "checkpointing" please see (Capriotti and Giles, 2012) and the reference text books (Griewank, 2000; Naumann, 2011)).

In the next section we will illustrate how these ideas can be employed for the efficient implementation of the pathwise derivative method.

II. MONTE CARLO APPLICATIONS

A. Pathwise Derivative Method

Option pricing problems can be typically formulated in terms of the calculation of expectation values of the form

$$V = \mathbb{E}_{\mathbb{Q}}\Big[P(X(T_1), \ldots, X(T_M))\Big]. \tag{35}$$

Here $X(t)$ is a N-dimensional vector and represents the value of a set of underlying market factors (e.g., stock prices, interest rates, foreign exchange pairs, etc.) at time t. $P(X(T_1), \ldots, X(T_M))$ is the discounted payout function of the priced security, and depends in general on M observations of those factors. In the following, we will indicate the collection of such observations with a $d = N \times M$ dimensional state vector $X = (X(T_1), \ldots, X(T_M))^t$, and with $\mathbb{Q}(X)$ the appropriate risk neutral distribution according to which the components of X are distributed.

The expectation value in (35) can be estimated by means of MC by sampling a number $N_{\rm MC}$ of random replicas of the underlying state vector $X[1], \ldots, X[N_{\rm MC}]$, sampled according to the distribution $\mathbb{Q}(X)$, and evaluating the payout $P(X)$ for each of them. This leads to the estimate of the option value V as

$$V \simeq \frac{1}{N_{\rm MC}} \sum_{i_{\rm MC}=1}^{N_{\rm MC}} P\left(X[i_{\rm MC}]\right). \tag{36}$$

Due to the central limit theorem, the standard error is $\Sigma/\sqrt{N_{\rm MC}}$, where $\Sigma^2 = \mathbb{E}_{\mathbb{Q}}[P(X)^2] - \mathbb{E}_{\mathbb{Q}}[P(X)]^2$ is the variance of the sampled payout.

The pathwise derivative method allows the calculation of the sensitivities of the option price V (35) with respect to a set of N_θ parameters $\theta = (\theta_1, \ldots, \theta_{N_\theta})$, with a single simulation. Indeed, whenever the payout function is regular enough, e.g., Lipschitz-continuous, and under additional conditions that are often satisfied in financial pricing (see, e.g., Glasserman, 2004), one can write the sensitivity $\langle \bar{\theta}_k \rangle \equiv \partial V / \partial \theta_k$ as

$$\langle \bar{\theta}_k \rangle = \mathbb{E}_{\mathbb{Q}}\Big[\frac{\partial P_\theta(X)}{\partial \theta_k}\Big] \tag{37}$$

where we have added the subscript θ to P to allow for the possibility of an explicit dependence of the payout on θ.

In general, the calculation of Eq. (37) can be performed by applying the chain rule and averaging on each MC path the so-called pathwise derivative estimator

$$\bar{\theta}_k \equiv \frac{\partial P_\theta(X)}{\partial \theta_k} = \sum_{j=1}^{d} \frac{\partial P_\theta(X)}{\partial X_j} \times \frac{\partial X_j}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k}. \tag{38}$$

The second term in Eq. (38) arises due to the explicit dependence of the payout on θ.

As an example we can consider the special case in which the state vector $X = (X(T_1), \ldots, X(T_M))$ is a path of a N-dimensional diffusive process, of the form

$$dX(t) = \mu(X(t), t, \theta)\, dt + \sigma(X(t), t, \theta)\, dW_t, \tag{39}$$

with $X(t_0) = X_0$. Here the drift $\mu(X, t, \theta)$ and volatility $\sigma(X, t, \theta)$ are an N-dimensional vector and $N \times N$ matrix, respectively, and W_t is a N-dimensional Brownian motion with instantaneous correlation matrix $\rho(t)$ defined by $\rho(t)\, dt = \mathbb{E}_{\mathbb{Q}}\left[dW_t dW_t^t\right]$. In this case, the pathwise derivative estimator (38) may be rewritten as

$$\bar{\theta}_k = \sum_{l=1}^{M}\sum_{j=1}^{N} \frac{\partial\, P_\theta(X(T_1), \ldots, X(T_M))}{\partial X_j(T_l)} \frac{\partial X_j(T_l)}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k}\ , \tag{40}$$

where we have relabeled the d components of the state vector X grouping together different observations $X_j(T_1), \ldots, X_j(T_M)$ of the same (j-th) asset.

The pathwise derivative estimators of the sensitivities are mathematically equivalent[2] to the finite difference estimators (in the following also referred to as "bump and reval" or "bumping") defined using the same random numbers in both simulations, and for a vanishingly small perturbation. As a result, the implementation effort associated with the pathwise derivative method is generally justified if the computational cost of the estimator (38) is less than the corresponding one associated with bumping.

Apart from simple applications, naïve implementations of the pathwise derivative method do not generally lead to significant computational savings with respect to finite difference estimators. This is typically the case when the perturbation of a model parameter affects in a non trivial way a large number of components of the tangent process, as it is typically the case in interest rate, credit and commodity models.

For non path dependent options in the context of Libor Market Models, Giles and Glasserman, 2006 have shown that that the pathwise derivative method can be efficiently implemented by expressing the calculation of the estimator (38) in terms of linear algebra operations, and utilize adjoint methods to reduce the computational complexity by rearranging appropriately the order of the calculations. Recently these algebraic adjoint approaches, have been successfully generalized by Leclerc *et al.*, 2009 for the case of Bermudan swaptions , and by Denson and Joshi, 2011 for more accurate simulation schemes.

B. AAD and the Pathwise Derivative Method

General Design

AAD provides a general design and programming paradigm for the efficient implementation of the pathwise derivative method.

In a MC simulation, the evolution of the process X is usually simulated, possibly by means of an approximate discretization scheme (Glasserman, 2004), by sampling $X(t)$ on a discrete grid of points, $0 = t_0 < t_1 < \ldots < t_n < \cdots < t_{N_s}$, a superset of the observation times $(T_1, \ldots, T_M)$. Here $t_0 = 0$ indicates the time the expectation values (35) are conditional on, e.g., usually the time the expectation value is calculated. The left panel of Fig. 6 illustrates schematically the sequence of operations that are typically performed in order to generate each sample $X[i_{\mathrm{MC}}]$ in Eq. (36). Here the state vector at time t_{n+1} is obtained by means of a function of the form

$$X(t_{n+1}) = \mathtt{PROP}_n[\{X(t_m)\}_{m\le n}, Z(t_n), \theta], \tag{41}$$

mapping the set of state vector values on the discretization grid up to t_n, $\{X(t_m)\}_{m\le n}$, into the value of the state vector at time $t_n + 1$. Note that, in general, this is a function of the model parameters θ and of the particular time step considered. Here $Z(t_n)$ indicates the vector of uncorrelated random numbers which are used for the MC sampling in the step $n \to n+1$. Note that, the initial values of the state vector $X(t_0)$ are known quantities at the beginning of the simulation. As a result, they can be considered as components of the model parameter vector so that the $n = 0$ step is of the form,

$$X(t_1) = \mathtt{PROP}_0[Z(t_0), \theta], \tag{42}$$

as also indicated schematically in the figure.

Once the full set of state vector values on the simulation time grid $\{X(t_m)\}_{m\le N_s}$ is obtained, the subset of values corresponding to the observation dates is passed to the payout function, evaluating the payout estimator $P_\theta(X)$ for the specific random sample X

$$(X(T_1), \ldots, X(T_M)) \to P_\theta(X(T_1), \ldots, X(T_M)). \tag{43}$$

Overall, the evaluation of a MC sample of a pathwise estimator can be seen as an algorithm implementing a function of the form $\theta \to P(\theta)$. As a result, it is possible to design its adjoint counterpart $(\theta, \bar{P}) \to (P, \bar{\theta})$ which gives (for $\bar{P} = 1$) the pathwise derivative estimator in Eq. (37). The right panel of Fig. 6 illustrates the structure of the adjoint algorithm. This can be simply obtained by reversing the flow of the computations, and associating to each function its adjoint counterpart. In particular, the first step of the adjoint algorithm is the adjoint of the payout evaluation. This is a function of the form

$$(\bar{X}, \bar{\theta}) = \bar{P}(X, \theta), \tag{44}$$

where $\bar{X} = (\bar{X}(T_1), \ldots, X(T_M))$ is the adjoint of the state vector on the observation dates, and $\bar{\theta}$ is the adjoint of the model parameters vector, respectively

$$\bar{X}(T_m) = \frac{\partial P_\theta(X)}{\partial X(T_m)}, \tag{45}$$

$$\bar{\theta} = \frac{\partial P_\theta(X)}{\partial \theta}, \tag{46}$$

[2] Provided that the state vector is a regular enough function of θ (Glasserman, 2004).

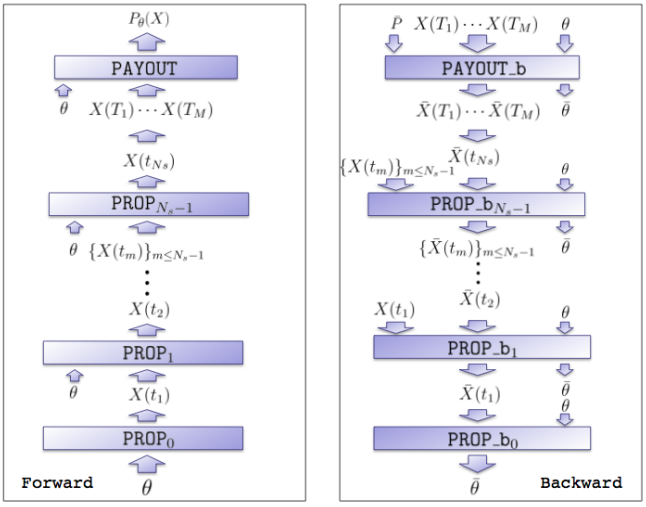

FIG. 6 Schematic illustration of the general AAD implementation of the pathwise derivative method.

for $m = 1, \ldots, M$. The adjoint of the state vector on the simulation dates corresponding to the observation dates are initialized at this stage. The remaining ones are initialized to zero.

The adjoint state vector is then calculated backwards in time through the adjoint of the propagation method (41), namely

$$(\{\bar{X}(t_m)\}_{m\le n}, \bar{\theta}) \mathrel{+}= \texttt{PROP\_b}_n[\{X(t_m)\}_{m\le n}, Z(t_n), \theta, \bar{X}(t_{n+1})], \quad (47)$$

for $n = N_s - 1, \ldots, 1$, giving

$$\bar{X}(t_m) \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial X(t_m)}, \quad (48)$$

with $m = 1, \ldots, n$,

$$\bar{\theta} \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial \theta}, \quad (49)$$

where we have used the notation += for the standard addition assignment operator. Here, according to the principles of AAD, the adjoint of the propagation method takes as arguments the inputs of its forward counterpart, namely the state vectors up to time t_n, $\{X(t_m)\}_{m\le n}$, the vector of random variates $Z(t_n)$, and the θ vector. The additional input is the adjoint of the state vector at time t_{n+1}, $\bar{X}(t_{n+1})$. The return values of $\texttt{PROP\_b}_n$ are the contributions associated with the step $n + 1 \to n$ to the adjoints of *i)* the state vector $\{\bar{X}(t_m)\}_{m\le n}$, Eq. (48); *ii)* the model parameters $\bar{\theta}_k$, $k = 1, \ldots, N_\theta$, Eq. (49).

The final step of the backward propagation corresponds to the adjoint of (42), giving

$$\bar{\theta} \mathrel{+}= \texttt{PROP\_b}_0[Z(t_0), \theta, \bar{X}(t_1)], \quad (50)$$

i.e., the final contribution to the adjoints of the model parameters. It is easy to verify that the final result is the pathwise derivative estimator in Eq. (38). The examples discussed in the following will provide a concrete examples for the steps described above.

C. Correlation Risk

In the description above we have assumed that the components of the vectors $Z(t_n)$, are uncorrelated standard normal random variables. Since these variables do not carry sensitivities to any model parameter, their adjoint is not calculated by $\texttt{PROP\_b}_n$. In a typical financial simulation setup, these variables are mapped into correlated counterparts $Z'(t_n)$ and then used to implement the propagation step $X(t_n) \to X(t_{n+1})$ so that the propagation step (41) is modified as

$$Z'(t_n) = \texttt{CORRELATE}(Z(t_n), \theta) \quad (51)$$
$$X(t_{n+1}) = \texttt{PROP}_n[\{X(t_m)\}_{m\le n}, Z'(t_n), \theta] \ , \quad (52)$$

where we have included the correlation parameters in the vector θ. The corresponding adjoint steps in the backward propagation read therefore

$$(\{\bar{X}(t_m)\}_{m\le n}, \bar{\theta}, \bar{Z}'(t_n)) \mathrel{+}= \texttt{PROP\_b}_n[\{X(t_m)\}_{m\le n}, Z'(t_n), \theta, \bar{X}(t_{n+1})] \ . \quad (53)$$

and

$$\bar{\theta} \mathrel{+}= \texttt{CORRELATE\_b}(Z(t_n), \theta, \bar{Z}'(t_n)), \quad (54)$$

The quantities $\{\bar{X}(t_m)\}_{m\le n}$ and $\bar{\theta}$ in Eq. (53) are given respectively by Eqs. (48) and (49) and

$$\bar{Z}'(t_n) \mathrel{+}= \sum_{j=1}^{N} \bar{X}_j(t_{n+1}) \frac{\partial X_j(t_{n+1})}{\partial Z'(t_n)}. \quad (55)$$

Equation (54) corresponds instead to

$$\bar{\theta} \mathrel{+}= \sum_{j=1}^{N} \bar{Z}'_j(t_n) \frac{\partial Z'_j(t_n)}{\partial \theta} \quad (56)$$

updating the adjoints of the correlation model parameters. In the common situation in which the variable $Z'(t_n)$ are jointly normal, the last step typically involves accumulating the adjoint of the Cholesky factor as discussed in (Capriotti and Giles, 2010).

We recall that the Cholesky factorization of a correlation matrix ρ produces a lower triangular matrix C such that $\rho = CC^t$ so that one can write $Z' = CZ$. The natural AAD approach would average the values of $\bar{C}$ from each of the MC paths:

$$\bar{C} = \bar{Z}' Z'^t \ . \quad (57)$$

This average $\bar{C}$ can be converted into derivatives with respect to the entries of the correlation matrix ρ by means

```
PROP(n, L[,], Z, lambda[], L0[])

  if(n=0)
    for(i= 1 .. N)
      L[i,n] = L0[i]; Lhat[i,n] = L0[i];

  for (i = 1 .. eta[n]-1)
    L[i,n+1] = L[i,n];  // settled rates

  sqez = sqrt(h)*Z;
  v = 0.; v_pc = 0.;
  for (i = eta[n] .. N)
    lam = lambda[i-eta[n]+1];
    c1 = del*lam; c2 = h*lam;
    v += (c1*L[i,n])/(1.+del*L[i,n]);
    vrat = exp(c2*(-lam/2.+v)+lam*sqez);
    // standard propagation with the Euler drifts
    Lhat[i,n+1] = L[i,n]*vrat;
    // (n + 1) drift term
    v_pc += (c1*Lhat[i,n+1])/(1.+del*Lhat[i,n+1]);
    vrat_pc = exp(c2*(-lam/2.+(v_pc+v)/2.)+lam*sqez);
    // actual propagation using the average drift
    L[i,n+1]  = L[i,n]*vrat_pc;
    // store what is needed for the reverse sweep
    hat_scra[i,n+1] = vrat*((v-lam)*h+sqez);
    scra[i,n+1] = vrat_pc*(((v_pc+v)/2.-lam)*h+sqez);
```

FIG. 7 Pseudocode implementing the propagation method $\texttt{PROP}_n$ (41) for the Libor Market Model of Eq. (63) for $d_W = 1$, under the predictor corrector Euler approximation (64), and the volatility parameterization (62).

of the adjoint of the Cholesky factorization (Smith, 1995), namely a function of the form

$$\bar{\rho} = \texttt{CHOLESKY\_B}(\rho, \bar{C}) \tag{58}$$

providing

$$\bar{\rho}_{i,j} = \sum_{l,m=1}^{N} \frac{\partial C_{l,m}}{\partial \rho_{i,j}} \bar{C}_{l,m} \ . \tag{59}$$

The pseudocode for the adjoint Cholesky factorization is given in (Capriotti and Giles, 2010). It is important to note that because the Cholesky factor is a path invariant quantity the Cholesky factorization and its adjoint are executed *once per simulation.*

D. Examples

1. Libor Market Model Simulation

As a second example, in order to make the connection with previous algebraic implementations of adjoint methods (Denson and Joshi, 2011; Giles and Glasserman, 2006; Leclerc *et al.*, 2009), we discuss the implementation of AAD in the Libor Market Model. Here we indicate with T_i, $i = 1, \ldots, N+1$, a set of $N+1$ bond maturities, with spacings $\delta = T_{i+1} - T_i$, assumed constant for simplicity. The dynamics of the Libor rate as seen at time t for the interval $[T_i, T_{i+1})$, $L_i(t)$, takes the form

$$\frac{dL_i(t)}{L_i(t)} = \mu_i(L(t))dt + \sigma_i(t)^t dW_t, \tag{60}$$

$0 \le t \le T_i$, and $i = 1, \ldots, N$, where W_t is a d_W-dimensional standard Brownian motion, $L(t)$ is the N-dimensional vector of Libor rates, and $\sigma_i(t)$ the d_W-dimensional vector of volatilities, at time t. Here the drift term in the spot measure, as imposed by the no arbitrage conditions (Brace *et al.*, 1997), reads

$$\mu_i(L(t)) = \sum_{j=\eta(t)}^{i} \frac{\sigma_i^t \sigma_j \delta L_j(t)}{1 + \delta L_j(t)}, \tag{61}$$

where $\eta(t)$ denotes the index of the bond maturity immediately following time t, with $T_{\eta(t)-1} \le t < T_\eta(t)$. As is common in the literature, to keep this example as simple as possible, we take each vector σ_i to be a function of time to maturity

$$\sigma_i(t) = \sigma_{i-\eta(t)+1}(0) = \lambda(i - \eta(t) + 1). \tag{62}$$

Equation (60) can be simulated by applying a Euler discretization to the logarithms of the forward rates, e.g., by dividing each interval $[T_i, T_{i+1})$ into N_s steps of equal width, $h = \delta/N_s$. This gives

$$\begin{aligned} \frac{L_i(t_{n+1})}{L_i(t_n)} &= \exp\left[\left(\mu_i(L(t_n)) - ||\sigma_i(t_n)||^2/2\right)h \right. \\ &+ \left. \sigma_i^t(n) Z(t_n)\sqrt{h}\right], \end{aligned} \tag{63}$$

for $i = \eta(nh), \ldots, N$, and $L_i(t_{n+1}) = L_i(t_n)$ if $i < \eta(nh)$. Here Z is a d_W-dimensional vector of independent standard normal variables.

In a recent paper, Denson and Joshi, 2011 extended the original adjoint implementation to the propagation of the Libor under the predictor-corrector drift approximation, consisting in replacing the drift in (61) with

$$\begin{aligned} \mu_i^{pc}(L(t_n)) &= \frac{1}{2} \sum_{j=\eta(nh)}^{i} \left(\frac{\sigma_i^t \sigma_j \delta L_j(t_n)}{1 + \delta L_j(t_n)} \right. \\ &+ \left. \frac{\sigma_i^t \sigma_j \delta \hat{L}_j(t_{n+1})}{1 + \delta \hat{L}_j(t_{n+1})} \right) \end{aligned} \tag{64}$$

where $\hat{L}_j(t_{n+1})$ is calculated from $L_j(t_n)$ using the evolution (63), i.e., with the simple Euler drift (61).

The pseudocode for the propagation of the Libor rates for $d_W = 1$, corresponding to the function PROP in (41), is shown in Fig. 7. Here, as discussed in Giles and Glasserman, 2006, the computational cost of implementing Eq. (63) is minimized by first evaluating

$$v_i(t_n) = \sum_{j=\eta(nh)}^{i} \frac{\sigma_j \delta L_j(t_n)}{1 + \delta L_j(t_n)}, \tag{65}$$

```
PROP_b(n, L[,], Z, lambda[], L0[], lambda_b[], L0_b[])

  v_b = 0.; v_pc_b = 0.;
  for (i=N .. eta[n])
    lam = lambda[i-eta[n]+1];
    c1 = del*lam; c2 = lam*h;
    //L[i,n+1] = L[i,n]*vrat_pc
    vrat_pc = L[i,n+1]/L[i,n];
    vrat_pc_b = L[i,n]*L_b[i,n+1];
    L_b[i,n] = vrat_pc*L_b[i,n+1];
    // vrat_pc = exp(c2*(-lam/2.+(v_pc+v)/2.)+lam*sqez)
    lambda_b[i-eta[n]+1] += scra[i,n+1]*vrat_pc_b;
    v_pc_b += vrat_pc*lam*h*vrat_pc_b/2.;
    v_b += vrat_pc*lam*h*vrat_pc_b/2.;
    // v_pc += (c1*Lhat[i,n+1])/(1.+del*Lhat[i,n+1])
    rpip = 1./(del*Lhat[i,n+1]+1.);
    Lhat_b[i,n+1] += (c1-c1*Lhat[i,n+1]*del*rpip)*rpip*v_pc_b;
    c1_b = Lhat[i,n+1]*rpip*v_pc_b;
    // Lhat[i,n+1] = L[i,n]*vrat
    vrat_b = L[i,n]*Lhat_b[i,n+1];
    vrat = Lhat[i,n+1]/L[i,n];
    L_b[i,n] += vrat*Lhat_b[i,n+1];
    // vrat = exp(lam*h*(-lam/2.+v)+lam*sqez)
    lambda_b[i-eta[n]+1] += hat_scra[i,n+1]*vrat_b;
    v_b += vrat*lam*h*vrat_b;
    // v += (c1*L[i,n])/(1.+del*L[i,n])
    rpip = 1./(del*L[i,n]+1.);
    L_b[i,n] += (c1-c1*L[i,n]*del*rpip)*rpip*v_b;
    c1_b += L[i+n]*rpip*v_b;
    // lam = lambda[i-eta[n]+1]; c1 = del*lam
    lambda_b[i-eta[n]+1] += del*c1_b;

  for (i=eta[n]-1 .. 1)
    // L[i,n+1] = L[i,n]
    L_b[i,n] += L_b[i,n];

  if(n=0)
    for(i=1 .. N)
      // L[i,n] = L0[i]
      L0_b[i] = L0[i,0];
```

FIG. 8 Adjoint of the propagation method `PROP_b`$_n$ (47) for the Libor Market Model of Eq. (60) for $d_W = 1$, under the predictor corrector Euler approximation (64), and the volatility parameterization (62). The corresponding forward method is shown in Fig. 7. The instructions commented are the forward counterpart to the adjoint instructions immediately after.

as a running sum for $i = \eta(nh), \ldots, N$, so that $\mu_i = \sigma_i^t v_i$.

The algebraic formulation discussed in Denson and Joshi, 2011 comes with a significant analytical effort. Instead, as illustrated in Fig. 8, the AAD implementation is quite straightforward. According to the general design of AAD, this simply consists of the adjoints of the instructions in the forward sweep executed in reverse order. In this example, the information computed by `PROP` that is required by `PROP_b` is stored in the vectors `scra` and `hat_scra`. By inspecting the structure of the pseudocode it also appears clear that the computational cost of `PROP_b` is of the same order as evaluating the original function `PROP`.

As a standard test case in the literature, here we have considered contracts with expiry T_n to enter in a swap with payments dates $T_{n+1}, \ldots, T_{N+1}$, with the holder of the option paying a fixed rate K

$$V(T_n) = \sum_{i=n+1}^{N+1} B(T_n, T_i)\delta(S_n(T_n) - K)^+, \tag{66}$$

where $B(T_n, T_i)$ is the price at time T_n of a bond maturing at time T_i

$$B(T_n, T_i) = \prod_{l=n}^{i-1} \frac{1}{1 + \delta L_l(T_n)}, \tag{67}$$

and the swap rate reads

$$S_n(T_n) = \frac{1 - B(T_n, T_{N+1})}{\delta \sum_{l=n+1}^{N+1} B(T_n, T_l)}. \tag{68}$$

Here we consider European style payouts. The extension to Bermuda options of Leclerc and collaborators (Leclerc *et al.*, 2009) can be obtained with a simple modification of the original algorithm.

The remarkable computational efficiency of the implementation discussed above is illustrated in Fig. 9. Here we plot the execution time for the calculation of all the Delta, $\partial V/\partial L_i(0)$, and Vega, $\partial V/\partial \sigma_i(n)$, relative to the calculation of the swaption value as obtained with the implementation above and by finite differences. As the maturity of the swaption increases, the number of risk to compute also increases.

In practical terms, results as shown in Fig. 9 mean that AAD produces orders of magnitude speed ups with respect to bumping. As a result, a calculation that would ordinarily take many hours can be performed, *on the same hardware*, in a matter of a few minutes. By the same token, calculations for which a large amount of computer power is generally deployed in order to achieve an acceptable performance, can be executed with a fraction of the computational resources and energy consumed (Andreasen, 2023). This significantly reduces the costs and the environmental impact of financial institutions.

2. Correlation Risk and Binning

The complication with the implementation discussed in Sec.II.C is that it gives an estimate for the correlation risk, but it does not provide a corresponding confidence interval. An alternative approach would be to convert $\bar{C}$ to $\bar{\rho}$ for each individual path, and then compute the average and standard deviation of $\bar{\rho}$ in the usual way. However, this is in general rather costly (Capriotti and Giles, 2010). An excellent compromise between these two extremes is to divide the N_{MC} paths into N_b 'bins' of equal size. For each bin, an average value of $\bar{C}$ is computed and converted into a corresponding value for $\bar{\rho}$. These N_b estimates for $\bar{\rho}$ can then be combined in the usual way to form an overall estimate and confidence interval for the correlation risk.

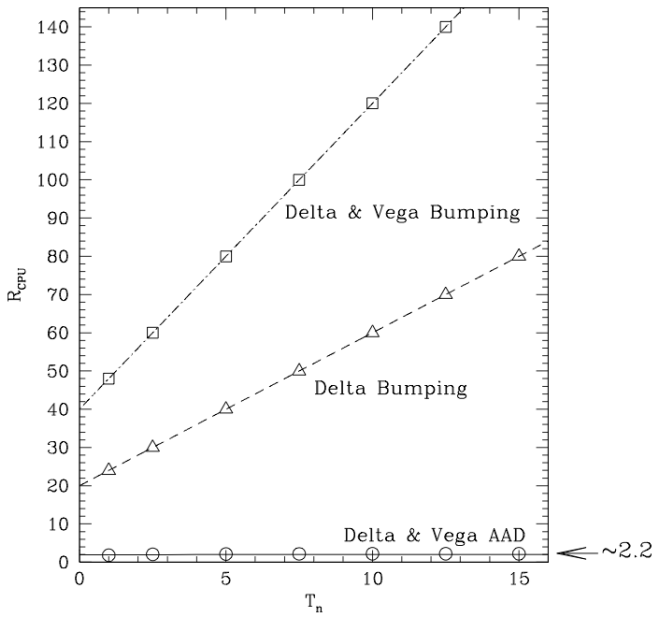

FIG. 9 Ratio of the CPU time required for the AAD calculation of the Delta and Vega and the time to calculate the option value for the swaption in Eq. (66) as a function of the option expiry T_n. The time to calculate Delta and Vega using bumping is also shown. Lines are guides for the eye.

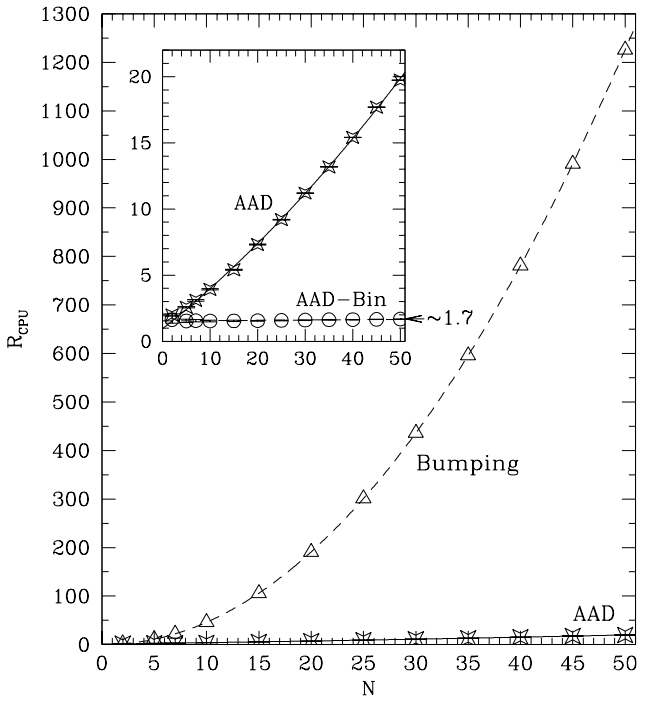

FIG. 10 Ratios of the CPU time required for the calculation of the option value, and correlation Greeks, and the CPU time spent for the computation of the value alone, as functions of the number of names in the basket. Symbols: Bumping (one-sided finite differences) (triangles), AAD without binning (i.e. $N_b = N_{MC}$) (stars), AAD with binning ($N_b = 20$) (empty circles). Lines are guides for the eye, and the MC uncertainties are smaller than the symbol sizes.

The computational benefits can be understood by considering the computational costs for both the standard evaluation and the adjoint pathwise derivative calculation. In the standard evaluation, the cost of the Cholesky factorization is $O(N^3)$, and the cost of the MC sampling is $O(N_{MC}N^2)$, so the total cost is $O(N^3 + N_{MC}N^2)$. Since N_{MC} is always much greater than N, the cost of the Cholesky factorization is usually negligible. The cost of the adjoint steps in the MC sampling is also $O(N_{MC}N^2)$, and when using N_b bins the cost of the adjoint Cholesky factorization is $O(N_bN^3)$. To obtain an accurate confidence interval, but with the cost of the Cholesky factorisation being negligible, requires that N_b is chosen so that $1 \ll N_b \ll N_{MC}/N$. Without binning, i.e., using $N_b = N_{MC}$, the cost to calculate the average of the estimators (59) is $O(N_{MC}N^3)$, and so the relative cost compared to the evaluation of the option value is $O(N)$.

The remarkable computational efficiency of AAD is illustrated in Fig. 10 for the Second to Default Swap. Here we plot the ratio of the CPU time required for the calculation of the value of the option, and all its pairwise correlation sensitivities, and the CPU time spent for the computation of the value alone, as functions of the number of names in the basket. As expected, for standard finite-difference estimators, such ratio increases quadratically with the number of names in the basket. Without the use of binning, i.e., if we perform the adjoint of the Cholesky factorization on each MC path, the relative cost of the AAD calculation scales linearly with N. With binning the relative cost of AAD is constant (see inset of Fig. 10). Already for medium sized basket ($N \simeq 20$) the cost associated with bumping is over 100 times more expensive than the one of AAD (Capriotti and Giles, 2010).

The binning procedure described above can be generalized to any situation in which the standard solution procedure involves a common preprocessing step before any of the path calculations are performed. Other examples would include calibration of model parameters to market prices, or a cubic spline construction of a local volatility surface. In each case, there is a linear relationship between the forward mode sensitivities before and after the preprocessing step, and therefore a linear relationship between the corresponding adjoint sensitivities.

E. Discontinuous Payoffs

The biggest limitation of the pathwise derivative method (both in its naïve and in its adjoint implementation) is that it cannot handle discontinuous payoffs because of the previously mentioned regularity conditions required to take the derivative inside the expectation as in Eq. (37). This requirement is generally cited in the literature as a shortcoming of the pathwise derivative method. Indeed, it potentially limits the practical utility of the method to a great extent as the majority of the

payout functions commonly used for structured derivatives contains discontinuities, e.g., in the form of digital features, random variables counting discrete events, or barriers (Hull, 2002). A similar limitation appears when trying to apply the pathwise derivative method to compute second order risk of piecewise differentiable payoffs (such those typically used in practice).

Fortunately, the Lipschitz requirement turns out to be more of a theoretical than a practical limitation. Indeed, a practical way of addressing non-Lipschitz payouts is to smooth out explicitly the payoff at hand replacing the Heaviside functions it contains with a smooth alternative (Capriotti, 2011). This comes at the cost of introducing a finite bias in the sensitivity estimates. However, such bias can generally be reduced to levels that are considered acceptable in financial practice.

In some circumstances it is reasonably straightforward to perform some analytical work and integrate exactly the singularities introduced by differentiating under expectation Heaviside functions. Indeed, in a distributional sense, differentiating an Heaviside function gives rise to a Dirac's delta which is easy to integrate. Examples of such an approach are (Joshi and Kainth, 2004) and (Capriotti *et al.*, 2011). This approach has the additional benefit of removing some, often a significant part, of the variance of the payoff estimator thus reducing the number of Monte Carlo simulations required to achieve the same level of accuracy (Capriotti *et al.*, 2011). However, this analytical work, being payout dependent, is often too onerous to be of practical utility in a practitioner's context.

A third type of approach is using conditional expectations to smooth out the payoff (Glasserman, 2004). This can easily be illustrated for digital options, where one can stop the path simulation one time step before maturity $T_M = t_{N_s}$. Conditional on the value $X(t_{N_s-1})$, an Euler discretization for the final time step has the form

$$\begin{aligned} X(t_{N_s}) = X(t_{N_s-1}) &+ \mu\left(X(t_{N_s-1}), t_{N_s-1}, \theta\right) h_{N_s-1} \\ &+ \sigma\left(X(t_{N_s-1}), t_{N_s-1}, \theta\right) \Delta W_{t_{N_s-1}} \end{aligned} \tag{69}$$

where $\Delta W_{t_{N_s-1}} = W_{t_{N_s}} - W_{t_{N_s-1}}$, which defines a (multivariate) distribution for $X(t_{N_s})$. A path simulation can be performed in the usual way for the first $N_s - 1$ time step, while on the final time step, one instead considers the full distribution of possible values for $\Delta W_{t_{N_s-1}}$. This gives a Gaussian distribution for $X(t_{N_s})$, which, in the scalar case, reads

$$\begin{aligned} &p_\theta\left(X(t_{N_s}) \mid X(t_{N_s-1})\right) \\ &= \frac{1}{\sqrt{2\pi\sigma_W^2}} \exp\left(-\frac{\left(X(t_{N_s}) - \mu_W\right)^2}{2\sigma_W^2}\right) \end{aligned} \tag{70}$$

where

$$\begin{aligned} \mu_W(X(t_{N_s-1})) &= X(t_{N_s-1}) + \mu\left(X(t_{N_s-1}), t_{N_s-1}, \theta\right) h_{N_s-1}, \\ \sigma_W(X(t_{N_s-1})) &= \sigma\left(X(t_{N_s-1}), t_{N_s-1}, \theta\right) \sqrt{h_{N_s-1}} \ . \end{aligned} \tag{71}$$

Hence, the conditional expectation for the value of a digital payoff with strike K,

$$P(X(T_M)) = \mathbf{1}(X(T_M) - K) \equiv \begin{cases} 1, & X(T_M) > K \\ 0, & X(T_M) \leq K \end{cases} ,$$

where $\mathbf{1}()$ is the Heaviside theta function, reads

$$\begin{aligned} &\mathbb{E}\left[P\left(X(t(N_s)) \mid X(t_{N_s-1})\right]\right. \\ &= \int_{-\infty}^{\infty} \mathbf{1}\left(z - K\right) p_\theta\left(z \mid X(t_{N_s-1})\right) \mathrm{d} = \Phi\left(\frac{\mu_W - K}{\sigma_W}\right) \end{aligned} \tag{72}$$

where $\Phi(\cdot)$ is the cumulative Normal distribution function. The MC estimator for the option value is now

$$\widehat{V} = \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} \mathbb{E}\left[P\left(X(t_{N_s}) \mid X(t_{N_s-1}^{(i_{\mathrm{MC}})})\right)\right] \ .$$

and because the conditional expectation $\mathbb{E}\left[P\left(X(t_{N_s}) \mid X(t_{N_s-1})\right]\right.$ is a differentiable function of the input parameters (provided the drift and volatility are regular enough (Glasserman, 2004)) the pathwise sensitivity approach can now be used.

There are two difficulties in using this form of conditional expectation in real-world applications. The first is that the integral arising from the conditional expectation will often become a multi-dimensional integral without an obvious closed-form value (e.g., consider a digital option based on the median of a basket of 20 stocks), and the second is that it again requires payoff specific changes to the MC valuation infrastructure.

One solution, dubbed 'vibrato' in (Giles, 2009), is to use a Monte Carlo estimate of the conditional expectation, and use the so-called Likelihood Ratio Method (LRM) (Glasserman, 2004) to compute its sensitivities. Going back to the simple example above one can write

$$\begin{aligned} &\frac{\partial}{\partial\theta}\mathbb{E}\left[P\left(X(t_{N_s})\right) \mid X(t_{N_s-1})\right] \\ &= \mathbb{E}\left[P\left(X(t_{N_s})\right)\left(\frac{\partial \log p_\theta}{\partial\theta}\right)_{\mathrm{total}} \mid X(t_{N_s-1})\right] \end{aligned} \tag{73}$$

with

$$\begin{aligned} &\left(\frac{\partial \log p_\theta}{\partial\theta}\right)_{\mathrm{total}} \\ &= \frac{\partial \log p_\theta}{\partial X(t_{N_s-1})}\frac{\partial X(t_{N_s-1})}{\partial\theta} + \frac{\partial \log p_\theta}{\partial\theta} \ . \end{aligned} \tag{74}$$

Re-arranging we get

$$\begin{aligned} &\frac{\partial}{\partial\theta}\mathbb{E}\left[P\left(X(t_{N_s})\right) \mid X(t_{N_s-1}\right] \\ &= \mathbb{E}\left[P\left(X(t_{N_s})\right)\frac{\partial \log p_s}{\partial X(t_{N_s-1})} \mid X(t_{N_s-1})\right]\frac{\partial X(t_{N_s-1})}{\partial\theta} \\ &+ \mathbb{E}\left[P\left(X(t_{N_s})\right)\frac{\partial \log p_\theta}{\partial\theta} \mid X(t_{N_s-1})\right] \ . \end{aligned} \tag{75}$$

As as result, in the backward sweep, one can initialize the adjoint as

$$\bar{X}(t_{N_s-1}) = \mathbb{E}\left[P\left(X(t_{N_s})\right)\frac{\partial \log p_s}{\partial X(t_{N_s-1})} \mid X(t_{N_s-1})\right]$$
$$\bar{\theta} = \mathbb{E}\left[P\left(X(t_{N_s})\right)\frac{\partial \log p_\theta}{\partial \theta} \mid X(t_{N_s-1})\right] \quad (76)$$

and then continue backwards down the path in the usual way (Giles, 2009).

F. Regression based Monte Carlo

An important extension of the AAD scheme presented above is the one for derivatives with early-exercise feature (Glasserman, 2004). Capriotti *et al.*, 2017 presented the AAD version of the celebrated least-square algorithms of (Tsitsiklis and Roy, 2001) and (Longstaff and Schwartz, 2001) and, by discussing in detail examples of practical relevance, demonstrated how accounting for the contributions associated with the regression functions is crucial to obtain accurate estimates of the Greeks, especially in XVA applications (Silotto *et al.*, 2023). Similarly, for Bermudan swaptions, Antonov, 2016 demonstrated the significant computational advantages of the approach with respect to brute force sensitivity calculations. Here we briefly present these extensions following (Capriotti *et al.*, 2017).

The value $V(t)$ of a Bermudan option is the supremum of the option value over all possible exercise policies:

$$\frac{V(t)}{N(t)} = \sup_{\tau \in \mathcal{T}(t)} \mathbb{E}_t\left[\frac{E(\tau)}{N(\tau)}\right] \quad (77)$$

where $E(t)$ is the exercise value of the option, and $N(t)$ is the value of the numeraire asset at time t. We denote by $T_1, \ldots, T_M$ the exercise dates of the option, $\mathcal{D}(t) = \{T_m \geq t\}$ and by $\eta(t)$ the smallest integer such that $T_{\eta(t)+1} > t$. An exercise policy is represented mathematically by a stopping time taking values in $\mathcal{D}(t)$. $\mathcal{T}(t)$ is the set of stopping times taking values in $\mathcal{D}(t)$.

Given then hold value of the Bermudan-style option,

$$\frac{H(t)}{N(t)} = \mathbb{E}_t\left[\frac{V\left(T_{\eta(t)+1}\right)}{N\left(T_{\eta(t)+1}\right)}\right], \quad (78)$$

the option holder, following an optimal exercise policy, will exercise his option if the exercise value is larger than the hold value, i.e.,

$$V\left(T_{\eta(t)}\right) = \max\left(E\left(T_{\eta(t)}\right), H\left(T_{\eta(t)}\right)\right). \quad (79)$$

This, leads to the so-called dynamic programming formulation (Glasserman, 2004),

$$\frac{H(t)}{N(t)} = \mathbb{E}_t\left[\max\left(\frac{E\left(T_{\eta(t)+1}\right)}{N\left(T_{\eta(t)+1}\right)}, \frac{H\left(T_{\eta(t)+1}\right)}{N\left(T_{\eta(t)+1}\right)}\right)\right], \quad (80)$$

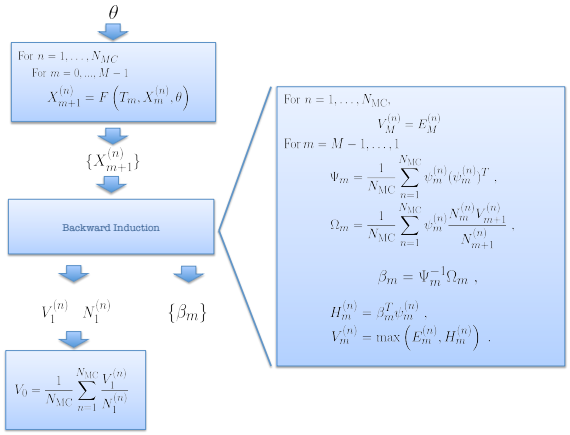

FIG. 11 Schematic representation of the a regression based MC algorithm.

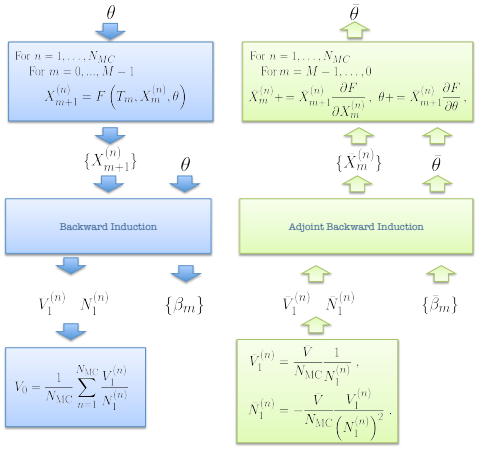

FIG. 12 Schematic representation of the a regression based MC algorithm and of its adjoint.

for $T_\eta \leq t < T_{\eta+1}$, and $\eta = 1, \ldots, M-1$. Starting from the terminal condition $H\left(T_M\right) \equiv 0$, the equation above defines a backward iteration in time for $H(t)$.

If the underlying risk factor process $\{X(t)\}_{0 \leq t \leq T}$ is a generic k-dimensional Markov process, the hold value $H(t)$ is a function of the state vector at time t, namely,

$$H_t(x) = \mathbb{E}\left[\frac{N(X(t))}{N\left(X\left(T_{m+1}\right)\right)} V\left(X\left(T_{m+1}\right)\right) \mid X(t) = x\right]. \quad (81)$$

Regression-based MC (Longstaff and Schwartz, 2001; Tsitsiklis and Roy, 2001) techniques provide an effective way of computing conditional expectation values of the form above.

In the context of the valuation of Bermudan-style op-

tions, the hold value on an exercise date T_m

$$H_{T_m}(x) = \mathbb{E}\left[\frac{N\left(X\left(T_m\right)\right)}{N\left(X\left(T_{m+1}\right)\right)} V\left(X\left(T_{m+1}\right)\right) \mid X\left(T_m\right) = x\right]. \tag{82}$$

is assumed to be of the form

$$\hat{H}_m(x) = \beta_m^t \psi(x) \tag{83}$$

where $\psi(x) = \left(\psi_1(x), \ldots, \psi_d(x)\right)^t$ is a vector of d basis functions. The vector of coefficients $\beta_m = \left(\beta_{1m}, \ldots, \beta_{dm}\right)^t$ can be determined by regressing

$$N\left(X\left(T_m\right)\right) / N\left(X\left(T_{m+1}\right)\right) V\left(X\left(T_{m+1}\right)\right)$$

versus

$$\psi\left(X\left(T_{m+1}\right)\right) \ .$$

This gives

$$\beta_m = \Psi_m^{-1} \Omega_m, \tag{84}$$

where we define the $d \times d$ matrix

$$\Psi_m = \mathbb{E}\left[\psi\left(X\left(T_m\right)\right) \psi^t\left(X\left(T_m\right)\right)\right] \tag{85}$$

and the $d \times 1$ vector

$$\Omega_m = \mathbb{E}\left[\frac{N\left(X\left(T_m\right)\right) V\left(X\left(T_{m+1}\right)\right)}{N\left(X\left(T_{m+1}\right)\right)} \psi\left(X\left(T_m\right)\right)\right] \ . \tag{86}$$

These equations provide a straightforward recipe to compute the regression coefficients β_m by substituting Ψ_m and Ω_m with their sample average over N_{MC} replications. The schematic illustration of the algorithm is displayed in Fig. 11. Using the rules illustrated in Sec. I.D, it is straightforward to design the AAD version of the algorithm as in Fig. 12 with the implementation of the adjoint version of the backward induction depicted in stylized form in Fig. 13, see e.g., (Capriotti *et al.*, 2017).

The hold value obtained by regression defines an exercise policy whereby on each exercise date T_m the option is exercised if

$$E\left(X\left(T_m\right)\right) > H_m\left(X\left(T_m\right)\right) \equiv \beta_m^t \psi\left(X\left(T_m\right)\right). \tag{87}$$

Such policy, corresponding in general to a suboptimal stopping time, will result in a lower-bound estimator for the Bermudan-style option value (Glasserman, 2004) which is often what is used in the financial practice. The resulting algorithm consists of a pre-simulation in which the regression coefficients are computed and an *independent* simulation evaluating the estimator

$$P^{(n)} = \sum_{m=1}^{M}\left[\mathbf{1}^{(n)}\left(t_1, t_m\right) \mathbf{1}\left(E_m^{(n)} > H_m^{(n)}\right) \frac{E_m^{(n)}}{N_m^{(n)}}\right] \tag{88}$$

where

$$\mathbf{1}^{(n)}\left(t_1, t_m\right) = \left(\prod_{i=1}^{m-1} \mathbf{1}\left(H_i^{(n)} > E_i^{(n)}\right)\right), \tag{89}$$

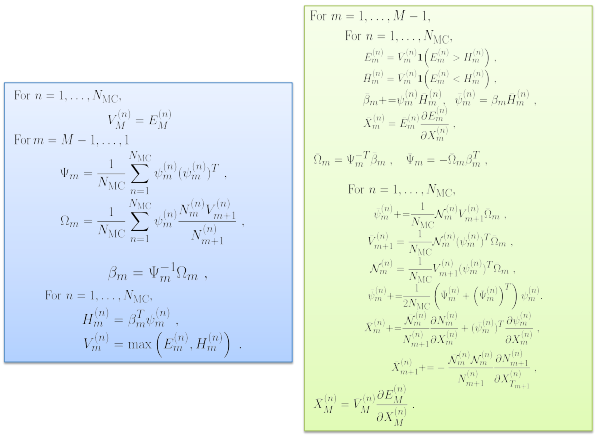

FIG. 13 Schematic representation of the adjoint of the backward induction algorithm.

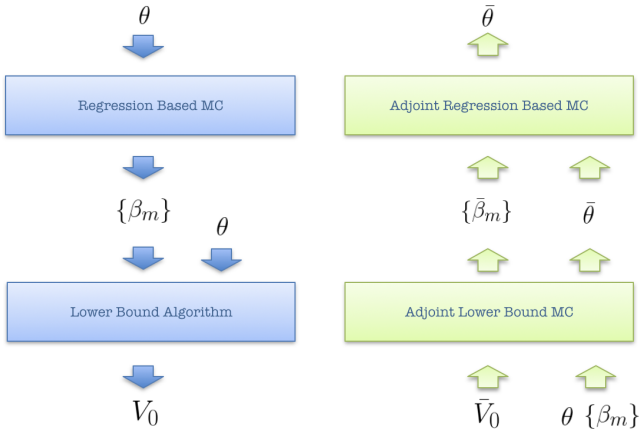

FIG. 14 Schematic representation of the lower bound algorithm and of its adjoint.

and the convention $\mathbf{1}^{(n)}\left(t_1, t_1\right) = 1$. This is illustrated in Fig. 14.

The payoff estimator for Bermudan-style options (88) is not differentiable with respect to the pathwise value of the approximate exercise boundary $H_m^{(n)}$, and it requires the regularization described in Sec. II.E. A common approximation is to assume that that regression exercise boundary is close to optimality so that the value of the contract is approximatively continuous across the exercise boundary and no regularization is required. Under this assumption, no contribution to the sensitivities is associated with the perturbations of the exercise boundary, and one can therefore keep the regression coefficients fixed while calculating the sensitivities. The accuracy of this approximation depends on the quality of the exercise boundary. In general though, a payoff regularization needs to be applied and the contribution of the sensitivities arising from the regression coefficients needs to be kept into account (Capriotti *et al.*, 2017). This is partic-

ularly important when regression based approaches are applied for XVA applications where omitting this step could lead to significant mis-hedging, see e.g. (Capriotti *et al.*, 2017).

G. Other Monte Carlo Applications

On a different line of research, Jain *et al.*, 2019 extended the stochastic grid bundling method (SGBM), a "regress-later" (Glasserman and Yu, 2004) based Monte Carlo scheme for pricing early-exercise options (Jain and Oosterlee, 2015), with an adjoint method to compute in an efficient manner sensitivities along the paths with an acceptable accuracy. They also proposed the application of the method for the efficient implementation of the ISDA standard initial margin model, an application also discussed in (Antonov *et al.*, 2017).

One of the most successful applications of AAD is in the context of XVA, as originally proposed in (Capriotti *et al.*, 2011), due the computational complexity associated with such portfolio risk measures. The calculation of risk on XVA generally involves the simultaneous evaluation of conditional values of large 'netting sets' of trades referencing in turn a large number of market risk factors. As a result, it requires computationally intensive MC simulations. For typical applications, AAD allows a reduction of the computational cost by hundreds of times allowing one to perform in minutes risk runs that would take otherwise several hours or could not even be performed overnight without large parallel computers. Similar results, were also obtained in (Huge and Savine, 2017). Here the authors considered a reformulation of the XVA problem in terms of a least square regression approach to which AAD is applied, similarly to (Capriotti *et al.*, 2017). Similar applications were also considered by (Fries, 2019).

AAD can also be used to obtain the Hessian of computer implemented functions (Griewank and Walther, 2008; Naumann, 2011). For a scalar function, the cost for computing the Hessian by AAD, in units of the cost to evaluate the original function, can be shown to be bounded by a linear function in the number of inputs (Griewank and Walther, 2008; Naumann, 2011). For the calculation of the full Hessian, this represents a saving with respect to standard bumping of order number of inputs. However, such computational cost has the same dependence on the number of inputs of computing gradients by AAD and applying the tangent mode of AD to the calculation of the gradient. Forming finite difference estimators of the Hessian using AAD to compute the first order derivatives would also share the same cost. The appeal of the latter approach is the simplicity of the implementation and the fact that it does not require regularity (Lipschitz continuity) of the gradient of the function.

Joshi and Yang, 2011 pursued an algorithmic adjoint approach for the calculation of second order Greeks in the context of the Libor Market Model and they showed

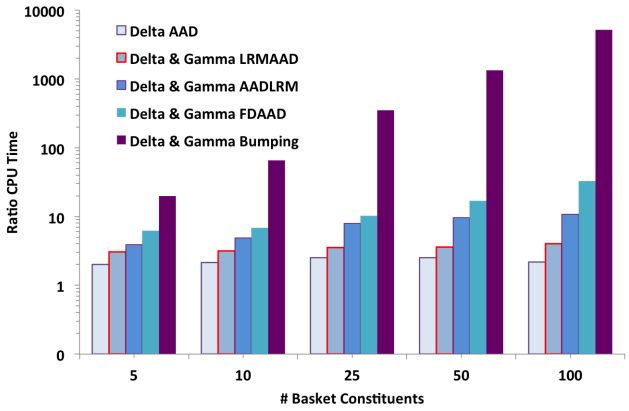

FIG. 15 Cost of computing Delta and Gamma relative to the cost of computing the value for a Basket option as a function of the number of assets in the basket.

that, as expected on general grounds, the complexity of the Hessian calculation is a linear function of the number of state variables times the complexity of the original algorithm. Joshi and Yang, 2010 also applied a similar approach to the calculation second order of portfolio credit derivatives such as synthetic collateralized debt obligation (CDO) tranches, while Chan *et al.*, 2015 applied it to different simulation schemes in the Heston model.

Other approaches for the calculation of second order risk can be obtained by combining different methods for the calculation of first order risk. For example, Capriotti, 2015 showed how AAD can be combined with the pathwise derivative and likelihood ratio method (LRM) (Glasserman, 2004) to construct efficient Monte Carlo estimators of second order price sensitivities of derivative portfolios. This can be done by applying AAD to the LRM estimators (AADLRM) or, viceversa, applying LRM to the AAD estimators (LRMAAD). The paper demonstrated with a numerical example how the proposed technique can be straightforwardly implemented to greatly reduce the computation time of second order risk. As illustrated in Fig. 15 the mixed AADLRM and LRMAAD estimators are far more efficient than using finite-differences and significatively more efficient than the mixed finite-difference AAD (FDAAD) approach. All the mixed estimators result in over one order of magnitude savings in computation time even for medium sized basket $N \sim 10$. As expected, while the pure finite-difference estimators display a computational complexity scaling as $O(N^2)$, both the AADLRM and the FDAAD approaches scale as $O(N)$. However, the dependence on N for the AADLRM approach is much weaker because only a part of the estimator scales linearly with the number of assets. This is generally true also for the LRMAAD estimator. However, for this simple problem the LRMAAD estimator can be computed more efficiently. This results in the remarkable outcome that the full Gamma matrix can be computed at a cost that is approximately four times the cost of computing the value of the option

irrespective of the number of underlying assets in the basket.

Similarly Pironneau *et al.*, 2016 combined two methods, Vibrato (Giles, 2009) and AAD. They showed that this combined technique is faster than standard finite difference, more stable than automatic differentiation of second order derivatives and more general than Malliavin Calculus (Fournié *et al.*, 1999).

III. PDE APPLICATIONS

Although we limited the discussion here to the pathwise derivative method and MC simulations, the ideas presented here are applicable in general to implement the calculation of sensitivities for any numerical algorithm, including PDEs for which the technique was first introduced in computational finance by Achdou and Pironneau, 2005. In this section we will focus on these.

A. Option prices and backward PDEs

Option pricing problems can be often formulated in terms of a linear parabolic PDE of second order of the form

$$\frac{\partial V}{\partial t} + \mu(x,t,\theta)\,\frac{\partial V}{\partial x} + \frac{1}{2}\sigma^2(x,t,\theta)\,\frac{\partial^2 V}{\partial x^2} - \nu(x,t,\theta)\,V = 0, \tag{90}$$

where

$$V_t(\theta) = V(x_t,t,\theta) \equiv \mathbb{E}\left[\exp\left(-\int_t^t \nu(x_u,u,\theta)\mathrm{d}u\right) P(x_T,\theta)\,\middle|\, x_t\right] \tag{91}$$

is the value of a derivative contract at time t; see, e.g., (Andersen and Piterbarg, 2010). The expectation is taken under a suitable probability measure, depending on the financial context, given the value of a state variable x_t at time $t \geq 0$. At the maturity date $T > t$, the value $P_\theta(x_T)$ of the financial derivative depends on the realization of the risk factor $\{x_t\}_{0\leq t}$ that satisfies

$$\mathrm{d}x_t = \mu(x_t,t,\theta)\mathrm{d}t + \sigma(x_t,t,\theta)\mathrm{d}W_t \tag{92}$$

where $\mu(x,t,\theta)$ and $\sigma(x,t,\theta)$ are the drift and the volatility functions, and $\{W_t\}_{0\leq t}$ is a one-dimensional Brownian motion. As before, $\theta = (\theta_1,\ldots,\theta_{N_\theta})$ represents the vector of N_θ parameters the model is dependent on. By supplying appropriate spatial boundary conditions, see, e.g., (Andersen and Piterbarg, 2010), and the terminal condition $V(x,T,\theta) = P_\theta(x)$ at maturity T, Eq. (90) can be solved backwards in time for the value $V(x,t,\theta)$ of the derivative security at any time $t \leq T$.

The Black-Scholes PDE for the price of European-style claims (Hull, 2002) is of the form (90) where $\mu(x,t,\theta) = (r(t)-\delta(t))\,x$, $\sigma(x,t,\theta) = \sigma(t)x$ and $\nu(x,t,\theta) = r(t)$. Here $r(t)$ and $\delta(t)$ denote the (deterministic) risk-free interest rate and dividend yield, respectively.

B. Numerical solutions by finite-difference discretization for backward PDEs

The solution $V_{t_0}(\theta) = V(x_{t_0},t_0,\theta)$ of the PDE (90) can be found numerically by discretization on the rectangular domain $(t,x) \in [t_0,T]\times[x_{\min},x_{\max}]$ where $x_{\min}$ and $x_{\max}$ (such that $x_{\min} < x_{t_0} < x_{\max}$) are constants obtained by means of probabilistic considerations, see (Andersen and Piterbarg, 2010). In particular, by denoting *i)* the points on the time axis by $t_m = t_0 + m\Delta t$ where $m = 0,\ldots,M$ and $\Delta t = (T-t_0)/M$, and *ii)* the points on the spatial axis by $x_j = x_{\min} + j\Delta x$, where $j = 0,\ldots,N+1$ and $\Delta x = (x_{\max}-x_{\min})/(N+1)$, one can discretise the PDE (90) with finite-difference approximations for the first and second derivatives. A standard discretization scheme, see, e.g., (Andersen and Piterbarg, 2010), results in a matrix iteration of the form[3]

$$L_B(t_m,\phi,\theta)V^m(\theta) = R_B(t_m,\phi,\theta)V^{m+1}(\theta) \tag{93}$$

where $V^m(\theta) = (V(x_1,t_m,\theta),\ldots,V(x_N,t_m,\theta))^t$ and $V(t_m,x_j,\theta)$ indicate the finite-difference approximation to the solution of the PDE (90)[4]. We introduce the $N\times N$ tri-diagonal matrices

$$L_B(t_m,\phi,\theta) = \mathbb{I} - \phi\Delta t D(\widetilde{t}_m(\phi),\theta), \tag{94}$$

$$R_B(t_m,\phi,\theta) = \mathbb{I} + (1-\phi)\Delta t D(\widetilde{t}_m(\phi),\theta), \tag{95}$$

where $\widetilde{t}_m(\phi) = (1-\phi)t_m + \phi\, t_m$. Both expressions are defined in terms of the tri-diagonal matrix $D(t,\theta)$ given by

$$\begin{aligned}
&[D(t,\theta)]_{j,j} = c_j(t,\theta),\\
&[D(t,\theta)]_{j,j+1} = u_j(t,\theta),\\
&[D(t,\theta)]_{j+1,j} = l_{j+1}(t,\theta),
\end{aligned} \tag{96}$$

where $j = 1,\ldots,N$ in the first equation and $j = 1,\ldots,N-1$ for the second and third and the coefficients $c_j(t,\theta)$, $u_j(t,\theta)$ and $l_j(t,\theta)$ are defined in terms of the functions $\mu(x,t,\theta)$, $\sigma(x,t,\theta)$ and $\nu(x_j,t,\theta)$ in the PDE (90) (Capriotti *et al.*, 2015).

The parameter ϕ is bounded between $\phi = 0$, corresponding to the fully explicit scheme, and $\phi = 1$, corresponding to the fully implicit scheme (Andersen and Piterbarg, 2010).

Given the value of the derivative at maturity $V_j^M(\theta) = P_\theta(x_j)$, Equation (93) can be recursively solved, by utilizing standard tri-diagonal solvers for $m = M-1,\ldots,0$, in order to find the vector $V_j^0(\theta)$. From this, the value of the derivative $V_{t_0} = V(x_{t_0},t_0,\theta)$, corresponding to the

[3] Here for simplicity of exposition we omit the boundary term, see, e.g., (Capriotti *et al.*, 2015)

[4] To keep the notation as light as possible, we denote the exact solution of the PDE (90) and its finite-difference approximation with the same symbol.

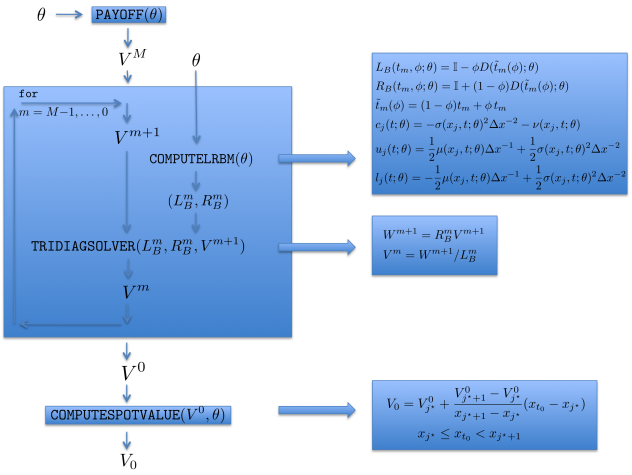

FIG. 16 Schematic illustration of an algorithm solving the PDE (90) by finite differences.

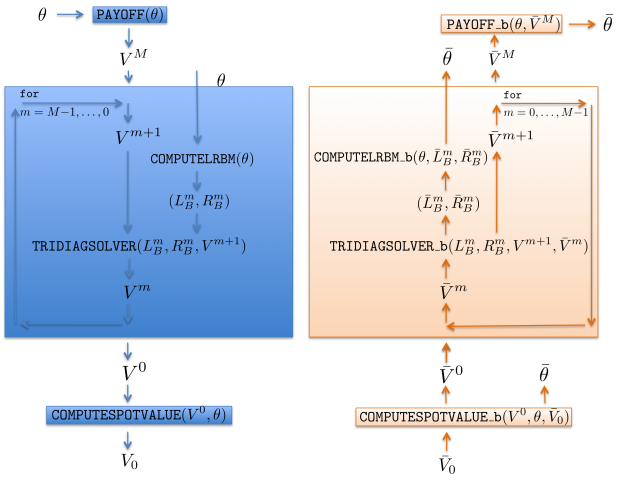

FIG. 17 Schematic illustration of an adjoint (right) of the algorithm for solving the PDE (left) in Eq. (90) by finite differences.

state variable x_{t_0} observed at time t_0, can be computed by means of, e.g., linear interpolation,

$$V_{t_0} = V^0_{j^\star} + \frac{V^0_{j^\star+1} - V^0_{j^\star}}{x_{j^\star+1} - x_{j^\star}}(x_{t_0} - x_{j^\star}), \tag{97}$$

with $j^\star$ such that $x_{j^\star} \le x_{t_0} < x_{j^\star+1}$. The associated algorithm (see Fig. 16) is given as follows:

(S1) Initialize the value vector on the final time slice $V^M_j(\theta) = P_\theta(x_j)$ with $j = 0, \dots, N$:

$$V^M = \texttt{PAYOFF}(\theta). \tag{98}$$

Note that, here and in the following we omit the dummy dependence of the pseudocode functions on the grid points as well as on the parameter ϕ as we want to focus on the sensitivities of the model parameters θ.

(S2) For $m = M-1, \dots, 0$ execute the following steps:

a) Compute the coefficient vectors $c^m(\theta) \equiv c(\widetilde{t}_m(\phi), \theta)$, $u^m(\theta) \equiv u(\widetilde{t}_m(\phi), \theta)$, and $l^m(\theta) \equiv l(\widetilde{t}_m(\phi), \theta)$:

$$(c^m, u^m, l^m) = \texttt{COMPUTECOEFFM}(\theta). \tag{99}$$

b) Compute the matrices $L^m_B(\theta) \equiv L_B(t_m, \phi, \theta)$ and $R^m_B(\theta) \equiv R_B(t_m, \phi, \theta)$ in Equations (94) and (95) from the coefficients vectors $c^m(\theta)$, $u^m(\theta)$, and $l^m(\theta)$:

$$(L^m_B, R^m_B) = \texttt{COMPUTELRB}(c^m, u^m, l^m). \tag{100}$$

c) Given V^{m+1}, solve Equation (93) for V^m by calling a suitable tri-diagonal solver such as

$$V^m = \texttt{TRIDIAGSOLVER}(L^m_B, R^m_B, V^{m+1}), \tag{101}$$

which we can represent mathematically as the following sequence of operations:

$$\begin{aligned} W^{m+1} &= R^m_B V^{m+1}, \\ V^m &= W^{m+1}/L^m_B, \end{aligned} \tag{102}$$

where we adopted the notation "B/A" to represent finding the solution X of the linear system $AX = B$.

(S3) Compute $V_{t_0} = V(x_{t_0}, t_0, \theta)$ with a suitable interpolation scheme, e.g., the scheme of (97), by calling a method of the kind

$$V_{t_0} = \texttt{COMPUTESPOTVALUE}(V^0)\ . \tag{103}$$

Since the matrix (96) is tri-diagonal, the cost of a single iteration of Equation (93) is $O(N)$. As a result, the overall computation complexity of the algorithm above is $O(NM)$.

Incorporating intermediate cash flows, or Bermudan optionality is also straighforward but we omit the description for simplicity referring the interested reader to (Capriotti *et al.*, 2015).

C. AAD and backward PDEs

The evaluation of the numerical solution of the PDE (90) by means of the algorithm described in Section III.B can be seen as a computer-implemented function mapping $\theta \to V_{t_0}(\theta)$. By following the principles of AAD, it is possible to design its adjoint counterpart $(\theta, \bar{V}_{t_0}) \to (V_{t_0}, \bar{\theta})$ which gives (for $\bar{V}_{t_0} = 1$) the sensitivities

$$\bar{\theta}_k = \frac{\partial V_{t_0}(\theta)}{\partial \theta_k}, \tag{104}$$

for $k = 1, \ldots, N_\theta$. The adjoint of the solution of the backward PDE in Section III.B consists therefore of Steps 1-3 followed by their corresponding adjoint, executed in reverse order, as illustrated in Fig. 17:

($\bar{\text{S3}}$) Set $\bar{V}_{t_0} = 1$, and execute

$$\bar{V}^0 = \texttt{COMPUTESPOTVALUE\_b}(V^0, \bar{V}_{t_0})$$

to compute

$$\bar{V}^0_j = \bar{V}_{t_0} \frac{\partial V_{t_0}}{\partial V^0_j}$$

for $j = 1, \ldots, N$, according to rule (97) .

($\bar{\text{S2}}$) For $m = 0, \ldots, M-1$, in opposite order than in (S2) of Section III.B, execute

$\bar{c}$) Given $\bar{V}^m$, execute the adjoint of function (101), namely

$$(\bar{L}^m_B, \bar{R}^m_B, \bar{V}^{m+1}) = \texttt{TRIDIAGSOLVER\_b}(L^m_B, R^m_B, V^{m+1}, \bar{V}^m) \ , \qquad (105)$$

which computes

$$[\bar{L}^m_B]_{j,l} = \sum_{r=1}^N \bar{V}^m_r \frac{\partial V^m_r}{\partial [L^m_B]_{j,l}}, \quad [\bar{R}^m_B]_{j,l} = \sum_{r=1}^N \bar{V}^m_r \frac{\partial V^m_r}{\partial [R^m_B]_{j,l}},$$

$$\bar{\beta}^{m+1}_j = \sum_{r=1}^N \bar{V}^m_r \frac{\partial V^m_r}{\partial \beta^{m+1}_j}, \qquad \bar{V}^{m+1}_j = \sum_{r=1}^N \bar{V}^m_r \frac{\partial V^m_r}{\partial V^{m+1}_j},$$

for $j = 1, \ldots, N$ and $l = 1, \ldots, N$.

$\bar{b}$) Compute the adjoint of the function (100), that is

$$(\bar{c}^m, \bar{u}^m, \bar{l}^m) = \texttt{COMPUTELRB\_b}(c^m, u^m, l^m, \bar{L}^m_B, \bar{R}^m_B),$$

which produces the adjoint of the coefficient vectors

$$\bar{c}^m_j = [\bar{L}^m_B]_{j,j} \frac{\partial [L^m_B]_{j,j}}{\partial c^m_j} + [\bar{R}^m_B]_{j,j} \frac{\partial [R^m_B]_{j,j}}{\partial c^m_j},$$

$$\bar{u}^m_j = [\bar{L}^m_B]_{j,j+1} \frac{\partial [L^m_B]_{j,j+1}}{\partial u^m_j} + [\bar{R}^m_B]_{j,j+1} \frac{\partial [R^m_B]_{j,j+1}}{\partial u^m_j},$$

$$\bar{l}^m_{j+1} = [\bar{L}^m_B]_{j+1,j} \frac{\partial [L^m_B]_{j+1,j}}{\partial l^m_{j+1}} + [\bar{R}^m_B]_{j+1,j} \frac{\partial [R^m_B]_{j+1,j}}{\partial l^m_{j+1}},$$

where $j = 1, \ldots, N$ in the first equation and $j = 1, \ldots, N-1$ in the second and third. Here we have used the fact that each component of the vectors c^m, u^m and l^m appears only in one element of the three main diagonals of the matrices L^m_B and R^m_B. By Equations (94) and (95) it is immediate to verify that

$$\frac{\partial [L^m_B]_{j,j}}{\partial c^m_j} = \frac{\partial [L^m_B]_{j,j+1}}{\partial u^m_j} = \frac{\partial [L^m_B]_{j+1,j}}{\partial l^m_{j+1}} = -\phi,$$

$$\frac{\partial [R^m_B]_{j,j}}{\partial c^m_j} = \frac{\partial [R^m_B]_{j,j+1}}{\partial u^m_j} = \frac{\partial [R^m_B]_{j+1,j}}{\partial l^m_{j+1}} = 1 - \phi,$$

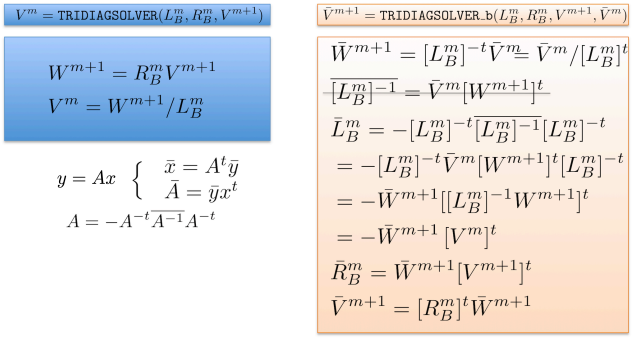

FIG. 18 Schematic illustration of a tri-diagonal solver and of its adjoint counterpart.

for $0 < \phi < 1$. For the fully explicit, $\phi = 0$, (resp. the fully explicit case, $\phi = 1$), L^m_B (resp. R^m_B) is the identity matrix and $\bar{L}^m_B$ (resp. $\bar{R}^m_B$) is identically zero.

$\bar{a}$) Compute the adjoints of the coefficients,

$$\bar{\theta} = \texttt{COMPUTECOEFFM\_b}(\theta, \bar{c}^m, \bar{u}^m, \bar{l}^m). \qquad (106)$$

This produces the following contribution of the adjoint of the $\bar{\theta}$ vector

$$\bar{\theta}_k = \sum_{j=1}^N \left[\bar{c}^m_j \frac{\partial c^m_j(\theta)}{\partial \theta_k} + \bar{u}^m_j \frac{\partial u^m_j(\theta)}{\partial \theta_k} + \bar{l}^m_j \frac{\partial l^m_j(\theta)}{\partial \theta_k} \right]$$

for $k = 1, \ldots, N_\theta$, with $u^m_N \equiv 0$ and $l^m_1 \equiv 0$.

($\bar{\text{S1}}$) Compute the adjoint of the vector V^M in (98) by executing $\bar{\theta} \mathrel{+}= \texttt{PAYOFF\_b}(\theta, \bar{V}^M)$. This gives the vector elements

$$\bar{\theta}_k \mathrel{+}= \sum_{j=1}^N \bar{V}^M_j \frac{\partial P(x_j, \theta)}{\partial \theta_k},$$

for $k = 1, \ldots, N_\theta$, associated with the explicit dependence of the payoff on the model parameters θ (if any).

One can verify that the execution of the steps above produces the sensitivities (104) of the option value with respect to the parameters θ. According to the general result of AAD (6), the cost to compute all the components of the adjoint vector $\bar{\theta}$ is a small multiplier of order four times the cost of computing (S1) to (S4), therefore resulting in an overall computation complexity of $O(NM)$.

We note that obtaining the adjoint COMPUTESPOT_b of the linear scheme in Equation (97) is straightforward. The procedure consists of setting $\bar{V}^0_j = 0$ for $j \notin \{j^\star, j^\star + 1\}$, and allocating $\bar{V}^0_{j^\star}$ and $\bar{V}^0_{j^\star+1}$ with their coefficients

in Equation (97), namely

$$\bar{V}^0_{j^\star} = \bar{V}_{t_0}\big(1 - \frac{x_{t_0} - x_{j^\star}}{x_{j^\star+1} - x_{j^\star}}\big), \quad \bar{V}^0_{j^\star+1} = \bar{V}_{t_0}\frac{x_{t_0} - x_{j^\star}}{x_{j^\star+1} - x_{j^\star}}.$$

The adjoint function `TRIADIAGSOLVER_b`, which gives the adjoint of (102), is produced by

$$\begin{aligned}
&\bar{W}^{m+1} = [L^m_B]^{-t}\,\bar{V}^m,\\
&\overline{[L^m_B]^{-1}} = \bar{V}^m\left[W^{m+1}\right]^t,\\
&\bar{L}^m_B = -\,[L^m_B]^{-t}\,\overline{[L^m_B]^{-1}}\,[L^m]^{-t}_B,\\
&\bar{R}^m_B = \bar{W}^{m+1}\left[V^{m+1}\right]^t,\\
&\bar{V}^{m+1} = [R^m_B]^t\,\bar{W}^{m+1}.
\end{aligned} \tag{107}$$

Here we have used the fact that the adjoint of the linear operation $y = Ax$ is given by $\bar{x} = A^t\bar{y}$ and $\bar{A} = \bar{y}x^t$, and the identity $\bar{A} = -A^{-t}\overline{A^{-1}}A^{-t}$, which holds for any invertible matrix A, see (Giles, 2008; **?**). [5]

The computational cost of the instructions above is $O(N^2)$. In order to reduce the computational cost to $O(N)$, as in the original sequence (102), one needs to avoid the matrix inversion in the first instruction of (107). This is obtained by utilising the solution of a linear system and then by combining the first three instructions of Eq. (107) and the second of Eq. (102). We thus have:

$$\begin{aligned}
\bar{L}^m_B &= -\,[L^m_B]^{-t}\,\bar{V}^m\left[W^{m+1}\right]^t[L^m]^{-t}_B\\
&= -\bar{W}^{m+1}\left[[L^m]^{-1}_B\,W^{m+1}\right]^t = -\bar{W}^{m+1}\,[V^m]^t\ .
\end{aligned}$$

Then, the resulting algorithm is given by

$$\begin{aligned}
&\bar{W}^{m+1} = \bar{V}^m / [L^m_B]^t,\\
&\bar{L}^m_B = -\bar{W}^{m+1}\,[V^m]^t,\\
&\bar{R}^m_B = \bar{W}^{m+1}\left[V^{m+1}\right]^t,\\
&\bar{V}^{m+1} = [R^m_B]^t\,\bar{W}^{m+1}\ .
\end{aligned} \tag{108}$$

We emphasise that only the elements on the three main diagonals of $\bar{L}^m_B$ and $\bar{R}^m_B$ contribute to the sensitivities, so that only $3N$ multiplications are required for their computation in the second and third instruction of Eq. (108). The overall computational cost of the adjoint tri-diagonal solver is $O(N)$, exactly as for the forward counterpart (102), and as expected from the general result (6).

The execution of the adjoint instructions (108) requires the vector V^m. This is a manifestation of the general feature of the adjoint implementation which require *i)* the execution of the original code, *ii)* the storage of the intermediate results and final outputs before the execution of its adjoint counterpart. In this case, `TRIADIAGSOLVER_b` needs to contain a forward sweep replicating the instructions (102) in order to compute V^m. Alternatively, if the values V_m were to be stored during the calculation in the forward sweep of (S1)-(S3), then one could use the stored values directly as inputs in `TRIADIAGSOLVER_b`. This scheme is more efficient as it avoids repeating the forward sweep. The first implementation comes with a reduced memory consumption as it does not store the vectors V^m for $m = 0, \ldots, M$ and is an example of the technique "checkpointing" (Capriotti and Giles, 2012).

[5] A collection of useful results for generic matrix functions is contained in (Goloubentsev *et al.*, 2021a).

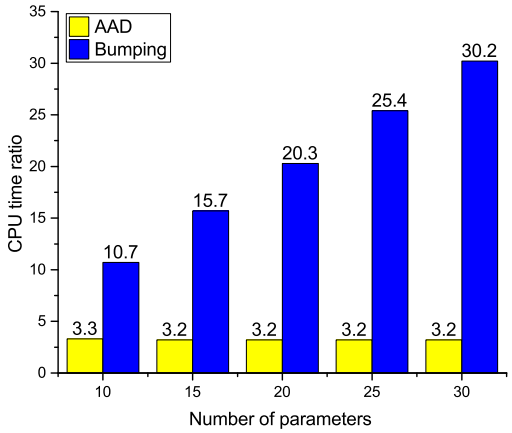

FIG. 19 Cost of computing the sensitivities for a defaultable discount bond option, relative to the cost of a single valuation, as a function of the number of sensitivities.

As in the MC application, AAD results in very significant computational savings as illustrated for instance in Fig. 19 for a simple example discussed in (Capriotti *et al.*, 2015). As expected, also in this case, the cost of computing any number of sensitivities relative to the cost of a single valuation is constant.

D. Other PDE Applications

An analysis similar to the one presented in the previous section can be conducted also for Kolmogorow forward equations, such those that are satisfied by transition density and Arrow-Debreu prices (Andersen and Piterbarg, 2010). This was demonstrated in (Capriotti *et al.*, 2015) and used for the calibration of terms structure or default intensity models.

Taking an algebraic approach (Denson and Joshi, 2010) demonstrates how the adjoint PDE method can be used to compute Greeks in Markov-functional models. They demonstrated the speed and accuracy of the method using a Markov-functional interest rate model, also showing how the model Greeks can be converted into market Greeks, along similar lines as in Sec. IV.

Finally, in (Bain *et al.*, 2019), the authors proposed a generic calibration framework to both vanilla and no-touch options for a large class of continuous semi-

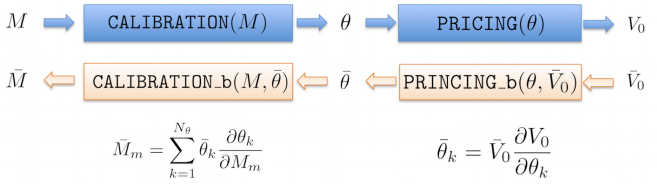

FIG. 20 Schematic illustration of the calibration and pricing sequence and its adjoint.

martingale models based upon a forward partial integro-differential equation (PIDE) which allows fast computation of up-and-out call prices for the complete set of strikes, barriers and maturities. Here they considered a Heston-type local-stochastic volatility model with local vol-of-vol, as well as two path-dependent volatility models where the local volatility component depends on the running maximum.

IV. CALIBRATION ALGORITHM: AAD AND THE IMPLICIT FUNCTION THEOREM

The valuation of a derivative security can be generally separated in two distinct steps, a calibration and a pricing step, see Fig. 20,

$$\theta = \texttt{CALIBRATION}(M), \tag{109}$$

the parameters of the model $\theta = (\theta_1, \ldots, \theta_{N_\theta})$, are calibrated in order to reprice simple and liquidly-traded financial instruments. We denote the price of such instruments with the market parameter vector $M = (M_1, \ldots, M_{N_M})$. In the pricing step, the parameters θ are mapped to the values of the derivative security, or portfolio of N_V securities:

$$V = \texttt{PRICING}(\theta), \tag{110}$$

so that the concatenation of the calibration of the calibration and the pricing step can be seen as a map of the form $M \to \theta \to V$.

The calibration step (109) typically involves an iterative routine, e.g., performing a numerical root search or least-square minimisation. While the calculation of the sensitivities with respect to the internal model parameters $\partial V/\partial\theta$ obtained by the adjoint of the pricing step (110), $\bar\theta = \texttt{PRICING\_b}(\theta, \bar V)$, which computes

$$\bar\theta_k = \sum_{i=1}^{N_V} \bar V_i \frac{\partial V_i}{\partial \theta_k},$$

for $k = 1, \ldots, N_\theta$, is sometimes useful, what is required for the risk management of the portfolio of the derivative securities are the sensitivities $\partial V/\partial M$ with respect to the liquid market prices because they define the size of the hedges. These can be obtained, according to the general principles of AAD, by reversing the order of computations so the adjoint of the algorithm consists of the adjoint pricing step, combined with the adjoint calibration step (see Fig. 20)

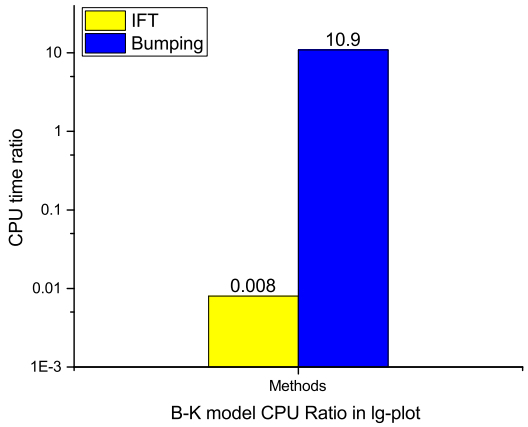

FIG. 21 Cost of computing the market parameter sensitivities for a defaultable discount bond option relative to the cost of a single calibration and valuation for the Black-Karasinski model (Capriotti *et al.*, 2015).

$$\bar M = \texttt{CALIBRATION\_b}(M, \bar\theta), \tag{111}$$

giving

$$\bar M_m = \sum_{k=1}^{N_\theta} \bar\theta_k \frac{\partial \theta_k}{\partial M_m},$$

for $m = 1, \ldots, N_M$. The overall adjoint algorithm can be seen therefore as a map of the form $\bar V \to \bar\theta \to \bar M$.

The adjoint calibration step (109) can be implemented according to the general rules of AAD (Section I.D), paying attention to its iterative nature. However, following the work by (Christianson, 1998), (Henrard, 2011) and (Capriotti and Lee, 2014), a much better performance can be obtained by exploiting the so-called *implicit function theorem* (IFT), as described below. Here we consider the case in which the calibration algorithm in Equation (111) consists of the numerical solution of a system of equations of the form

$$G_i(M, \theta) = 0 \tag{112}$$

where $M \in \mathcal{R}^{N_M}$, $\theta \in \mathcal{R}^{N_\theta}$, and $i = 1, \ldots, N_\theta$. The function $G_i(M, \theta)$ is often of the form

$$G_i(M, \theta) = T_i(M) - V_i(\theta) \tag{113}$$

where $V_i(\theta)$ is the price of the i-th calibration instrument as produced by the model to be calibrated, and $T_i(M)$ are the prices of the target instruments, possibly generated by a simpler model utilised as a quoting mechanism.

As noted above, the adjoint calibration can be implemented in terms of the adjoint of the numerical scheme

solving (112). The associated computational cost is expected to be a few times the cost of solving the numerical system (112) (but approximately less than 4 times the cost, according to the general result of AAD). Better performance can be obtained by the IFT. Under mild regularity conditions, the IFT says that if there is a solution (M_0, θ_0) to the root finding problem (112), such that $G_i(M_0, \theta_0) = 0$, and the matrix of derivatives $[\partial G/\partial\theta]_{ij} = \partial G_i(M_0, \theta_0)/\partial\theta_j$ is invertible, then one can define in the vicinity of M_0 an implicit function $\theta = \theta(M)$ such that

$$G_i(M, \theta(M)) = 0. \tag{114}$$

The derivatives $\partial\theta/\partial M$ of such function can be expressed in terms of the derivatives of the objective function G. Indeed, by differentiating (114) with respect to M, one obtains

$$\frac{\partial G_i}{\partial M_m} + \sum_{j=1}^{N_\theta} \frac{\partial G_i}{\partial \theta_j}\frac{\partial \theta_j}{\partial M_m} = 0$$

for $m = 1, \ldots, N_M$, or equivalently

$$\frac{\partial \theta_k}{\partial M_m} = -\left[\left(\frac{\partial G}{\partial \theta}\right)^{-1}\frac{\partial G}{\partial M}\right]_{km},$$

with $[\partial G/\partial M]_{ij} = \partial G_i/\partial M_j$. This relation allows the computation of the sensitivities of the function $\theta(M)$, locally defined in an implicit manner by Equation (112), in terms of the sensitivities of the function $G(M, \theta)$. These can be computed by the corresponding adjoint function $(\bar{M}, \bar{\theta}) = \bar{G}(M, \theta, \bar{G})$ giving,

$$\bar{M}_m = \sum_{i=1}^{N_\theta} \bar{G}_i \frac{\partial G_i}{\partial M_m}, \qquad \bar{\theta}_k = \sum_{i=1}^{N_\theta} \bar{G}_i \frac{\partial G_i}{\partial \theta_k}.$$

This method is far more efficient and stable than calculating the derivatives of the implicit functions $M \to \theta(M)$ by differentiating directly the calibration step either by bumping or by applying AAD. This is because $G(M, \theta)$ in (113) are *explicit* functions of the market and model parameters, which are easy to compute and differentiate. Moreover, by avoiding the numerical noise produced by the finite difference approximation to the calibration procedure, the accuracy of the sensitivities is improved when compared with the bumping scheme.

The remarkable computational gains that can be achieved with the AAD IFT scheme are shown in Figure 21 (left, yellow column) for a credit risk application involving the solution of the PDE for a defaultable bond under the Black-Karasinki (B-K) model (Capriotti *et al.*, 2015). Here we plot the ratio of time necessary to convert the model sensitivities into market sensitivities by both, the ADD-IFT approach and standard finite differences, relative to the cost of performing a *single* calibration and valuation. For this application, the time necessary to compute the Jacobian $\partial\theta/\partial M$ and model parameter sensitivities $\partial V/\partial\theta$ by the AAD-IFT approach is just 0.8%

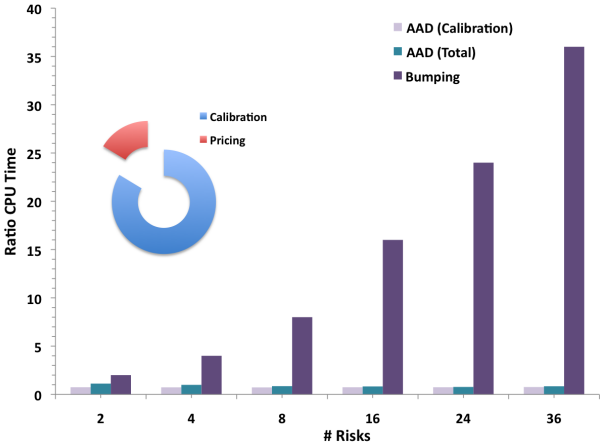

FIG. 22 Cost of computing the sensitivities - relative to the cost of a single valuation - as a function of the number of sensitivities for a portfolio of CDS (Capriotti and Lee, 2014)

the amount of time necessary to perform a single calibration and valuation, thus resulting in 3 orders of magnitude speed-up with respect to standard bumping (Capriotti *et al.*, 2015).

V. OTHER FINANCIAL APPLICATIONS

AAD is potentially beneficial in every numerical scheme involving the calculation of derivatives.

One of the earliest applications was a MC based calibration of financial market models, for which Käbe *et al.*, 2009 proposed the application of adjoint methods in combination with a multilayer approach. For a lognormal variance model the authors showed how the adjoint-based MC algorithm reduces computation time from more than three hours (for finite difference approximation of the gradient) to less than ten minutes.

AAD can be beneficial for the risk management of flow credit products, which are evaluated by simple quadrature methods, such as credit default swap (CDS), CDS indices and swaptions, as demonstrated in (Capriotti and Lee, 2014). Here the authors showed how by combining adjoint ideas with the IFT one can avoid the necessity of repeating multiple times the calibration of the hazard rate curves which, especially for flow products, often represent the bottle neck in the computation of spread and interest rate risk. The remarkable computational efficiency achievable by combining AAD with the IFT is illustrated in Fig. 22 showing how sensitivities can be computed in $\sim 25\%$ *less* time than performing a single calibration and valuation. This typically results in orders of magnitudes savings in computational time with respect to the standard bump and reval method. In addition, since AAD produces analytical derivatives rather than finite differences approximations the calculation is much more robust numerically than bumping, which is

instead often affected by the problem that arbitrary perturbations of credit spreads, recovery rates or of the discount curve may lead to an arbitrageable hazard rate curve.

Goloubentsev *et al.*, 2021b recently revisited the problem of applying AAD to the IFT and illustrated how automatic tools can be utilized to implement calibration algorithms based on the IFT with minimal manual intervention. The efficient implementation of AAD applied to calibration processes in parallel architecture is discussed in (Brito *et al.*, 2022; Goloubentsev and Lakshtanov, 2022).

Finally, there is a deep connection of machine learning (ML) and AAD. As a matter of fact the so-called backpropagation algorithm for the calibration of artificial neural networks (Baydin *et al.*, 2018) is simply another instance of AAD. More advanced applications include the combination of ML and AAD. For instance Huge and Savine, 2020 introduced the notion of 'Differential ML' combining AAD with ML in the context of risk management of financial Derivatives. Differential ML is a general extension of supervised learning, where ML models are trained on examples of not only inputs and outputs but also differentials of outputs with respect to the inputs. Similar ideas are also discussed in (Ludkovski, 2023). Other applications combining ML ideas and AAD include the calibration of the rough Heston model (Rosenbaum and Zhang, 2021) via neural networks.

VI. FROM FINANCE TO PHYSICS

Not much longer after its introduction to Finance, AAD made its appearance in computational Physics (Sorella and Capriotti, 2010), and in particular in the field of simulations based on first principles quantum mechanics – or *ab-initio* (Tuckerman *et al.*, 1996) – of strongly correlated electronic systems. For these systems, quantum Monte Carlo allows to accurately account for electronic correlations, which are key for the descriptions of several materials, including high temperature superconductors (Fradkin *et al.*, 2015). In particular, structural optimization and molecular dynamics at finite temperature, requires the calculation of the forces acting on atoms, e.g., in a molecule or a condensed matter system, which can be formally expressed as the calculation of derivatives of an energy functional with respect to the atomic coordinates. However, because of its high computational cost, the use of quantum Monte Carlo for these problems had been fairly limited and the prerogative of less accurate techniques such Density Functional Theory (Cohen *et al.*, 2012). In this background, (Sorella and Capriotti, 2010) showed how by means of AAD the calculation of forces of an atomic system can be implemented in a straightforward fashion and executed much more efficiently than with existing techniques. A sizable and diverse literature in computational Physics has followed this seminal work ranging from the study of the physical properties of Hydrogen at very high pressure (Mazzola *et al.*, 2014) and of several other compounds (Nakano *et al.*, 2020), the efficient implementation of Tensor Networks for strongly correlated lattice systems, such as quantum antiferromagnets (Liao *et al.*, 2019), and even quantum computation (Kottmann *et al.*, 2021).

VII. CONCLUSIONS

Over the past two decades, Adjoint Algorithmic Differentiation (AAD) has revolutionized the way risk is computed in the financial industry. Following the seminal work by Giles and Glasserman, 2006 introducing for the first time adjoint methods for MC applications in a financial context, several early works have illustrated (Capriotti, 2008, 2011; Capriotti and Giles, 2012) how the algorithmic approach of AAD can be employed, making the adjoint implementation practical in an industrial environment. Like its algebraic counterpart, the proposed method allows the calculation of the complete risk at a computational cost which is at most 4 times the cost of calculating the P&L of the portfolio itself, resulting in remarkable computational savings, often of several order of magnitude, with respect to standard finite-differences.

In this review, we have shown how AAD can be used to implement the adjoint calculation of price sensitivities in a straightforward manner and in wide generality and have reviewed the most significant applications appearing in the recent literature. Among others, we have covered Monte Carlo, Partial Differential Equation and Calibration applications in Finance. We have also described how AAD was recently introduced to ab-initio Quantum Monte Carlo simulations, one of the sparse examples in which Finance, a field with a long history of borrowing concepts and techniques from Applied Mathematics and Natural Science, "gave back" some knowledge to Physics.

In many situations, AAD allows one to perform in minutes risk runs that would take otherwise several hours or could not even be performed overnight without the use of massively parallel computers. AAD therefore makes possible real time risk management, allowing investment firms to hedge their positions and manage their capital allocation more effectively, reduce their infrastructure costs, and ultimately attract more business, increase profitability and reduce their environmental impact.

Acknowledgments

It is a pleasure to acknowledge a fruitful collaboration over several years with Jacky Lee, Adam Peacock, Matthew Peacock, Mark Stedman, Uwe Naumann, Alex Prideaux, Yupeng Jiang, and Andrea Macrina.

Disclosure of Interests

No potential competing interest was reported by the authors. No funding was received for this research. The opinions and views expressed in this paper are solely those of the authors and do not reflect those of their respective employers.

Appendix A: Adjoint programming in a nutshell

A detailed tutorial on the programming techniques that are useful for adjoint implementations is beyond the scope of this paper. However, when hand-coding the adjoint counterpart of a set of instructions it is often enough to keep in mind just a few practical recipes. For instance:

a) Each intermediate differentiable variable U can be used not only by the subsequent instruction but also by several others occurring later in the program. As a result, the adjoint of U has several contributions, one for each instruction of the original function in which U was on the right hand side of the assignment operator. Hence, by exploiting the linearity of differential operators, it is generally easier to program according to a syntactic paradigm in which adjoints are always updated so that the adjoint of an instruction of the form

$$V = V(U)$$

reads

$$\bar{U}_i = \bar{U}_i + \sum_k \frac{\partial V_k(U)}{\partial U_i} \bar{V}_k \ .$$

This implies that the adjoints have to be appropriately initialized. In particular, to cope with input variables that are changed by the algorithm (see next point), it is generally best to initialize the adjoint of a given variable to zero on the instruction in which it picks up its first contribution (i.e., immediately before the adjoint counterpart of the last instruction of the original code in which the variable was to the right of the assignment operator). For instance, the adjoint of the following sequence of instructions where x is the input, u and v are local variables, and y is the output

$$\begin{aligned} u &= F(x) \\ v &= G(x,u) \\ y &= H(v) \end{aligned}$$

can be written as:

$$\begin{aligned} \bar{v} &= 0 \\ \bar{v} &= \bar{v} + \frac{\partial H(v)}{\partial v}\bar{y} \\ \bar{u} &= 0 \\ \bar{u} &= \bar{u} + \frac{\partial G(x,u)}{\partial u}\bar{v} \\ \bar{x} &= 0 \\ \bar{x} &= \bar{x} + \frac{\partial G(x,u)}{\partial x}\bar{v} \\ \bar{x} &= \bar{x} + \frac{\partial F(x)}{\partial x}\bar{u} \ , \end{aligned}$$

where $\bar{y}$ is the input, $\bar{u}$ and $\bar{v}$ are local variables, and $\bar{x}$ is the output. Note that the life-cycle of an adjoint variable terminates after the adjoint of the instruction that initializes the corresponding forward variable. For instance, in the example above $\bar{y}$ can be reset to zero after the second adjoint instruction, $\bar{v}$ after the sixth, and $\bar{u}$ after the seventh. Doing so explicitly, although somewhat redundant, is often a helpful programming idiom.

b) In some situations the input U of a function $V = V(U)$ is modified by the function. As above, this situation is easily analyzed by introducing an auxiliary variable U' representing the value of the input after the functions evaluation. Therefore, the original function can be thought of the form $(V, U') = (V(U), U'(U))$, where $V(U)$ and $U'(U)$ do not mutate their inputs, in combination with the assignment $U = U'$, overwriting the original input U. The adjoint of this pair of instructions clearly reads

$$\begin{aligned} \bar{U}'_i &= 0 \\ \bar{U}'_i &= \bar{U}'_i + \bar{U}_i \ , \end{aligned}$$

where we have used the fact that the auxiliary variable U' is not used elsewhere (so $\bar{U}'_i$ does not have any previous contribution), and

$$\begin{aligned} \bar{U}_i &= 0 \\ \bar{U}_i &= \bar{U}_i + \sum_k \frac{\partial V_k(U)}{\partial U_i}\bar{V}_k + \sum_l \frac{\partial U'_l(U)}{\partial U_i}\bar{U}'_l \ . \end{aligned}$$

where, again, we have also used the fact that the original input U is not used after the instruction $V = V(U)$ as it gets overwritten. One can therefore eliminate altogether the adjoint of the auxiliary variable $\bar{U}'$ and simply write

$$\bar{U}_i \mathrel{+}= \sum_k \frac{\partial V_k(U)}{\partial U_i}\bar{V}_k \ .$$

Very common examples of this situation are given by increments of the form

$$U_i = a\,U_i + b$$

with a and b constant with respect to U. According to the above recipe, the adjoint counterpart of this instruction simply reads

$$\bar{U}_i = a\,\bar{U}_i \ .$$

These situations are common in iterative loops where a number of variables are typically updated at each iteration.

c) Each function, subroutine or method can be abstracted as a function with some inputs and some outputs even if some of these variables are implicit. For instance, in an object oriented language, a class constructor can be seen as a function whose (implicit) outputs are the member variables of the class. These member variables, say θ, can be also seen as implicit inputs of all the other methods of the class, e.g.,

$$Y = \texttt{METHOD}(X, \theta) \ . \tag{A1}$$

Hence, the corresponding adjoint methods – in addition to the sensitivities to its explicit inputs – generally produce the sensitivities with respect to the member variables, $\bar{\theta}$, e.g.,

$$(\bar{X}, \bar{\theta}) \mathrel{+}= \texttt{METHOD\_b}(X, \theta, \bar{Y}) \ , \tag{A2}$$

where we have used the standard addition assignment operator +=.

References

Achdou, Y., and O. Pironneau, 2005, *Computational methods for option pricing* (SIAM).

Andersen, L., and V. Piterbarg, 2010, *Interest Rate Modeling* (Atlantic Financial Press).

Andreasen, J., 2023, Wilmott **2023**(128), ISSN 1541-8286, URL `http://dx.doi.org/10.54946/wilm.1115880`.

Antonov, A., 2016, Avaliable at SSRN 2839362 .

Antonov, A., S. Issakov, and A. McClelland, 2017, Available at SSRN 3040061 .

Bain, A., M. Mariapragassam, and C. Reisinger, 2019, arXiv preprint arXiv:1911.00877 .

Baydin, A. G., B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, 2018, Journal of Marchine Learning Research **18**, 1.

Brace, A., D. Gatarek, and M. Musiela, 1997, Mathematical Finance **7**, 127.

Brito, J., A. Goloubentsev, and E. Goncharov, 2022, arXiv preprint arXiv:2204.05204 .

Broadie, M., and P. Glasserman, 1996, Management Science **42**, 269.

Capriotti, L., 2008, US Patent 0312530 .

Capriotti, L., 2011, Journal of Computational Finance **3**, 3.

Capriotti, L., 2015, Algorithmic Finance **4**, 81.

Capriotti, L., and M. Giles, 2010, Risk **23**, 79.

Capriotti, L., and M. Giles, 2012, Risk **25**, 92.

Capriotti, L., Y. Jiang, and A. Macrina, 2015, International Journal of Financial Engineering **2**, 1550039.

Capriotti, L., Y. Jiang, and A. Macrina, 2017, Algorithmic Finance **6**, 35.

Capriotti, L., and J. Lee, 2014, Risk **27**, 48.

Capriotti, L., M. Peacock, and J. Lee, 2011, Risk **24**, 86.

Cesa, M., 2017, Probability, Uncertainty and Quantitative Risk **2**, 1.

Chan, J. H., M. S. Joshi, and D. Zhu, 2015, Journal of Risk **17**, 4.

Christianson, B., 1998, Optimization Methods and Software **9**, 307.

Cohen, A. J., P. Mori-Sánchez, and W. Yang, 2012, Chemical reviews **112**, 289.

Denson, N., and M. Joshi, 2010, *Fast Greeks for Markov-functional Models Using Adjoint PDE Methods*, Research paper series: Centre for Actuarial Studies (Centre for Actuarial Studies, Fac. of Economics & Commerce, University of Melbourne), URL `https://books.google.com/books?id=KawXyAEACAAJ`.

Denson, N., and M. Joshi, 2011, Journal of Computational Finance **14**, 115.

Fournié, E., J.-M. Lasry, J. Lebuchoux, P.-L. Lions, and N. Touzi, 1999, Finance and Stochastics **3**, 391.

Fradkin, E., S. A. Kivelson, and J. M. Tranquada, 2015, Reviews of Modern Physics **87**, 457.

Fries, C. P., 2019, Quantitative Finance **19**, 1043.

Geeraert, S., C.-A. Lehalle, B. A. Pearlmutter, O. Pironneau, and A. Reghai, 2017, ESAIM: Proceedings and Surveys **59**, 56.

Giering, R., and T. Kaminski, 2006, ACM Transaction on Mathematical Software **24**, 437.

Giles, M., 2008, Unpublished .

Giles, M., 2009, in *Monte Carlo and Quasi-Monte Carlo Methods 2008* (Springer), p. 369.

Giles, M., M. Duta, J.-D. Müller, and N. Pierce, 2003, AIAA Journal **41**, 198.

Giles, M., and P. Glasserman, 2006, Risk **19**, 88.

Giles, M., and N. Pierce, 2000, Flow, Turbulence and Combusion **65**, 393.

Glasserman, P., 2004, *Monte Carlo Methods in Financial Engineering* (Springer, New York).

Glasserman, P., and B. Yu, 2004, in *Monte Carlo and Quasi-Monte Carlo Methods 2002*, edited by H. Niederreiter (Springer Berlin Heidelberg, Berlin, Heidelberg), p. 213.

Goloubentsev, A., D. Goloubentsev, and E. Lakshtanov, 2021a, arXiv preprint arXiv:2109.04913 .

Goloubentsev, D., and E. Lakshtanov, 2019, Wilmott **2019**, 8.

Goloubentsev, D., and E. Lakshtanov, 2022, International Journal of Wavelets, Multiresolution and Information Processing **20**(03), 2040004.

Goloubentsev, D., E. Lakshtanov, and V. Piterbarg, 2021b, Available at SSRN 3984964 .

Griewank, A., 2000, *Evaluating derivatives: principles and techniques of algorithmic differentiation (first edition)* (SIAM, Philadelphia).

Griewank, A., and A. Walther, 2008, *Evaluating derivatives: principles and techniques of algorithmic differentiation (second edition)* (SIAM).

Henrard, M., 2011, OpenGamma Quantitative Research, **1**, 1.

Henrard, M., 2017, *Algorithmic differentiation in finance explained* (Springer).

Huge, B., and A. Savine, 2017, Available at SSRN 2966155 .

Huge, B., and A. Savine, 2020, arXiv preprint

arXiv:2005.02347 .
Hull, J. C., 2002, *Options, Futures and Other Derivatives* (Prentice Hall, New Jersey).
Jain, S., A. Leitao, and C. Oosterlee, 2019, Journal of Computational Science **33**, 95.
Jain, S., and C. W. Oosterlee, 2015, Applied Mathematics and Computation **269**, 412.
Joshi, M., and C. Yang, 2010, Available at SSRN 1689348 .
Joshi, M., and C. Yang, 2011, IIEE Transactions **43**, 878.
Joshi, M. S., and D. Kainth, 2004, Quantitative Finance **4**, 266.
Käbe, C., J. H. Maruhn, and E. W. Sachs, 2009, Finance and Stochastics **13**, 351.
Kottmann, J. S., A. Anand, and A. Aspuru-Guzik, 2021, Chemical science **12**, 3497.
Leclerc, M., Q.Liang, and I. Schneider, 2009, Risk **22**, 84.
Liao, H.-J., J.-G. Liu, L. Wang, and T. Xiang, 2019, Physical Review X **9**, 031041.
Longstaff, F. A., and E. Schwartz, 2001, Review of Financial Studies **14**, 113.
Ludkovski, M., 2023, Annual Review of Statistics and its Application **10**, 271.
Mazzola, G., S. Yunoki, and S. Sorella, 2014, Nature Communications **5**, 3487.
Nakano, K., C. Attaccalite, M. Barborini, L. Capriotti, M. Casula, E. Coccia, M. Dagrada, C. Genovese, Y. Luo, G. Mazzola, *et al.*, 2020, The Journal of Chemical Physics **152**, 234111.
Naumann, U., 2008a, in *C. Bischof et al.: Advances in Automatic Differentiation* (Springer), pp. 13–22.
Naumann, U., 2008b, Mathematical Programming **112**(2), 427, ISSN 0025-5610, URL `http://dx.doi.org/10.1007/s10107-006-0042-z`.
Naumann, U., 2009, Journal of Discrete Algorithms **7**, 402.
Naumann, U., 2011, *The art of differentiating computer programs: an introduction to algorithmic differentiation* (SIAM).
Naumann, U., and J. Toit, 2018, Journal of Computational Finance **21**, 23.
Pironneau, O., G. Sall, *et al.*, 2016, arXiv preprint arXiv:1606.06143 .
Rosenbaum, M., and J. Zhang, 2021, arXiv preprint arXiv:2107.01611 .
Savine, A., 2018, *Modern computational finance: AAD and parallel simulations* (John Wiley & Sons).
Silotto, L., M. Scaringi, and M. Bianchetti, 2023, Annals of Operations Research , 1.
Smith, S., 1995, Journal of Computational and Graphic Statistics **4**, 134.
Sorella, S., and L. Capriotti, 2010, The Journal of chemical physics **133**, 234111.
Tsitsiklis, J., and B. V. Roy, 2001, EEE Transactions on Neural Networks **12**, 694.
Tuckerman, M. E., P. J. Ungar, T. Von Rosenvinge, and M. L. Klein, 1996, The Journal of Physical Chemistry **100**, 12878.