The Journal of Computational Finance (3–35) Volume 14/Number 3, Spring 2011

Fast Greeks by algorithmic differentiation

Luca Capriotti
Quantitative Strategies, Investment Banking Division, Credit Suisse Group,
Eleven Madison Avenue, New York, NY 10010-3086, USA;
email: luca.capriotti@credit-suisse.com

We show how algorithmic differentiation can be used to efficiently implement the pathwise derivative method for the calculation of option sensitivities using Monte Carlo simulations. The main practical difficulty of the pathwise derivative method is that it requires the differentiation of the payout function. For the type of structured options for which Monte Carlo simulations are usually employed, these derivatives are typically cumbersome to calculate analytically, and too time consuming to evaluate with standard finite-difference approaches. In this paper we address this problem and show how algorithmic differentiation can be employed to calculate these derivatives very efficiently and with machine-precision accuracy. We illustrate the basic workings of this computational technique by means of simple examples, and we demonstrate with several numerical tests how the pathwise derivative method combined with algorithmic differentiation – especially in the adjoint mode – can provide speed-ups of several orders of magnitude with respect to standard methods.

1 INTRODUCTION

Monte Carlo simulations are becoming the main tool in the financial services industry for pricing and hedging complex derivatives securities. In fact, as a result of the ever-increasing level of sophistication of the financial markets, a considerable fraction of the pricing models employed by investment firms is too complex to be treated by analytic or deterministic numerical methods. For these models, Monte Carlo simulation is the only computationally feasible pricing technique.

It is a pleasure to acknowledge useful discussions and correspondence with Mike Giles, Paul Glasserman, Laurent Hascoët, Jacky Lee, Jason McEwen, Adam and Matthew Peacock, Alex Prideaux and David Shorthouse. Special thanks are also due to Mike Giles for a careful reading of the manuscript and valuable feedback. Constructive criticisms and suggestions by the two anonymous referees and the associate editor are also gratefully acknowledged. The algorithmic differentiation codes used in this paper were generated using TAPENADE, developed at INRIA. The opinion and views expressed in this paper are those of the author alone and do not necessarily represent those of Credit Suisse Group.

The main drawback of Monte Carlo simulations is that they are generally computationally expensive. These efficiency issues become even more dramatic when Monte Carlo simulations are used for the calculation of the "Greeks", or price sensitivities, which are necessary to hedge the financial risk associated with a derivative security. Indeed, the standard method for the calculation of the price sensitivities, also known as "bumping", involves perturbing the underlying model parameters in turn, repeating the simulation and forming finite-difference approximations, thus resulting in a computational burden increasing linearly with the number of sensitivities computed. This becomes very significant when the models employed depend on a large number of parameters, as is typically the case for the sophisticated models for which Monte Carlo simulations are used in practice.

Alternative methods for the calculation of price sensitivities have been proposed in the literature (for a review see, for example, Glasserman (2004)). Among these, the pathwise derivative method (Broadie and Glasserman (1996); Chen and Fu (2002); and Glasserman (2004)) provides unbiased estimates at a computational cost that for simple problems is generally smaller than that of bumping. The main limitation of the technique is that it involves the differentiation of the payout function. These derivatives are usually cumbersome to evaluate analytically, thus making the practical implementation of the pathwise derivative method problematic. Of course, the payout derivatives can always be approximated by finite-difference estimators. However, this involves multiple evaluations of the payout function, and it is in general computationally expensive.

A more efficient implementation of the pathwise derivative method was proposed in a remarkable paper by Giles and Glasserman (2006) for the LIBOR market model and European-style payouts, and was recently generalized for simple Bermudan options by Leclerc *et al* (2009). The main advantage of this method is that it allows the calculation of the price sensitivities at a greatly reduced price with respect to the standard implementation. One drawback, as in the standard pathwise derivative method, is that it involves the calculation of the derivatives of the payout function with respect to the value of the underlying market factors.

In this paper we illustrate how the problem of the efficient calculation of the derivatives of the payout can be overcome by using algorithmic differentiation (AD) (see Griewank (2000)). Indeed, AD makes possible the automatic generation of efficient coding implementing the derivatives of the payout function. In particular, the tangent (or forward) mode of AD is well suited for highly vectorized payouts depending on a small number of observations of a limited number of underlying assets. This situation arises, for instance, when the same payout function with different parameters is used to price several trades depending on the same (small) set of market observations, and the sensitivities associated with each trade are required. On the other hand, the adjoint (or backward) mode is most efficient in the more common situations where the number of observations of the underlying assets is larger than the number of securities simultaneously evaluated in the payout, or when interest lies in the aggregated risk of a portfolio. In these cases, the implementation of the pathwise derivative method by means of the adjoint mode of AD – adjoint algorithmic differentiation (AAD) – can provide speed-ups of several orders of magnitude with respect to standard methods.

In companion papers (Capriotti and Giles (2010) and Capriotti and Giles (2011)), we show how AAD can be used to efficiently implement not only the derivatives of the payout function but also the so-called tangent state vector (see the next section). This allows us to make the ideas proposed in Giles and Glasserman (2006) completely general and applicable to virtually any derivative or model commonly used in the financial industry.[1]

The remainder of this paper is organized as follows. In the next section, we set the notation and briefly review the pathwise derivative method. The basic workings of AD are then introduced in Section 3 by means of simple examples. In Section 4 we discuss in detail how AD can be used to generate efficient coding for the calculation of the derivatives of the payout function. Here we present numerical results comparing the efficiency of the tangent and adjoint modes as a function of the number of derivatives computed. From this discussion it will be clear that, in most circumstances, the adjoint mode is the one that is best suited when calculating the risk of complex derivatives. The computational efficiency of the pathwise derivative method with adjoint payouts is discussed and tested with several numerical examples in Section 5. We draw the conclusions of this paper in Section 6.

2 THE PATHWISE DERIVATIVE METHOD

Option pricing problems can typically be formulated in terms of the calculation of expectation values of the form (Harrison and Kreps (1979)):

$$V = \mathbb{E}_{\mathbb{Q}}[P(X(T_1), \dots, X(T_M))] \tag{2.1}$$

Here $X(t)$ is an N-dimensional vector and represents the value of a set of underlying market factors (stock prices, interest rates, foreign exchange pairs, etc) at time t. $P(X(T_1), \dots, X(T_M))$ is the payout function of the priced security, and depends in general on M observations of those factors. In the following, we will denote the collection of such observations by a $d = (N \times M)$-dimensional state vector $X = (X(T_1), \dots, X(T_M))^{\mathrm{T}}$, and by $\mathbb{Q}(X)$ the appropriate risk-neutral distribution (Harrison and Kreps (1979)) according to which the components of X are distributed.

[1] The connection between AD and the adjoint approach of Giles and Glasserman (2006) is also discussed in Giles (2007).

The expectation value in (2.1) can be estimated by means of Monte Carlo by sampling a number N_{MC} of random replicas of the underlying state vector $X[1], \dots, X[N_{\mathrm{MC}}]$, sampled according to the distribution $\mathbb{Q}(X)$, and evaluating the payout $P(X)$ for each of them. This leads to the central limit theorem (Kallenberg (1997)) estimate of the option value V as:

$$V \simeq \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} P(X[i_{\mathrm{MC}}]) \tag{2.2}$$

with standard error $\Sigma/\sqrt{N_{\mathrm{MC}}}$, 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 (Equation (2.1)) with respect to a set of N_θ parameters $\theta = (\theta_1, \dots, \theta_{N_\theta})$, with a single simulation. This can be achieved by noticing that, whenever the payout function is regular enough (eg, Lipschitz continuous), and under additional conditions that are often satisfied in financial pricing (see, for example, Glasserman (2004)), the sensitivity $\bar{\theta}_k \equiv \partial V/\partial\theta_k$ can be written as:

$$\bar{\theta}_k = \mathbb{E}_{\mathbb{Q}}\left[\frac{\partial P_\theta(X)}{\partial\theta_k}\right] \tag{2.3}$$

In general, the calculation of Equation (2.3) can be performed by applying the chain rule, and averaging on each Monte Carlo path the so-called pathwise derivative estimator:

$$\bar{\theta}_k \equiv \frac{\partial P_\theta(X)}{\partial\theta_k} = \sum_{j=1}^{d} \frac{\partial P_\theta(X)}{\partial X_j}\frac{\partial X_j}{\partial\theta_k} + \frac{\partial P_\theta(X)}{\partial\theta_k} \tag{2.4}$$

It is worth noting (although it is generally overlooked in the academic literature) that the payout $P_\theta(X(\theta))$ may depend on θ not only implicitly through the vector $X(\theta)$, but also explicitly. The second term in Equation (2.4) is therefore important and needs to be kept in mind when implementing the pathwise derivative method.

The matrix of derivatives of each state variable in (2.4), or tangent state vector, is by definition given by:

$$\frac{\partial X_j}{\partial\theta_k} = \lim_{\Delta\theta\to 0} \frac{X_j(\theta_1, \dots, \theta_k + \Delta\theta, \dots, \theta_{N_\theta}) - X_j(\theta)}{\Delta\theta} \tag{2.5}$$

This gives the intuitive interpretation of $\partial X_j/\partial\theta_k$ in terms of the difference between the sample of the jth component of the state vector obtained after an infinitesimal "bump" of the kth parameter, $X_j(\theta_1, \dots, \theta_k + \Delta\theta, \dots, \theta_{N_\theta})$, and the base sample $X_j(\theta)$, both calculated on the same random realization.

In the special case in which the state vector $X = (X(T_1), \dots, X(T_M))$ is a path of an N-dimensional diffusive process, the pathwise derivative estimator (2.4) may be rewritten as:

$$\bar{\theta}_k = \sum_{l=1}^{M} \sum_{j=1}^{N} \frac{\partial P(X(T_1), \dots, X(T_M))}{\partial X_j(T_l)} \frac{\partial X_j(T_l)}{\partial \theta_k} + \frac{\partial P_\theta(X)}{\partial \theta_k} \tag{2.6}$$

where we have relabeled the d components of the state vector X grouping together different observations $X_j(T_1), \dots, X_j(T_M)$ of the same (jth) asset. In particular, the components of the tangent vector for the kth sensitivity corresponding to observations at times $(T_1, \dots, T_M)$ along the path of the jth asset, say:

$$\Delta_{jk}(T_l) = \frac{\partial X_j(T_l)}{\partial \theta_k} \tag{2.7}$$

with $l = 1, \dots, M$, can be obtained by solving a stochastic differential equation (Kunita (1990); Protter (1997); and Glasserman (2004)).

The pathwise derivative estimators of the sensitivities are mathematically equivalent[2] to the estimates obtained by bumping, using the same random numbers in both simulations, and for a vanishingly small perturbation. In fact, when using the same set of random numbers for the base and bumped simulations of the expectation in (2.2), the finite-difference estimator of the kth sensitivity is equivalent to the average over the Monte Carlo paths of the quantity:

$$\frac{P(X(\theta^{(k)})[i_{\mathrm{MC}}]) - P(X(\theta)[i_{MC}])}{\Delta\theta} \tag{2.8}$$

with $\theta^{(k)} = (\theta_1, \dots, \theta_k + \Delta\theta, \dots, \theta_{N_\theta})$. In the limit $\Delta\theta \to 0$, this is equivalent in turn to Equation (2.3). As a result, the pathwise derivative method and bumping provide in the limit $\Delta\theta \to 0$ exactly the same estimators for the sensitivities, ie, estimators with the same expectation value, and the same Monte Carlo variance.

Since bumping and the pathwise derivative method provide estimates of the option sensitivities with comparable variance, the implementation effort associated with the latter is generally justified if the computational cost of the estimator (2.3) is less than the corresponding one associated with bumping.

The computational cost of evaluating the tangent state vector is strongly dependent on the problem considered. In some situations, its calculation can be implemented at a cost that is smaller than that associated with the propagation of the perturbed paths in bumping. Apart from very simple models, this is the case, for instance, in the

[2] Provided that the state vector is a sufficiently regular function of θ (Glasserman (2004) and Protter (1997)).

examples considered by Glasserman and Zhao (1999) in the context of the LIBOR market model.

However, when an efficient implementation of the tangent state vector is possible, the calculation of the gradient of the payout can constitute a significative part of the total computational cost, decreasing or eliminating altogether the benefits of the pathwise derivative method. Indeed, payouts of structured products often depend on hundreds of observations of several underlying assets, and their calculation is generally time consuming. For these payouts, the analytic calculation of the gradient is usually too cumbersome so that finite differences are the only practical route available (Giles and Glasserman (2006)). The multiple evaluation of the payout to obtain a large number of gradient components can by itself make the pathwise derivative method less efficient than bumping.

The efficient calculation of the derivatives of the payout is therefore critical for the successful implementation of the pathwise derivative method. In the following, we will illustrate how such an efficient calculation can be achieved by means of AD. In particular, we will show that the adjoint mode of AD allows the gradient of the payout function to be obtained at a computational cost that is bounded by approximately four times the cost of evaluating the payout itself, thus solving one of the main implementation difficulties – and performance bottlenecks – of the pathwise derivative method.

We will begin by reviewing the main ideas behind this powerful computational technique in the next section.

3 ALGORITHMIC DIFFERENTIATION

Algorithmic differentiation is a set of programming techniques first introduced in the early 1960s aimed at accurately and efficiently computing 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. This makes it possible to generate automatically a computer program that evaluates efficiently and with machine-precision accuracy the derivatives of the function (Griewank (2000)).

What makes AD particularly attractive when compared to standard (eg, finite-difference) methods for the calculation of the derivatives is its computational efficiency. In fact, AD aims to exploit 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 the following, we will review these ideas in more detail. In particular, we will describe the two basic approaches to AD, the so-called tangent (or forward) and adjoint (or backward) modes. These differ in regard to 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. Griewank (2000) contains a complete introduction to AD. Here, we will only recall the main results in order to clarify how this technique is beneficial in the implementation of the pathwise derivative method. We will begin by stating the results regarding the computational efficiency of the two modes of AD, and we will justify them by discussing a toy example in detail.

3.1 Computational complexity of the tangent and adjoint modes of algorithmic differentiation

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

$$(y_1, \dots, y_m)^{\mathrm{T}} = F(x_1, \dots, x_n) \tag{3.1}$$

In its simplest form, AD aims to produce coding evaluating the sensitivities of the outputs of the original program with respect to its inputs, ie, to calculate the Jacobian of the function F:

$$J_{ij} = \frac{\partial F_i(x)}{\partial x_j} \tag{3.2}$$

with $F_i(x) = y_i$.

The tangent mode of AD allows the calculation of the function F and of its Jacobian at a cost – relative to that for F – that can be shown, under a standard computational complexity model (Griewank (2000)), to be bounded by a small constant, ω_T, times the number of independent variables, namely:

$$\frac{\mathrm{Cost}[F\,\&\,J]}{\mathrm{Cost}[F]} \leqslant \omega_T n \tag{3.3}$$

The value of the constant ω_T can be also bounded using a model of the relative cost of algebric operations, nonlinear unary functions and memory access. This analysis gives $\omega_T \in [2, \frac{5}{2}]$ (Griewank (2000)).

The form of the result (3.3) 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. As we will illustrate in the next section, the tangent mode avoids repeating the calculations of quantities that are left unchanged by the perturbations of the different inputs, and it is therefore generally more efficient than bumping.

Consistent with Equation (3.3), 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, ie, a single column of the Jacobian (3.2), at a cost that 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:

$$\frac{\text{Cost}[F\,\&\,\mathcal{L}_c(J)]}{\text{Cost}[F]} \leqslant \omega_T \tag{3.4}$$

This makes the tangent mode particularly well suited for the calculation of (linear combinations of) the columns of the Jacobian matrix (3.2). Conversely, it is generally not the method of choice for the calculation of the gradients (ie, the rows of the Jacobian (3.2)) 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 (2000)):

$$\frac{\text{Cost}[F\,\&\,J]}{\text{Cost}[F]} \leqslant \omega_A m \tag{3.5}$$

with $\omega_A \in [3, 4]$, ie, AAD allows the calculation of the function F and of its Jacobian at a cost – relative to that for F – that 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 that 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.

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 as for a single row, namely:

$$\frac{\text{Cost}[F\,\&\,\mathcal{L}_r(J)]}{\text{Cost}[F]} \leqslant \omega_A \tag{3.6}$$

This clearly makes the adjoint mode particularly well suited for the calculation of (linear combinations of) the rows of the Jacobian matrix (3.2). 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 dependent ones ($m \ll n$).

We now provide justification of these results by discussing an explicit example in detail.

3.2 How algorithmic differentiation works: a simple example

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

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 2\log x_1x_2 + 2\sin x_1x_2 \\ 4\log^2 x_1x_2 + \cos x_1x_3 - 2x_3 - x_2 \end{pmatrix} \tag{3.7}$$

3.2.1 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, $w_1, \dots, w_N$, such that:

$$w_i = x_i, \qquad i = 1, \dots, n \tag{3.8}$$

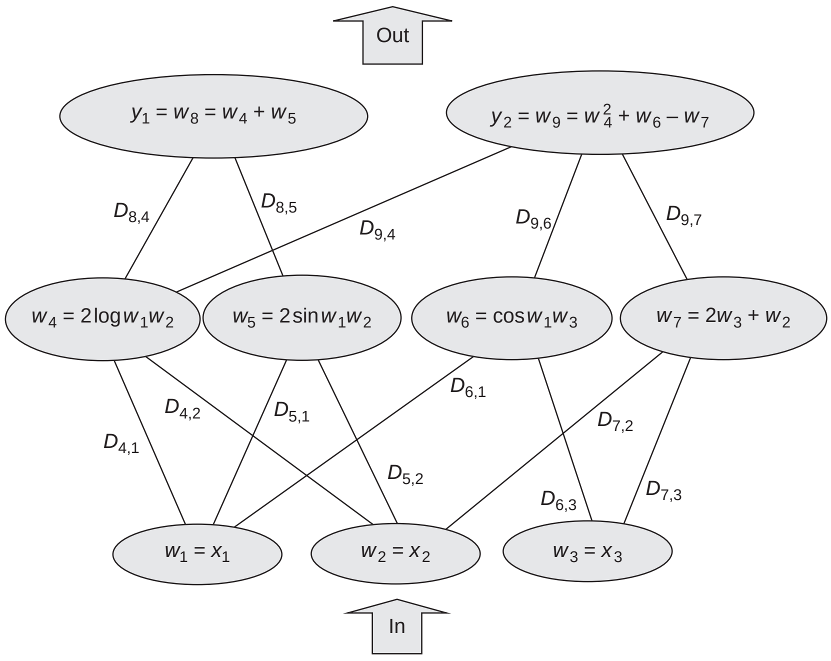
$$w_i = \Phi_i(\{w_j\}_{j \prec i}), \quad i = n+1, \dots, N \tag{3.9}$$

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, ie, $y_{i-N+m} = w_i$, $i = N - m + 1, \dots, N$. 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 (3.7), the internal calculations can be represented as follows:

$$\left.\begin{array}{l} w_1 = x_1, \qquad w_2 = x_2, \qquad w_3 = x_3 \\ \qquad\qquad\quad \downarrow \\ w_4 = \Phi_4(w_1, w_2) = 2\log w_1w_2 \\ w_5 = \Phi_5(w_1, w_2) = 2\sin w_1w_2 \\ w_6 = \Phi_6(w_1, w_3) = \cos w_1w_3 \\ w_7 = \Phi_7(w_2, w_3) = 2w_3 + w_2 \\ \qquad\qquad\quad \downarrow \\ y_1 = w_8 = \Phi_8(w_4, w_5) = w_4 + w_5 \\ y_2 = w_9 = \Phi_9(w_4, w_6, w_7) = w_4^2 + w_6 - w_7 \end{array}\right\} \tag{3.10}$$

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

FIGURE 1 Computational graph corresponding to the instructions (3.10) for the function in Equation (3.7).



to different criteria. Nevertheless, Equations (3.8) and (3.9) are an accurate representation of how the program is executed for a given value of the input vector x, ie, for a given instance of the internal controls. In this respect, AD aims to perform a piecewise differentiation of the program by reproducing the same controls in the differentiated code (Griewank (2000)).

The sequence of instructions (3.8) and (3.9) 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 Equation (3.7) the instructions (3.10) can be represented as in Figure 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:

$$D_{i,j} = \frac{\partial \Phi_i(\{w_k\}_{k \prec i})}{\partial w_j} \tag{3.11}$$

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

3.2.2 Tangent mode

Once the program implementing $F(x)$ is represented in terms of the instructions in (3.8) and (3.9) (or by a computational graph like that in Figure 1 on the facing page) the calculation of the gradient of each of its m components:

$$\nabla F_i(x) = (\partial_{x_1} F_i(x), \partial_{x_2} F_i(x), \dots, \partial_{x_n} F_i(x))^{\mathrm{T}} \tag{3.12}$$

simply involves the application of the chain rule of differentiation. In particular, by applying the rule starting from the independent variables, the tangent mode of AD is obtained:

$$\nabla w_i = e_i, \qquad i = 1, \dots, n \tag{3.13}$$

$$\nabla w_i = \sum_{j \prec i} D_{i,j} \nabla w_j, \quad i = n+1, \dots, N \tag{3.14}$$

where $e_1, e_2, \dots, e_n$ are the vectors of the canonical basis in $\mathbb{R}^n$, and $D_{i,j}$ are the local derivatives (3.11). For the example in Equation (3.7) this gives, for instance:

$$\begin{gathered}
\nabla w_1 = (1,0,0)^{\mathrm{T}}, \quad \nabla w_2 = (0,1,0)^{\mathrm{T}}, \quad \nabla w_3 = (0,0,1)^{\mathrm{T}} \\
\downarrow \\
D_{4,1} = 2w_2/(w_1 w_2), \quad D_{4,2} = 2w_1/(w_1 w_2) \\
\nabla w_4 = D_{4,1} \nabla w_1 + D_{4,2} \nabla w_2 \\
D_{5,1} = 2w_2 \cos w_1 w_2, \quad D_{5,2} = 2w_1 \cos w_1 w_2 \\
\nabla w_5 = D_{5,1} \nabla w_1 + D_{5,2} \nabla w_2 \\
D_{6,1} = -w_3 \sin w_1 w_3, \quad D_{6,3} = -w_1 \sin w_1 w_3 \\
\nabla w_6 = D_{6,1} \nabla w_1 + D_{6,3} \nabla w_3 \\
D_{7,2} = 1, \quad D_{7,3} = 2 \\
\nabla w_7 = D_{7,2} \quad \nabla w_2 + D_{7,3} \nabla w_3 \\
\downarrow \\
D_{8,4} = 1, \quad D_{8,5} = 1 \\
\nabla y_1 = \nabla w_8 = D_{8,4} \nabla w_4 + D_{8,5} \nabla w_5 \\
D_{9,4} = 2w_4, \quad D_{9,6} = 1, \quad D_{9,7} = -1 \\
\nabla y_2 = \nabla w_9 = D_{9,4} \nabla w_4 + D_{9,6} \nabla w_6 + D_{9,7} \nabla w_7
\end{gathered}$$

This leads to:

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

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

In the relations above, each component of the gradient is propagated 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 propagation in the tangent mode is more conveniently expressed by replacing the vectors ∇w_i with the scalars:

$$\dot{w}_i = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} \tag{3.15}$$

also known as tangents. Here λ is a vector in $\mathbb{R}^n$ specifying the chosen linear combination of columns of the Jacobian. Indeed, in this notation, the propagation of the chain rule (3.13) and (3.14) becomes:

$$\dot{w}_i = \lambda_i, \qquad i = 1, \dots, n \tag{3.16}$$

$$\dot{w}_i = \sum_{j \prec i} D_{i,j}\, \dot{w}_j, \quad i = n+1, \dots, N \tag{3.17}$$

At the end of the propagation we therefore find $\dot{w}_i$, $i = N - m + 1, \dots, N$:

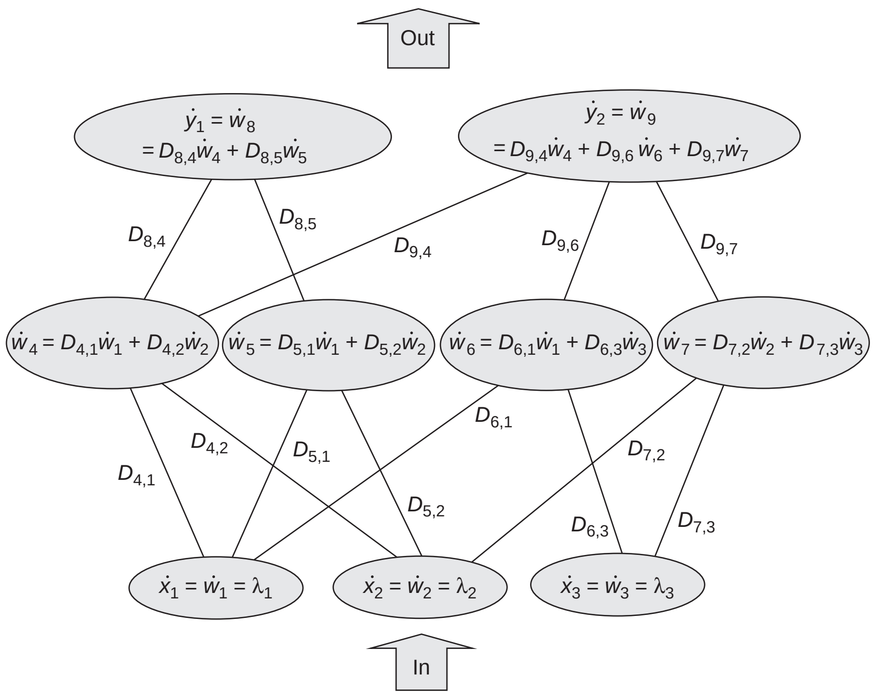
$$\dot{w}_i = \dot{y}_{i-N+m} = \sum_{j=1}^{n} \lambda_j \frac{\partial w_i}{\partial x_j} = \sum_{j=1}^{n} \lambda_j \frac{\partial y_{i-N+m}}{\partial x_j} \tag{3.18}$$

ie, a linear combination of the columns of the Jacobian.

As illustrated in Figure 2 on the facing page, the propagation of the chain rule (3.16) and (3.17) allows us to associate the tangent of the corresponding internal variable, say $\dot{w}_i$, with each node of the computational graph. This can be calculated as a weighted average of the tangents of the variables preceding it on the graph (ie, 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, ie, in the same direction followed in the evaluation of the original function, or forward. The propagation of the tangents can in fact proceed instruction by instruction, at the same time as the function is evaluated.

It is easy to realize that the cost for the propagation of the chain rule (3.16) and (3.17), for a given linear combination of the columns of the Jacobian, is of the same order as the cost of evaluating the function F itself. Hence, for the simple example considered here, Equation (3.4) 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 it is necessary to repeat $n = 3$ times the calculation of the computational graph in Figure 2 on the facing page, eg, by setting λ equal to each vector of the canonical basis in $\mathbb{R}^3$. As a result, the

FIGURE 2 Computational graph for the tangent mode differentiation of the function in Equation (3.7).


Out
$\dot{y}_1 = \dot{w}_8 = D_{8,4}\dot{w}_4 + D_{8,5}\dot{w}_5$
$\dot{y}_2 = \dot{w}_9 = D_{9,4}\dot{w}_4 + D_{9,6}\dot{w}_6 + D_{9,7}\dot{w}_7$
$D_{8,4}$
$D_{8,5}$
$D_{9,4}$
$D_{9,6}$
$D_{9,7}$
$\dot{w}_4 = D_{4,1}\dot{w}_1 + D_{4,2}\dot{w}_2$
$\dot{w}_5 = D_{5,1}\dot{w}_1 + D_{5,2}\dot{w}_2$
$\dot{w}_6 = D_{6,1}\dot{w}_1 + D_{6,3}\dot{w}_3$
$\dot{w}_7 = D_{7,2}\dot{w}_2 + D_{7,3}\dot{w}_3$
$D_{4,1}$
$D_{4,2}$
$D_{5,1}$
$D_{5,2}$
$D_{6,1}$
$D_{6,3}$
$D_{7,2}$
$D_{7,3}$
$\dot{x}_1 = \dot{w}_1 = \lambda_1$
$\dot{x}_2 = \dot{w}_2 = \lambda_2$
$\dot{x}_3 = \dot{w}_3 = \lambda_3$
In

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 Equation (3.3).

Finally, we remark that, by simultaneously carrying out the calculation of all the components of the gradient (or, more generally, of a set of n linear combinations of columns of the Jacobian) the calculation can be optimized by reusing a certain amount of computations (eg, the arc derivatives). This leads to a more efficient implementation also known as tangent multimode. Although the computational cost for the tangent multimode remains of the form (3.3) and (3.4), the constant ω_T for these implementations is generally smaller than for the standard tangent mode. This will also be illustrated in Section 4.

3.2.3 *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 propagating 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:

$$\bar{w}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} \tag{3.19}$$

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

$$\bar{w}_i = \frac{\partial y}{\partial w_i} = \sum_{j \succ i} \frac{\partial y}{\partial w_j} \frac{\partial w_j}{\partial w_i} = \sum_{j \succ i} D_{j,i} \bar{w}_j \tag{3.20}$$

where the sum is over the indices $j \succ i$ such that w_j depends explicitly on w_i. At the end of the propagation we therefore find $\bar{w}_i$, $i = 1, \dots, n$:

$$\bar{w}_i = \bar{x}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial w_i} = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial x_i} \tag{3.21}$$

ie, a given linear combination of the rows of the Jacobian (3.2).

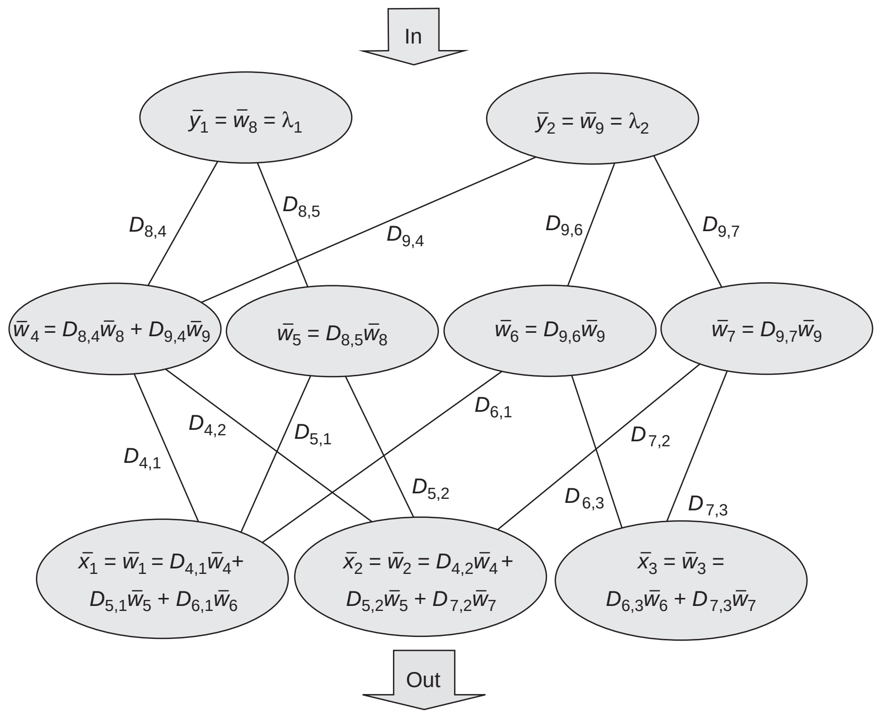
In particular, for the example in Equation (3.7) this gives:

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

It is immediately evident 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 Figure 3 on the facing page, Equation (3.20) 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 (ie, all the $\bar{w}_j$ such that $j \succ i$), with weights given by the local derivatives

FIGURE 3 Computational graph for the adjoint mode differentiation of the function in Equation (3.7).


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

associated with the respective arcs. As a result, the adjoints propagate through the computational graph from the dependent variables to the independent ones, ie, in the opposite direction to that of evaluation of the original function, or backward. The main consequence of this is that, in contrast to the tangent mode, the propagation of the adjoints cannot in general be 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 propagation of the adjoints can in general begin only after the construction of the computational graph has been completed, and the information on the values and dependences of the nodes on the graph (eg, the arc derivatives) has been appropriately stored.

It is easy to see that the cost for the propagation of the chain rule (3.20) for a given linear combination of the rows of the Jacobian is of the same order as the cost of evaluating the function F itself, in agreement with Equation (3.6). On the other hand, in order to get each row of the Jacobian, $m = 2$ times the calculation of the computational graph in Figure 3 on the preceding page has to be repeated, eg, 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 Equation (3.5).

3.3 Algorithmic differentiation tools

As illustrated in the previous examples, AD gives a clear set of prescriptions by which, given any computer function, the code implementing the tangent or adjoint mode for the calculation of its derivatives can be developed. This involves representing the computer function in terms of its computational graph, calculating the derivatives associated with each of the elementary arcs, and propagating either the tangents or the adjoints in the appropriate direction. This procedure, being mechanical in nature, can be automated. Indeed, 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 can be grouped into two main categories, namely source code transformation and operator overloading.[3]

Source code transformation tools are computer programs that take the source code of a function as input, 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 parser typically splits each instruction into the constituent unary or binary elementary operations for which the corresponding derivative functions are known.

On the other hand, the operator overloading approach exploits the flexibility of object-oriented languages in order to introduce new abstract data types suitable for representing 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.

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. Source code transformation involves more work but it is generally more transparent as it provides the code implementing the

[3] An excellent source of information in the field can be found at www.autodiff.org.

calculation of the derivatives as a sequence of elementary instructions. This simplicity facilitates compiler optimization, thus generally resulting in a faster execution. In the following section we consider examples of the source code transformation approach while discussing the calculation of the derivatives of the payout required for the implementation of the pathwise derivative method.

4 CALCULATING THE DERIVATIVES OF PAYOUT FUNCTIONS USING ALGORITHMIC DIFFERENTIATION

In this section we discuss a few examples illustrating how AD can be used to produce efficient coding for the calculation of the derivatives of the payout in (2.4).

Payouts of structured products are typically scalar functions of a large number of dependent variables. As a result, the adjoint mode of AD is generally best suited for the fast calculation of their derivatives. This will be clear from the examples discussed in Section 4.1. On the other hand, for vector-valued payouts, the adjoint and tangent modes of AD can often be combined in order to generate a highly efficient implementation, as discussed in Section 4.2. As specific examples, in the following we will consider European-style basket options and path-dependent “best of” Asian options.

4.1 Scalar payouts

4.1.1 Basket options

In Figure 4 on the next page we show the pseudocode for the payout function and its adjoint counterpart for a simple basket call option with payout:

$$P = P_r(X(T)) = \mathrm{e}^{-rT}\left(\sum_{i=1}^{N} w_i X_i(T) - K\right)^{+} \tag{4.1}$$

where $X(T) = (X_1(T), \dots, X_N(T))$ represent the value of a set of N underlying assets at time T, say a set of equity prices, w_i, $i = 1, \dots, n$, are the weights defining the composition of the basket, K is the strike price and r is the risk-free yield for the maturity considered. Here, for simplicity, we consider the case in which interest rates are deterministic. As a result, the interest rate r can be seen as a model parameter determined by the yield curve. For this example, we are interested in the calculation of the sensitivities with respect to r and the N components of the state vector X so that the other parameters (ie, strike and maturity) are seen here as dummy constants.

As illustrated in the pseudocode in part (a) of Figure 4 on the next page, the inputs of the computer function implementing the payout (4.1) are a scalar r and an N-dimensional vector X (we suppress the dependence on T from now on). The

FIGURE 4 Pseudocode for (a) the payout function and (b) its adjoint for the basket call option of Equation (4.1).

(a)

```
(P)= payout (r, X[N]){

  B = 0.0;
  for (i = 1 to N)
   B += w[i]* X[i];

  x = B - K;
  D = exp(-r * T);
  P = D * max(x, 0.0);
};
```

(b)

```
(P, r_b, X_b[N]) = payout_b(r, X[N], P_b){

                              // Forward sweep
  B = 0.0;
  for (i = 0 to N)
   B += w[i] * X[i];

  x = B - K;
  D = exp(-r * T);
  P = D * max(x, 0.0);
                              // Backward sweep
  D_b = max(x, 0.0) * P_b;

  x_b = 0.0;
  if (x > 0)
   x_b = D * P_b;

  r_b = - D * T * D_b;
  B_b = x_b;

  for (i = 0 to N)
   X_b[i] = w[i] * B_b;
};
```

output is a scalar P. On the other hand, the adjoint of the payout function, shown in part (b) of Figure 4, is of the form:

$$(P, \bar{r}, \bar{X}) = \bar{P}_r(X, \bar{P}) \tag{4.2}$$

ie, it has the scaling factor $\bar{P}$ as an additional scalar input, and the adjoints:

$$\bar{r} = \frac{\partial P_r(X)}{\partial r}\bar{P} \tag{4.3}$$

$$\bar{X}_i = \frac{\partial P_r(X)}{\partial X_i}\bar{P} \tag{4.4}$$

for $i = 1, \dots, N$ as additional outputs. In this and in the following figures we use the suffixes "_b" and "_d" to represent in the pseudocodes the "bar" and "dot" notations for adjoint and tangent quantities, respectively.

As discussed in Section 3, the adjoint payout function typically contains an initial forward sweep. This replicates the original payout script and evaluates the first output P of the function. The forward sweep is also used to keep a record of all the information necessary to calculate the arc derivatives (Section 3.2.1) that cannot be recovered efficiently going backwards on the computational graph. However, in this simple example, as shown in Figure 4 on the facing page, no information needs to be stored during the forward sweep, and the latter is just an exact replica of the original payout code.

Once the forward sweep is completed, the backward sweep propagates the adjoint quantities, reversing the order of the computations with respect to the original function. In the specific example, first the reverse sweep computes the adjoint counterpart of the very last instruction of the forward sweep:

$$P(D,x) = D(x)^+ \tag{4.5}$$

as seen as a function of the intermediate variables D and x. The adjoints of D and x are simply:

$$\bar{D} = \frac{\partial P(D,x)}{\partial D}\bar{P} = (x)^+\bar{P} \tag{4.6}$$

and:

$$\bar{x} = \frac{\partial P(D,x)}{\partial x}\bar{P} = D\vartheta(x)\bar{P} \tag{4.7}$$

where $\vartheta(x)$ is the Heaviside function. Then, taking the adjoint of the function $D = \exp(-rT)$ with constant T gives:

$$\bar{r} = \frac{\partial D(r)}{\partial r}\bar{D} = -DT\bar{D} \tag{4.8}$$

Finally, the adjoint of the instructions $x = x(B) = B - K$ (with K constant) and:

$$B = B(X) = \sum_{i=1}^{N} w_i X_i(T)$$

are computed in turn. Respectively, these are:

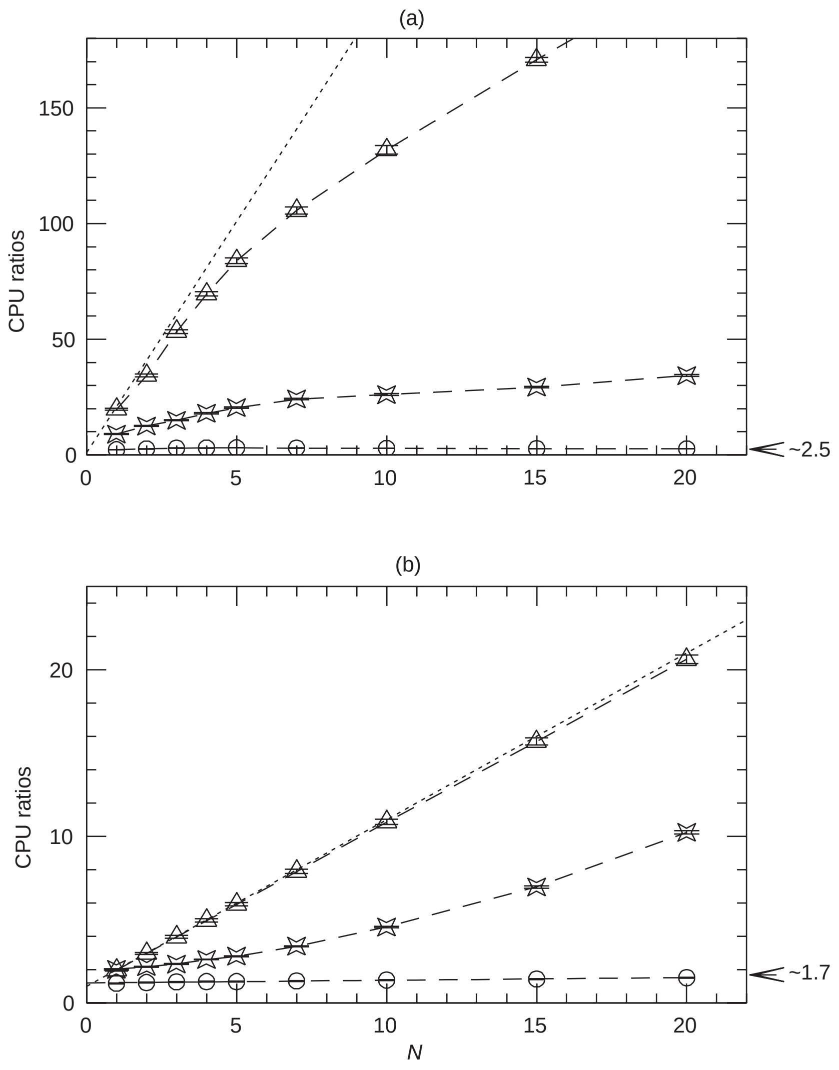
$$\bar{B} = \frac{\partial x(B)}{\partial B}\bar{x} = \bar{x} \tag{4.9}$$

and:

$$\bar{X}_i = \frac{\partial B(X)}{\partial X_i}\bar{B} = w_i\bar{B} \tag{4.10}$$

for $i = 1, \dots, N$. It is easy to recognize that the quantities $\bar{r}$ and $\bar{X}$ constructed this way represent the adjoints in Equations (4.3) and (4.4), respectively. For $\bar{P} = 1$, these clearly give the gradient of the payout function (4.1).

FIGURE 5 Ratios of the processing time required for the calculation of the value and of all the derivatives of the payout function, and the processing time spent on the computation of its value alone, as functions of the number of assets, N.



(a) "Best of" Asian option (4.12) for $M = 20$ observation times. (b) Basket option (4.1). Symbols: triangles, tangent; stars, tangent multimode; circles, adjoint. The dotted line represents the estimated CPU ratios associated with one-sided finite differences. The other lines are guides for the eye.

FIGURE 6 Pseudocode for the tangent payout function for the basket call option of Equation (4.1).

```
(P, P_d)= payout_d(r, X[N], r_d, X_d[N]){

  B = 0.0;
  for (i = 1 to N) {
   B += w[i]*X[i];
   B_d += w[i]*X_d[i];
  }

  x = B - K;
  x_d = B_d;

  D = exp(-r * T);
  D_d = -T * D * r_d;

  P = D * max(x, 0.0);
  P_d = 0;
  if(x > 0)
   P_d = D_d*x + D*x_d;

};
```

As mentioned in Section 3, AD cannot generally be expected to provide any meaningful results for those values of the input variables for which the function is not differentiable. This is the case, for instance, when the payout function involves the maximum function as in the example above. In this case, the derivatives for the set of inputs such that $B(X) = K$ are not defined, and are returned instead as zero. However, in the context of the pathwise derivative method, this does not create any practical difficulty as the set of points corresponding to these singularity constitutes a zero probability subset of the sample space in Equation (2.1). In any case, it is also generally possible to smooth out such singularities along the lines of the discussion in Section 4.3.

By inspecting the structure of the adjoint payout, it appears clear that its computational cost is just a small multiple (of order two) of the cost of evaluating the original payout. Indeed, it is easy to realize that the cost of the forward and backward sweeps are roughly the same, thus making the cost of calculating the complete gradient of the payout roughly twice the cost of evaluating the payout alone. In particular, the ratio of the processing time spent in the calculation of the adjoint payout and the processing time spent in the original payout function is independent of the number of inputs, in agreement with Equation (3.6).

The remarkable efficiency of the adjoint payout function is clearly illustrated in part (b) of Figure 5 on the facing page where the processing time ratio is plotted as a function of the number of assets N: the calculation of $N_{\mathrm{d}} = N + 1$ derivatives of the payout (one for each asset plus the derivative with respect to r) requires an

FIGURE 7 Pseudocode for the tangent multimode payout function for the basket call option of Equation (4.1).

```
(P, P_d[Nd]) = payout_dv(r, X[N], r_d[Nd], X_d[N,Nd]){

  B = 0.0;
  for (id = 1 to Nd)
     B_d[id] = 0.0;

  for (i = 1 to N) {
    B += w[i]*X[i];
    for (j = 1 to Nd)
      B_d[j] += w[i]*X_d[i,j];
  }
  x = B - K;
  for (j = 1 to Nd)
    x_d[j] = B_d[j];

  D = exp(-r * T);
  for (j = 1 to Nd)
    D_d[j] = -T * D * r_d[j];

  P = D * max(x, 0.0);
  P_d = 0;
  if(x > 0){
     for (j = 1 to Nd)
        P_d[j] = D_d[j]*x + D*x_d[j];
  }
};
```

extra overhead of about 70% with respect to the calculation of the payout itself for any number of underlying assets N. This is in stark contrast to the relative cost of evaluating the gradient by means of one-sided finite differences or the tangent mode of AD (also shown in the same figure), both scaling linearly with N.

The pseudocodes for the tangent payout, in the standard and in the multimode implementation, are given in Figure 6 on the preceding page and Figure 7, respectively. These are much more straightforward to understand because they correspond to a more natural application of the chain rule, as explained in Section 3.2.2. The tangent payout code must be run $N+1$ times, setting in turn one component of the tangent input vector $I = (\dot{r}, \dot{X})^{\mathrm{T}}$ to one and the remaining ones to zero. The tangent multimode payout needs instead to be run only once, and it is initialized by the set of $N_{\mathrm{d}} = N + 1$ tangent input vectors above. More precisely, the inputs of the tangent multimode payout are an N_{d}-dimensional vector $\dot{r}_j$, and the $N \times N_{\mathrm{d}}$ matrix $\dot{X}_{i,j}$, which can be chosen as:

$$\dot{r}_j = \delta_{j,1}, \qquad \dot{X}_{i,j} = \delta_{i,j-1} \tag{4.11}$$

for $i = 1, \dots, N$ and $j = 1, \dots, N_{\mathrm{d}}$.

FIGURE 8 Pseudocode for (a) the payout function and (b) its adjoint for the "best of" Asian option of Equation (4.12).

(a)

```
(P) = payout(X[N]){

 sum = 0.0;
 for(istep = 1 to Nstep){
  max_step = 0.0;

  for(i = 1 to Nass){
     ioff = istep + i *Nstep;
     branch_vector[ioff] = 0;

     if( X[ioff] > max_step ) {
         max_step = X[ioff]/X0[i];
         branch_vector[ioff] = 1;
     }
  }
  sum = sum + max_step/Nstep;
 }
 x = sum - K ;
 P = max(x, 0) ;
};
```

(b)

```
(P, X_b[N]) =  payout_b(P_b, X[N]) {

 (code as in payout)
                         // Backward Sweep
 x_b = 0.0;
 if (x > 0)
  x_b = P_b;

 sum_b = x_b;

 for(istep = Nstep to 1) {
   max_step_b = sum_b/Nstep;

   for(i = Nass to 1) {
     ioff = istep + i * Nstep;
     branch = branch_vector[ioff];

     if(branch == 1)
       X_b[ioff] = max_step_b/X0[i];
   }
  }
};
```

The vector of adjoints X_b[] is assumed initialized to zero. The forward sweep in the adjoint payout is omitted for brevity because its code is identical to that in part (a). Note that the array "branch_vector" can be omitted in the payout implementation, and serves a purpose only in the forward sweep.

As shown in Figure 5 on page 22, in both cases, the resulting cost of obtaining the gradient of the payout function is asymptotically proportional to the number of components in the basket. However, as anticipated in Section 3.2.2, the tangent multimode payout is significantly more efficient than the standard tangent mode because it avoids the multiple evaluation of the function value, and in general is able to reuse the value of the arc derivatives.

4.1.2 "Best of" Asian options

As a second example we consider a path-dependent option, namely a "best of" Asian option with (undiscounted) payout given by:

$$P(X) = (A(T_M) - K, 0)^+ \tag{4.12}$$

where $A(T_M) = \sum_{m=1}^{M} \chi(T_m)/M$, and $\chi(T_m)$ is the maximum return of the N underlying assets at time T_m, namely:

$$\chi(T_m) = \max_{i=1,\dots,N} \left[\frac{X_i(T_m)}{X_i(T_0)} \right] \tag{4.13}$$

where T_0 is a reference observation time. In this example, we are interested in the calculation of the derivatives of the payout function with respect to the $d = N \times M$ components of the state vector. The pseudocodes for the payout and its adjoint are shown in Figure 8 on the preceding page. Here $\chi(T_m)$ and $A(T_M)$ are represented by the variables "max_step" and "sum", respectively. As in the previous example, the initial part of the adjoint code – the forward sweep – essentially amounts to evaluating the payout function. However, in this case, some information needs to be stored during the forward sweep in order for the reverse sweep to be executed efficiently. This is contained in the array "branch_vector", tagging the asset with the largest return on each time step.

The backward sweep begins with the adjoint of the instructions:

$$P(x) = (x, 0)^+ \tag{4.14}$$

and:

$$x = x(A) = A(T_M) - K \tag{4.15}$$

These give, respectively:

$$\bar{x} = \vartheta(x)\bar{P} \tag{4.16}$$

and:

$$\bar{A}(T_M) = \bar{x} \tag{4.17}$$

Then, since the order of the calculations in the backward sweep is reversed with respect to that of the payout function, the loops on the number of assets and on the time steps are executed in opposite directions with respect to the original ones.[4] In particular, at each iteration of the loop on the time steps, the adjoint of $\chi(T_m)$:

$$\bar{\chi}(T_m) = \frac{\partial A(T_M)}{\partial \chi(T_m)} \bar{A}(T_M) = \frac{\bar{A}(T_M)}{M} \tag{4.18}$$

[4] However, note that in this example this is important only for the loop on the time steps, as the order is unimportant in the loop on the assets.

is calculated. In turn, for each asset, the adjoints:

$$\bar{X}_i(T_m) = \frac{\partial \chi(T_m)}{\partial X_i(T_m)} \bar{\chi}(T_m) = \frac{\delta_{i,i^\star(m)}}{X_{i^\star(m)}(T_0)} \bar{\chi}(T_m) \tag{4.19}$$

(where $i^\star(m)$ is the index corresponding to the asset with the highest return at time T_m) are finally computed. Note that, in order to perform the calculation of the derivative $\partial \chi(T_m)/\partial X_i(T_m)$ above, the algorithm needs to know which of the N assets assumed the maximum return at time T_m. This is precisely the content of the array "branch_vector" constructed during the forward sweep. Without this information, the backward sweep would have to perform on each time step an additional loop on the number of assets. This would result in a computational cost $O(N)$ higher.

As observed in the previous example, a simple inspection of the adjoint code reveals that its computational cost is a small multiple of the cost of the original payout. As a result, the relative cost of evaluating the adjoint payout with respect to evaluating the payout alone is independent of the number of input variables $d = M \times N$. This is clearly illustrated in part (a) of Figure 5 on page 22, showing that the calculation of the payout and of its derivatives with AAD is at most approximately 2.5 times more expensive than the original payout evaluation, for any number of underlying assets. Similar results can be obtained by keeping the number of assets constant and increasing the number of observations in time. As noted before, this is in contrast to the relative cost provided by the tangent mode of AD, scaling linearly with d (although with a smaller proportionality constant in the multimode implementation).[5]

4.2 Vector-valued payouts and the hybrid tangent–adjoint mode

In financial practice, it is not uncommon to use the same Monte Carlo simulation to evaluate simultaneously a portfolio of R contingent claims depending on a common pool of underlying assets. In these situations, the payouts are represented by vector-valued functions and each component, P_j, represents the value of the cashflows of one of the options in the portfolio.

The adjoint mode of AD is particularly well suited to the calculation of the sensitivities of the portfolio as a whole. Indeed, as discussed in Section 3.2.3, the adjoint mode of AD provides in general the most efficient way to evaluate the linear combination:[6]

$$\bar{X}_i = \sum_{j=1}^{R} \frac{\partial P_j}{\partial X_i} \bar{P}_j \tag{4.20}$$

[5] The pseudocodes for the tangent payouts for this example are omitted for brevity and are available upon request.

[6] In this discussion, we suppress for simplicity the explicit dependence of the payout on the parameter θ.

As a result, by choosing each weight $\bar{P}_j$ equal to the notional amount of the jth option, the value of the whole portfolio can be efficiently evaluated. These can be used in turn to construct the pathwise derivative estimator (2.4) with the substitution:

$$P(X) \to \sum_{j=1}^{R} \bar{P}_j P_j(X)$$

providing the aggregated sensitivities of the portfolio.

In contrast, in those cases in which the risk associated with each option in the portfolio is needed, the pathwise derivative method requires the calculation of the full Jacobian of the payout function, $J_{ij} = \partial P_j(X)/\partial X_i$. As discussed in Section 3, the cost of calculating this Jacobian with AAD, divided by the cost of evaluating the payout, scales linearly with the number of options in the portfolio R. Conversely, in the tangent mode the same ratio scales linearly with the dimension of the state vector $d = M \times N$. As a result, the adjoint mode can be expected to be more efficient than the tangent mode when the number of options in the portfolio is smaller than the dimension of the state vector. This depends on the specific pricing problem at hand.

Nevertheless, in some common cases it is possible to combine the tangent and adjoint modes to increase the efficiency of the calculation. This is best illustrated with a simple example. Imagine that a payout function can be decomposed as:

$$(P_1, \dots, P_R) = P(X) = F^{\mathrm{E}}(F^{\mathrm{I}}(X)) \tag{4.21}$$

with:

$$(Y_1, \dots, Y_J) = F^{\mathrm{I}}(X) \tag{4.22}$$

and:

$$(P_1, \dots, P_R) = F^{\mathrm{E}}(Y) \tag{4.23}$$

where $Y = (Y_1, \dots, Y_J)$, $J \ll R$ and $J \ll d$. Then a potentially efficient approach to calculating the Jacobian of $P(X)$ involves the following steps:

(1) Evaluate the forward sweep as in the standard adjoint mode. In particular, store the value of the intermediate variables Y.

(2) Apply the tangent mode of AD to get the matrix of derivatives $\partial F_j^{\mathrm{E}}(Y)/\partial Y_s$ with $j = 1, \dots, R$ and $s = 1, \dots, J$. The resulting cost, relative to that of evaluating F^{E}, scales linearly with J.

(3) For each $s = 1, \dots, J$, evaluate $\bar{X}_{s,i} = \partial F_s^{\mathrm{I}}(X)/\partial X_i$, $i = 1, \dots, d$, using the adjoint mode of AD at a cost that is a small multiple of that of evaluating $F^{\mathrm{I}}(X)$. The resulting cost of obtaining the matrix $\bar{X}_{s,i}$, relative to that of evaluating F^{I}, scales linearly with J.

(4) Construct the Jacobian:

$$\frac{\partial P_j}{\partial X_i} = \sum_{s=1}^{J} \frac{\partial F_j^{\mathrm{E}}(Y)}{\partial Y_s} \bar{X}_{s,i} \tag{4.24}$$

It is easy to realize that the cost of the Jacobian $J_{i,j}$ divided by that of evaluating the payout P scales linearly with J instead of R or d. This can result in significant savings with respect to the standard tangent or adjoint modes.

An elementary illustration of this situation is the generalization of the payouts considered in the previous sections for different values of the strike price, say $K_1, \dots, K_R$. In these cases, the payout functions can be decomposed as:

$$P_j(Y) = (Y - K_j)^+ \tag{4.25}$$

where Y is a scalar given by $\sum_{i=1}^{N} w_i X_i(T)$ for the basket option, and by $A(T_M)$ for the "best of" Asian contract. As a result, the Jacobian of the payout function reads:

$$\frac{\partial P_j(Y)}{\partial Y} \frac{\partial Y}{\partial X_i} \tag{4.26}$$

for $i = 1, \dots, d$ ($d = N$ and $d = N \times M$ for the basket and "best of" Asian options, respectively) and $j = 1, \dots, R$. The gradient $\partial P_j(Y)/\partial Y$ can be evaluated in general with the tangent mode at a cost that is a small multiple of the cost of evaluating Equation (4.25) above. In this simple example this gives:

$$\frac{\partial P_j(Y)}{\partial Y} = \vartheta(Y - K_j) \tag{4.27}$$

On the other hand, the quantities $\partial Y/\partial X_i$ are common to all the components of the payout function and need to be evaluated only once for all the options in the portfolio. In addition, since Y is a scalar, this can be done efficiently with the adjoint mode following exactly the same steps illustrated in Figure 4 on page 20 and Figure 8 on page 25. The resulting computational cost of the Jacobian $J_{ij} = \partial P_j(X)/\partial X_i$ is a small multiple of the cost of evaluating the payout itself. Remarkably, this multiple is independent of both the number of options in the portfolio and the dimension of the state vector.

4.3 Discontinuous payouts

As recalled in Section 2, the pathwise derivative method can be applied under a specific regularity condition requiring the payout function to be Lipschitz continuous (Glasserman (2004)). 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, eg, in the form of digital features, random variables counting discrete events, or barriers.

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 the singularities they contain. Clearly 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 more than acceptable in financial practice (see also Capriotti and Giles (2011)).

5 PATHWISE DERIVATIVE METHOD WITH ADJOINT PAYOUTS

As illustrated in the previous sections, AD, especially in the adjoint mode, allows an efficient calculation of the derivatives of the payout function, thus solving the main implementation difficulty of the pathwise derivative method.

Additional speed-ups can also be obtained through an efficient calculation of the tangent state vector. A remarkable example has been discussed, for instance, in Giles and Glasserman (2006) for the case of European options in a diffusive setting. Although this approach can in principle be generalized to path-dependent options, this may result in a degradation of its performance, with a computational cost roughly scaling with the number of observation times. Nonetheless, Equation (2.4) can be interpreted as a linear combination of the columns of the Jacobian defined by the tangent state vector Equation (2.5), with weights given by the derivatives of the payout function. As a result, it can be shown that the AAD paradigm can be used in general to obtain a highly efficient implementation of the complete pathwise derivative estimator. More precisely, AAD ensures that any number of sensitivities can be evaluated at a total computational cost that is bounded by roughly $\omega_A \simeq 4$ times the cost of evaluating the option value, independent of the number of sensitivities. A complete discussion of the implementation details of the full AAD approach is given in a forthcoming companion paper (Capriotti and Giles (2011)).

In the following, we will concentrate on those common cases in which the calculation of the tangent state vector, and of the sum in Equation (2.4), can be efficiently implemented without making use of the full AAD approach (Capriotti and Giles (2010) and Capriotti and Giles (2011)). This is, for instance, the case when each model parameter θ_k affects only a limited number of components $d_\theta \ll d$ of the state vector. In these cases, the matrix representing the tangent state vector (2.5) can be put in (at least approximately) block diagonal form so that both the calculation of its nonzero entries, and of the sum in Equation (2.4), can be performed at a cost $O(d_\theta N_\theta)$, ie, a factor $d_\theta / d \ll 1$ smaller than in the general case.

In a diffusive setting the situation described above is realized when each underlying asset depends only on a limited subset of model parameters θ. In these cases, eg, commonly arising in equity or foreign exchange models, in order to calculate the tangent state vector (2.5) only one or a limited number of diffusive equations needs to be simulated for each sensitivity. In this case, it is easy to see that the computational cost of the pathwise derivative method with adjoint payouts is likely to become smaller than that associated with bumping as the number of assets increases.

5.1 Numerical examples

5.1.1 Basket options

As a first illustration, we discuss the call option on a basket of N assets (4.1) considered in Section 4.1.1. Here we restrict ourselves to a lognormal model of the form:

$$X_i(t) = X_i^0 \exp((r - \sigma_i^2/2)t + \sigma_i W_i(t)) \tag{5.1}$$

where X_i^0 and σ_i are the spot price and volatility of the ith asset, respectively, and $W_i(t)$ are standard Brownian motions such that $\mathbb{E}[\mathrm{d}W_i(t)\,\mathrm{d}W_j(t')] = \rho_{ij}\,\mathrm{d}t$, where ρ is a positive semidefinite $N \times N$ correlation matrix. In this case, it is easy to derive analytically the form of the tangent state vector (2.7) at any time t, eg, for delta and vega:

$$\Delta_i(t) \equiv \frac{\partial X_i(t)}{\partial X_i^0} = \frac{X_i(t)}{X_i^0} \tag{5.2}$$

$$\mathcal{V}_i(t) \equiv \frac{\partial X_i(t)}{\partial \sigma_i} = X_i(t)(-\sigma_i t + W_i(t)) \tag{5.3}$$

respectively.

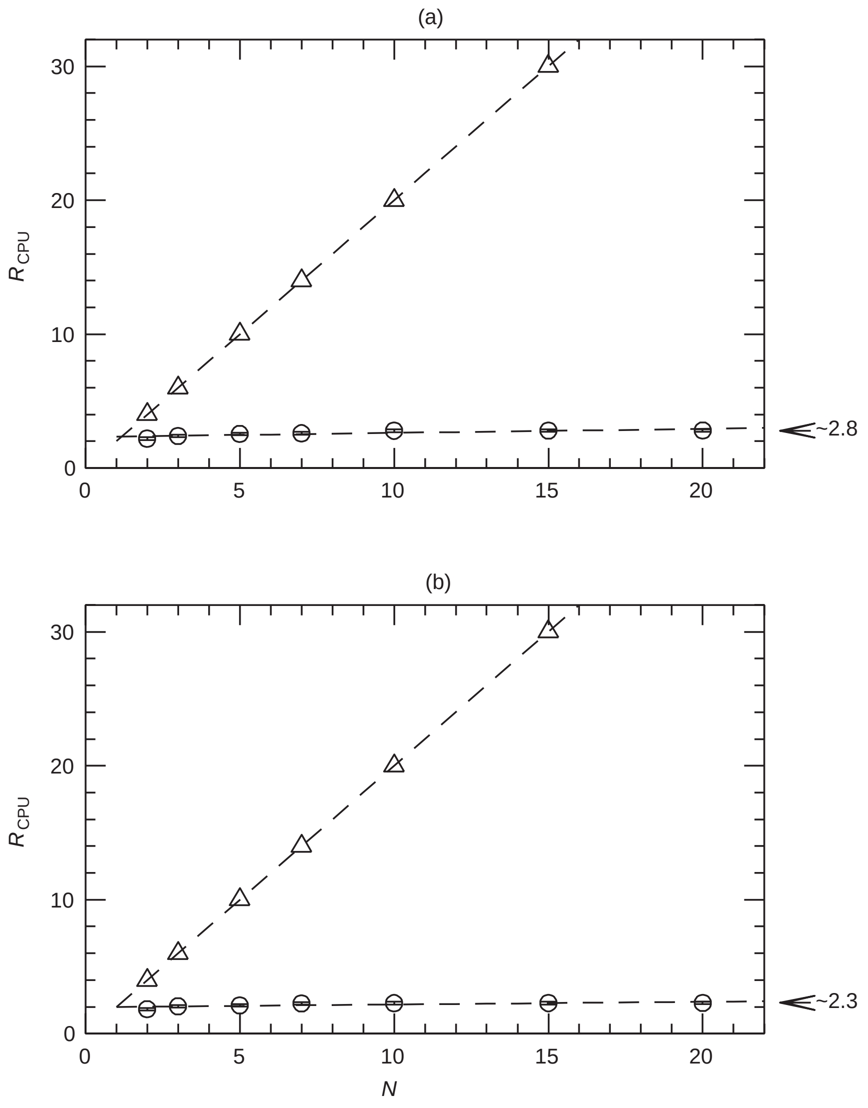
This example clearly falls under the situation discussed in the introduction to this section as each model parameter, eg, spot price or volatility, affects only a single underlying asset. As a result, the cost of the calculation of sensitivities by means of the pathwise derivative method with adjoint payouts is $O(N \times M)$ as it is the cost of calculating the value of the option, Equation (2.2). This means that all deltas and vegas for the basket call option (4.1) can be expected to be calculated at a cost that is a small multiple of the cost of calculating the value of the option, irrespective of the number of underlying assets in the basket.

This remarkable result is illustrated in the part (b) of Figure 9 on the next page, where we plot the processing time necessary to calculate the full delta and vega risk of the basket call option (4.1) divided by the time to calculate the value of the option, say:

$$R_{\mathrm{CPU}} = \frac{\mathrm{Cost[Value + Risk]}}{\mathrm{Cost[Value]}} \tag{5.4}$$

as a function of the number of underlying assets. As expected, the cost of evaluating the Greeks by means of the pathwise derivative method with adjoint payouts is a small

FIGURE 9 Processing time ratios (5.4) for the calculation of delta and vega risk by means of the pathwise derivative method with adjoint payouts (circles) as a function of the number of underlying assets.



(a) "Best of" Asian option Equation (4.12) for $M = 12$ observation times. (b) Basket option. The estimated processing time ratios for the calculation of the same sensitivities by means of bumping are also shown (triangles) for comparison pruposes. The lines are guides for the eye.

multiple of the cost of evaluating the value of the option itself. In this case the extra overhead is approximately 130% of the cost of calculating the option value. In contrast, the cost of estimating the same sensitivities by means of bumping grows linearly with the number of assets. As a result, the pathwise derivative method with adjoint payouts is computationally more efficient than bumping for any number of underlying assets, with savings larger than one order of magnitude already for medium-sized ($N \sim 10$) baskets.

5.1.2 "Best of" Asian options

As a second example, we consider the "best of" Asian option (4.12) discussed in Section 4.1.2. Here, we will report results for local volatility (Hull (2002)) diffusions simulated by means of a standard Euler discretization. In particular, the drift and volatility functions are assumed to take the form $\mu_i = r(t)X_i(t)$ and $\sigma_i = \sigma_i^F(X_i, t)X_i(t)$, where $r(t)$ is the deterministic instantaneous short rate, and $\sigma_i^F(x, t)$ is the instantaneous volatility function for the ith asset at time t. For the sake of this discussion, as is often the case in practice, we will regard the instantaneous volatility function as depending parametrically on the level of the "at-the-money" volatility $\sigma_i^{\text{ATM}}(t)$, namely $\sigma_i^F(x, t) = \sigma_i^F(x, \sigma_i^{\text{ATM}}(t))$. In the discussion below we will consider standard delta and vega with respect to a single perturbation of the at-the-money volatilities, namely $\sigma_i^{\text{ATM}}(t) \to \sigma_i^{\text{ATM}}(t) + \delta\sigma_i^{\text{ATM}}(t)$. Note that this example, although more complex than the previous one, still falls in the category of problems in which the calculation of each sensitivity involves the perturbation of the trajectories of just a single underlying asset.

In part (a) of Figure 9 on the facing page we plot the processing time ratio (5.4) for the calculation of delta and vega for $M = 12$ observation times, and for different numbers N of underlying assets. As observed before for the basket option, the relative cost associated with the pathwise derivative method with adjoint payouts is independent of the number of assets. Remarkably, even for this more complicated example the extra overhead associated with the calculation of the risk is limited to approximately 180% of the cost of calculating the option value. As a result, even for a single asset $N = 1$, the pathwise derivative method with adjoint payouts is computationally more efficient than bumping with savings that grow linearly to over one order of magnitude already for a relatively small number ($N \sim 12$) of underlying assets.

6 CONCLUSIONS

In this paper we have shown how algorithmic differentiation can be used to produce efficient coding for the calculation of the derivatives of the payout function, thus dealing with one of the main performance bottlenecks of the pathwise derivative method. With a variety of examples we demonstrated that the pathwise derivative method combined with algorithmic differentiation – especially in the adjoint mode – may provide speed-ups of several orders of magnitude with respect to standard methods. We also showed how the tangent mode can be combined with the adjoint mode for extra performance for vectorized payouts.

In addition to Monte Carlo methods, the efficient calculation of the derivatives of the payout function by means of algorithmic differentiation also has a natural application in the implementation of adjoint techniques in partial differential equation applications (Prideaux (2009)).

In forthcoming companion papers (Capriotti and Giles (2010) and Capriotti and Giles (2011)) we will build on these ideas to illustrate how AAD can be used for a highly efficient implementation of the complete pathwise derivative estimator. In particular, we will show how adjoint implementations like those of Giles and Glasserman (2006) and Leclerc *et al* (2009) can be seen as instances of AAD. This allows the fast calculation of the Greeks of complex path-dependent structured derivatives with virtually any model used in computational finance. In particular, we will discuss a variety of examples, including commodity structured products, correlation models for credit derivatives, the application to Bermudan options, and second-order risk. These results will demonstrate how algorithmic differentiation provides an extremely general framework for the calculation of risk in financial engineering.

REFERENCES

Broadie, M., and Glasserman, P. (1996). Estimating security price derivatives using simulation. *Management Science* **42**, 269–285.

Capriotti, L., and Giles, M. (2010). Fast correlation Greeks by adjoint algorithmic differentiation. *Risk* **29**, 77–83.

Capriotti, L., and Giles, M. (2011). Algorithmic differentiation: adjoint Greeks made easy. In preparation.

Chen, J., and Fu, M. C. (2001). Efficient sensitivity analysis of mortgage backed securities. *12th Annual Derivatives Securities Conference*, unpublished.

Giles, M. (2007). Monte Carlo evaluation of sensitivities in computational finance. *Proceedings of HERCMA Conference, Macedonia, September*.

Giles, M., and Glasserman, P. (2006). Smoking adjoints: fast Monte Carlo Greeks. *Risk* **19**, 88–92.

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

Glasserman, P., and Zhao, X. (1999). Fast Greeks by simulation in forward LIBOR models. *The Journal of Computational Finance* **3**, 5–39.

Griewank, A. (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Harrison, J., and Kreps, D. (1979). Martingales and arbitrage in multiperiod securities markets. *Journal of Economic Theory* **20**, 381–408.

Hull, J. C. (2002). *Options, Futures and Other Derivatives*. Prentice Hall, Upper Saddle River, NJ.

Kallenberg, O. (1997). *Foundations of Modern Probability*. Springer.

Kunita, H. (1990). *Stochastic Flows and Stochastic Differential Equations*. Cambridge University Press.

Leclerc, M., Liang, Q., and Schneider, I. (2009). Fast Monte Carlo Bermudan Greeks. *Risk* **22**, 84–88.

Prideaux, A. (2009). PhD Thesis, Oxford University.

Protter, P. (1997). *Stochastic Integration and Differential Equations*. Springer.