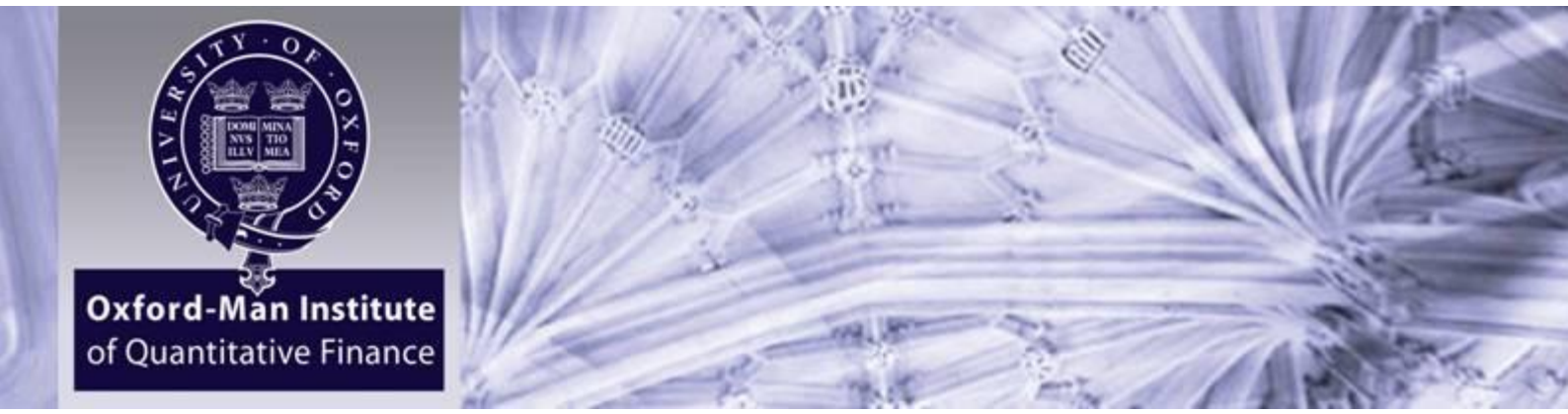

Monte Carlo evaluation of sensitives in computational finance

Michael B Giles

The Oxford-Man Institute, University of Oxford
Working paper, OMI08/07

Monte Carlo evaluation of sensitivities in computational finance

M.B. Giles

Abstract

In computational finance, Monte Carlo simulation is used to compute the correct prices for financial options. More important, however, is the ability to compute the so-called "Greeks", the first and second order derivatives of the prices with respect to input parameters such as the current asset price, interest rate and level of volatility.

This paper discusses the three main approaches to computing Greeks: finite difference, likelihood ratio method (LRM) and pathwise sensitivity calculation. The last of these has an adjoint implementation with a computational cost which is independent of the number of first derivatives to be calculated. We explain how the practical development of adjoint codes is greatly assisted by using Algorithmic Differentiation, and in particular discuss the performance achieved by the FADBAD++ software package which is based on templates and operator overloading within C++.

The pathwise approach is not applicable when the financial payoff function is not differentiable, and even when the payoff is differentiable, the use of scripting in real-world implementations means it can be very difficult in practice to evaluate the derivative of very complex financial products. A new idea is presented to address these limitations by combining the adjoint pathwise approach for the stochastic path evolution with LRM for the payoff evaluation.

I. Introduction

Monte Carlo simulation is the most popular approach in computational finance for determining the prices of financial options. This is partly due to its computational efficiency for "high-dimensional" problems involving multiple assets, interest rates or exchange rates, and partly due to its relative simplicity and the ease with which it can be parallelised across large compute clusters.

The accurate calculation of prices is only one objective of Monte Carlo simulation. Indeed, because the mathematical models are calibrated to actual prices observed in the marketplace, it can be argued that prices are largely determined by the market prices of a large range of frequently traded products. Where the Monte Carlo simulation plays a crucial role is in the calculation of the sensitivity of the prices to changes in various input parameters, such as the current asset price, interest rate and level of volatility. Both first and second order derivatives are essential for hedging and risk analysis, and even higher order derivatives are sometimes used They are known collectively as the "Greeks", as many of them have associated Greek letters; for example, Delta and Gamma are the first and second order derivatives with respect to the current asset price.

This paper discusses mathematical and computer science aspects of computing Greeks through Monte Carlo simulation. After a brief introduction to Monte Carlo simulation and the approximation of stochastic differential equations, the three main approaches to computing Greeks are presented: finite difference, likelihood ratio method (LRM) and pathwise sensitivity calculation. For further details on all three approaches, the interested reader is referred to the excellent book of Glasserman [14] which gives a comprehensive introduction to Monte Carlo methods for computational finance.

The last of the three methods, the pathwise sensitivity approach, leads very naturally to an adjoint implementation which makes it possible to compute the sensitivity to a large number of input parameters at a very low cost, little more than the cost of evaluating the price itself. The paper by Giles and Glasserman [12] which first introduced this technique for Monte Carlo simulations, generated considerable interest in the finance industry which faces the need to compute large numbers of sensitivities on a daily basis. This current paper is motivated by some of the feedback from the first paper, and in particular by comments on the difficulties involved in the development of adjoint codes, and the inherent limitations of the pathwise sensitivity approach.

M.B. Giles is Professor of Scientific Computing, Oxford University Computing Laboratory, Oxford, U.K., OX1 3QD. Email: giles@comlab.ox.ac.uk

The practical development of adjoint codes is based on ideas of Algorithmic Differentiation (AD) [16], and greatly assisted by using automated AD tools. After the first third of the paper has set the scene by introducing the Monte Carlo method and adjoint calculation of Greeks, the second third of the paper reviews the ideas of AD, and discusses how it can be used to create adjoint code which may be used either to validate hand-written adjoint code, or directly in parts of the process which are not computationally expensive. Results are presented for the performance using the FADBAD++ software package, which is based on templates and operator overloading within C++, and the TAC++ package, which uses the alternative approach of source code transformation.

The pathwise approach is not applicable when the financial payoff function is not differentiable. Even when the payoff is differentiable, the use of scripting in real-world implementations means it can be very difficult in practice to evaluate the derivative of very complex financial products. The final third of the paper addresses these limitations by presenting a new idea which combines the adjoint pathwise approach for the stochastic path evolution with LRM for the payoff evaluation.

II. An overview of Monte Carlo simulation

A. Stochastic differential equations

A stochastic differential equation (SDE) with general drift and volatility terms has the form

$$\mathrm{d}S(t) = a(S,t)\,\mathrm{d}t + b(S,t)\,\mathrm{d}W(t), \tag{1}$$

which is simply a shorthand for the more formal integral equation

$$S(t) = S(0) + \int_0^t a(S(t'),t')\;\mathrm{d}t' + \int_0^t b(S(t'),t')\;\mathrm{d}W(t'). \tag{2}$$

Here $S(t) \in \mathbb{R}^{d_1}$, and $W(t) \in \mathbb{R}^{d_2}$ is a random Wiener variable with the defining properties that for any $q<r<s<t$, each component of $W(t)-W(s)$ is Normally distributed with mean 0 and variance $t-s$, independent of $W(r)-W(q)$ and the other components of $W(t)-W(s)$. Because of the special nature of $W(t)$, the rightmost integral in (2) is not a standard Riemann or Lebesgue integral, but is instead an Itô integral, defined through a certain limiting procedure [19]. In many applications, $S(t)$ and $W(t)$ have the same dimensions, but in some cases $d_1 > d_2$.

The stochastic term $b(S,t)\,\mathrm{d}W$ models the uncertain, unpredictable events which influence asset prices, interest rates, exchange rates and other financial variables. Financial modelling aims to compute the expected value of quantities which depend on $S(t)$, averaging over the different possible paths the future may take. Monte Carlo simulation estimates this expectation by simulating a finite number of future paths, and averaging over that finite set.

In simple cases, the SDE may be explicitly integrated. For example, the scalar SDE which underlies the famous Black-Scholes model [3] is geometric Brownian motion

$$\mathrm{d}S(t) = r\,S\,\mathrm{d}t + \sigma\,S\,\mathrm{d}W(t),$$

where r is the constant risk-free interest rate and σ is a constant volatility. Using Itô calculus [19], the corresponding SDE for $X \equiv \log S$ is

$$\mathrm{d}X(t) = (r-\tfrac{1}{2}\sigma^2)\,\mathrm{d}t + \sigma\,\mathrm{d}W(t),$$

which may be integrated subject to initial conditions $X(0)=X_0=\log S_0$ to give

$$X(T) = X_0 + (r-\tfrac{1}{2}\sigma^2)\,T + \sigma\,W(T),$$

and hence

$$S(T) = S_0 \exp\left((r-\tfrac{1}{2}\sigma^2)\,T + \sigma\,W(T)\right).$$

In this case, the expected value of some financial payoff $P = f(S(T))$ can be expressed as

$$V \equiv \mathbb{E}\left[f(S(T))\right] = \int f(S(T))\; p_W(W)\; \mathrm{d}W, \tag{3}$$

where

$$p_W(W) = \frac{1}{\sqrt{2\pi\, T}}\; \exp\left(-\frac{W^2}{2T}\right)$$

is the probability density function for $W(T)$. Performing a change of variables, the expectation can also be expressed as

$$V \equiv \mathbb{E}\left[f(S(T))\right] = \int f(S)\; p_S(S)\; \mathrm{d}S, \tag{4}$$

where

$$p_S(S) \;=\; \left(\frac{\partial S}{\partial W}\right)^{-1} p_W \;=\; \frac{1}{S\sigma\sqrt{2\pi\, T}}\; \exp\left(-\tfrac{1}{2}\left(\frac{\log(S/S_0) - (r - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}\right)^2\right)$$

is the log-normal probability density function for $S(T)$.

The important distinction between these two forms for the expectation is that in the first the parameters S_0, r and σ enter the integral through the definition of $S(T)$, whereas in the second they enter through the definition of the probability density function p_S.

B. Monte Carlo sampling and numerical solution of SDEs

The Monte Carlo estimate for the same case of geometric Brownian motion is

$$\widehat{V} = M^{-1} \sum_m f(S^{(m)}),$$

where

$$S^{(m)} = S_0 \exp\left(\left(r - \tfrac{1}{2}\sigma^2\right) T + \sigma\, W^{(m)}\right),$$

with the M values $W^{(m)}$ being independent samples from the probability distribution for $W(T)$. The expected value for the Monte Carlo estimate $\widehat{V}$ is equal to the true expected value V. Because the samples are independent, the variance of the estimate is equal to $M^{-1}\mathbb{V}[f(S)]$, where $\mathbb{V}[f(S)]$ is the variance of a single sample. Thus the root-mean-square sampling error is proportional to $M^{-1/2}$.

In the general case in which the SDE can not be explicitly integrated, the time interval $[0, T]$ is split into N timesteps of size $h = T/N$, and $S^{(j)}$ is replaced by the approximation $\widehat{S}_N^{(j)}$, the value at the end of the N^{th} timestep in a numerical approximation of the SDE. The simplest approximation is the Euler discretisation,

$$\widehat{S}_{n+1} = \widehat{S}_n + a(\widehat{S}_n, t_n)\, h + b(\widehat{S}_n, t_n)\, \Delta W_n \tag{5}$$

in which the Brownian increments ΔW_n are all independent Normal variables with zero mean and variance h. Thus, each path involves N random inputs W_n, to produce the one random output $\widehat{S}_N$.

For more information on Monte Carlo methods, the accuracy of the Euler discretisation, other more accurate discretisations, and various techniques for reducing the variance of the estimator, see Glasserman [14].

C. Evaluating sensitivities

If $V(\theta)$ represents the expected value of the payoff $f(S(T))$ for a particular value of one of the input parameters (e.g. S_0, r or σ in the case of geometric Brownian motion) then for the purposes of hedging and risk analysis one often wants to evaluate $\partial V/\partial\theta$ and $\partial^2 V/\partial^2\theta$.

The simplest approach is to use a finite difference approximation,

$$\frac{\partial V}{\partial \theta} \approx \frac{V(\theta+\Delta\theta) - V(\theta-\Delta\theta)}{2\,\Delta\theta},$$
$$\frac{\partial^2 V}{\partial \theta^2} \approx \frac{V(\theta+\Delta\theta) - 2\,V(\theta) + V(\theta-\Delta\theta)}{(\Delta\theta)^2}.$$

The drawbacks of this approach are that it is computationally expensive, requiring two extra sets of Monte Carlo simulation for each input parameter θ, and care must be taken in the choice of $\Delta\theta$. If it is too large, the finite difference approximation error becomes significant, while if it is too small the variance can become very large if the payoff function $f(S)$ is not differentiable [14].

In the case of a scalar SDE for which one can compute a terminal probability distribution, the second approach, the Likelihood Ratio Method (LRM), differentiates (4) to obtain

$$\frac{\partial V}{\partial \theta} = \int f\,\frac{\partial p_S}{\partial \theta}\,\mathrm{d}S = \int f\,\frac{\partial(\log p_S)}{\partial \theta}\,p_S\,\mathrm{d}S = \mathbb{E}\left[f\,\frac{\partial(\log p_S)}{\partial \theta}\right].$$

The great advantage of this method is that it does not require the differentiation of $f(S)$. This makes it applicable to cases in which the payoff is discontinuous, and it also simplifies the practical implementation because banks usually have complicated flexible procedures through which traders specify payoffs. Second derivatives can also be computed using the LRM approach. Differentiating twice leads to

$$\frac{\partial^2 V}{\partial \theta^2} = \int f\,\frac{\partial^2 p_S}{\partial \theta^2}\,\mathrm{d}S = \mathbb{E}\left[f\,g\right],$$

where the so-called "score" g is defined as

$$g = p_S^{-1}\,\frac{\partial^2 p_S}{\partial \theta^2} = \frac{\partial^2 \log p_S}{\partial \theta^2} + \left(\frac{\partial \log p_S}{\partial \theta}\right)^2.$$

One drawback of LRM is that it requires that $\partial S/\partial W$ is non-zero, and in multi-dimensional cases that it is an invertible matrix. This condition, known as absolute continuity [14], is not satisfied in a few important applications such as the LIBOR market model which is discussed later. The bigger drawback of LRM is that it does not generalise well to path calculations with multiple small timesteps; in most cases it leads to an estimator with a variance which is $O(h^{-1})$, becoming infinite as $h \to 0$ [14].

In the same case of a scalar SDE with a terminal probability distribution, the third approach of pathwise sensitivity differentiates (3) to give

$$\frac{\partial V}{\partial \theta} = \int \frac{\partial f}{\partial S}\,\frac{\partial S(T)}{\partial \theta}\,p_W\,\mathrm{d}W = \mathbb{E}\left[\frac{\partial f}{\partial S}\,\frac{\partial S(T)}{\partial \theta}\right],$$

with the partial derivative $\partial S(T)/\partial\theta$ being evaluated at fixed W. Unlike LRM, this approach generalises very naturally to path calculations, with the Euler discretisation being differentiated, timestep by timestep with fixed Wiener path increments, to compute the sensitivity of the path to changes in the input parameter θ. Differentiating (5) yields

$$\frac{\partial \widehat{S}_{n+1}}{\partial \theta} = \left(1 + \frac{\partial a}{\partial S}\,h + \frac{\partial b}{\partial S}\,\Delta W_n\right)\frac{\partial \widehat{S}_n}{\partial \theta} + \frac{\partial a}{\partial \theta}\,h + \frac{\partial b}{\partial \theta}\,\Delta W_n. \tag{6}$$

Solving this in conjunction with (5) gives $\partial \widehat{S}_N / \partial\theta$ from which we get the Monte Carlo estimate for the first order sensitivity as the average of the sensitivity of M independent paths,

$$\frac{\partial \widehat{V}}{\partial \theta} = M^{-1} \sum_m \frac{\partial f}{\partial S}(\widehat{S}_N^{(m)}) \; \frac{\partial \widehat{S}_N^{(m)}}{\partial \theta}.$$

The second order sensitivity is easily obtained by differentiating a second time.

The key limitation of the pathwise sensitivity approach is the differentiability required of the drift and volatility functions, and the payoff function $f(S)$. The drift and volatility functions are usually twice differentiable, which is sufficient, but financial payoff functions are often discontinuous, and therefore do not satisfy the minimum requirements for first order sensitivities of being continuous and piecewise differentiable with a locally bounded derivative.

However, if the payoff function is suitable, then the pathwise sensitivity estimator has a much lower variance than the LRM estimator, and so it is computationally much more efficient. The efficiency can be further improved when multiple first order sensitivities are required through the use of the adjoint technique introduced by Giles and Glasserman [12] and described in the next section.

III. Adjoint implementation of pathwise sensitivities

A. Mathematical overview

Suppose we have a single Monte Carlo path calculation for a multi-dimensional SDE in which a set of input parameters α (this may include starting prices, forward rates, volatilities, etc.) leads to a final state vector S, which in turn is used to compute a payoff P:

$$\alpha \quad \longrightarrow \quad S \quad \longrightarrow \quad P.$$

This separation of the calculation into two phases, the path simulation and the payoff evaluation, accurately represents a clear distinction in real-world implementations. The path simulation is the computationally demanding phase and is usually implemented very efficiently in C/C++. The payoff evaluation is often implemented less efficiently, sometimes through the use of a scripting language. The reason for this is that the emphasis is on flexibility, making it easy for traders to specify a new financial payoff. The financial products change much more frequently than the SDE models.

We want to compute the derivative of P with respect to each of the elements of α, holding fixed the randomly generated Brownian path increments for this particular path calculation. Adopting the notation used in the Algorithmic Differentiation (AD) research community, let $\dot{\alpha}, \dot{S}, \dot{P}$ denote the derivative with respect to one particular component of α. Straightforward differentiation gives

$$\dot{S} = \frac{\partial S}{\partial \alpha} \; \dot{\alpha}, \qquad \dot{P} = \frac{\partial P}{\partial S} \; \dot{S},$$

and hence

$$\dot{P} = \frac{\partial P}{\partial S} \; \frac{\partial S}{\partial \alpha} \; \dot{\alpha}.$$

Again following the notation used in the AD community the adjoint quantities $\overline{\alpha}, \overline{S}, \overline{P}$ denote the derivatives of P with respect to α, S, P, respectively, with $\overline{P} = 1$ by definition. Differentiating again, with a superscript ^{T} denoting a matrix or vector transpose, one obtains

$$\overline{\alpha} \stackrel{\text{def}}{=} \left(\frac{\partial P}{\partial \alpha} \right)^T = \left(\frac{\partial P}{\partial S} \; \frac{\partial S}{\partial \alpha} \right)^T = \left(\frac{\partial S}{\partial \alpha} \right)^T \overline{S},$$

and similarly

$$\overline{S} = \left(\frac{\partial P}{\partial S} \right)^T \overline{P},$$

giving

$$\overline{\alpha} = \left(\frac{\partial S}{\partial \alpha}\right)^T \left(\frac{\partial P}{\partial S}\right)^T \overline{P}.$$

Note that whereas the standard pathwise sensitivity analysis proceeds forwards through the process (this is referred to as "forward mode" in AD terminology)

$$\dot{\alpha} \quad \longrightarrow \quad \dot{S} \quad \longrightarrow \quad \dot{P}$$

the adjoint analysis proceeds backwards ("reverse mode" in AD terminology),

$$\overline{\alpha} \quad \longleftarrow \quad \overline{S} \quad \longleftarrow \quad \overline{P}.$$

The forward and reverse modes compute exactly the same payoff sensitivities since $\dot{P} = \overline{\alpha}$. The only difference is in the computational efficiency. A separate forward mode calculation is required for each sensitivity that is required. On the other hand, there is only one payoff function (which may correspond to a portfolio consisting of multiple financial products) and so there is always only one reverse mode adjoint calculation to be performed, regardless of the number of sensitivities to be computed. Hence the adjoint calculation is much more efficient when one wants multiple sensitivities.

This is the key characteristic of adjoint calculations. Whenever one is interested in the derivative of one output quantity with respect to many input parameters, the adjoint approach is computationally much more efficient than the forward mode sensitivity calculation. This has resulted in adjoint calculations being used extensively in many areas of computational science such as data assimilation in weather prediction [5], [4], [5], [24] and engineering design optimisation [13], [21].

In the LIBOR testcase considered by Giles and Glasserman [12], up to 240 sensitivities were needed (120 with respect to initial forward rates, and 120 with respect to initial volatilities) and the adjoint approach was up to 50 times more efficient than the standard pathwise sensitivity calculation, while yielding identical results.

B. Algorithmic differentiation

The previous section outlined the mathematics of the discrete adjoint approach. We now look at how the adjoint computer program can be created through applying the ideas of Algorithmic Differentiation. To do this, we look at the mathematics of pathwise and adjoint sensitivity calculation at the lowest possible level to understand how AD works.

Consider a computer program which starts with a number of input variables u_i which can be represented collectively as an input vector $\mathbf{u}^0$. Each step in the execution of the computer program computes a new value as a function of two previous values; unitary functions such as $\exp(x)$ can be viewed as a binary function with no dependence on the second parameter. Appending this new value to the vector of active variables, the n^{th} execution step can be expressed as

$$\mathbf{u}^n \;=\; \mathbf{f}^n(\mathbf{u}^{n-1}) \;\equiv\; \begin{pmatrix} \mathbf{u}^{n-1} \\ f_n(\mathbf{u}^{n-1}) \end{pmatrix}, \tag{7}$$

where f_n is a scalar function of two of the elements of $\mathbf{u}^{n-1}$. The result of the complete N steps of the computer program can then be expressed as the composition of these individual functions to give

$$\mathbf{u}^N = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \ldots \circ \mathbf{f}^2 \circ \mathbf{f}^1(\mathbf{u}^0). \tag{8}$$

Defining $\dot{\mathbf{u}}^n$ to be the derivative of the vector $\mathbf{u}^n$ with respect to one particular element of $\mathbf{u}^0$, differentiating (7) gives

$$\dot{\mathbf{u}}^n = D^n \, \dot{\mathbf{u}}^{n-1}, \quad D^n \equiv \begin{pmatrix} I^{n-1} \\ \partial f_n / \partial \mathbf{u}^{n-1} \end{pmatrix}, \tag{9}$$

with I^{n-1} being the identity matrix with dimension equal to the length of the vector $\mathbf{u}^{n-1}$. The derivative of (8) then gives

$$\dot{\mathbf{u}}^N = D^N \, D^{N-1} \ldots \, D^2 \, D^1 \, \dot{\mathbf{u}}^0, \tag{10}$$

which gives the sensitivity of the entire output vector to the change in one particular element of the input vector. The elements of the initial vector $\dot{\mathbf{u}}^0$ are all zero except for a unit value for the particular element of interest. If one is interested in the sensitivity to N_I different input elements, then (10) must be evaluated for each one, at a cost which is proportional to N_I.

The above description is of the forward mode of AD sensitivity calculation [26], which is intuitively quite natural. The reverse, or adjoint, mode is computationally much more efficient when one is interested in the sensitivity of a small number of output quantities with respect to a large number of input parameters [15]. Defining the column vector $\overline{\mathbf{u}}^n$ to be the derivative of a particular element of the output vector u_i^N with respect to the elements of $\mathbf{u}^n$, then through the chain rule of differentiation we obtain

$$\begin{aligned} \left(\overline{\mathbf{u}}^{n-1}\right)^T &= \frac{\partial u_i^N}{\partial \mathbf{u}^{n-1}} = \frac{\partial u_i^N}{\partial \mathbf{u}^n} \, \frac{\partial \mathbf{u}^n}{\partial \mathbf{u}^{n-1}} = \left(\overline{\mathbf{u}}^n\right)^T D^n, \\ &\Longrightarrow \quad \overline{\mathbf{u}}^{n-1} = \left(D^n\right)^T \, \overline{\mathbf{u}}^n. \end{aligned} \tag{11}$$

Hence, the sensitivity of the particular output element to all of the elements of the input vector is given by

$$\overline{\mathbf{u}}^0 = \left(D^1\right)^T \, \left(D^2\right)^T \ldots \, \left(D^{N-1}\right)^T \, \left(D^N\right)^T \, \overline{\mathbf{u}}^N. \tag{12}$$

Note that the reverse mode calculation proceeds backwards from $n = N$ to $n = 1$. Therefore, it is necessary to first perform the original calculation forwards from $n = 1$ to $n = N$, storing all of the partial derivatives needed for D^n (or sufficient information to be able to re-compute them efficiently; in practice there is often a tradeoff between the storage and computational requirements) before then doing the reverse mode calculation.

If one is interested in the sensitivity of N_O different output elements, then (12) must be evaluated for each one, at a cost which is proportional to N_O. Thus the reverse mode is computationally much more efficient than the forward mode when $N_O \ll N_I$.

Looking in more detail at what is involved in (9) and (11), suppose that the n^{th} step of the original program involves the computation

$$c = f(a, b).$$

The corresponding forward mode step will be

$$\dot{c} = \frac{\partial f}{\partial a} \, \dot{a} + \frac{\partial f}{\partial b} \, \dot{b},$$

at a computational cost which is no more than a factor 3 greater than the original nonlinear calculation. In matrix form, this equation can be written as

$$\begin{pmatrix} \dot{a} \\ \dot{b} \\ \dot{c} \end{pmatrix}^n = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \frac{\partial f}{\partial a} & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \dot{a} \\ \dot{b} \end{pmatrix}^{n-1}.$$

Transposing the matrix gives

$$\begin{pmatrix} \overline{a} \\ \overline{b} \end{pmatrix}^{n-1} = \begin{pmatrix} 1 & 0 & \frac{\partial f}{\partial a} \\ 0 & 1 & \frac{\partial f}{\partial b} \end{pmatrix} \begin{pmatrix} \overline{a} \\ \overline{b} \\ \overline{c} \end{pmatrix}^n,$$

and hence the corresponding reverse mode step consists of two calculations:

$$\begin{aligned} \overline{a} &:= \overline{a} + \frac{\partial f}{\partial a}\,\overline{c} \\ \overline{b} &:= \overline{b} + \frac{\partial f}{\partial b}\,\overline{c}. \end{aligned}$$

At worst, this has a theoretical cost which is a factor 4 greater than the original nonlinear calculation [16]. In practice, as we will see later, the ratio of the execution times can be even smaller than this.

C. AD tools

Algorithmic Differentiation gives a clear prescriptive process by which a reverse mode adjoint code may be written. However, a manual programming implementation can be tedious and error-prone, even for an expert. This has led to the development of AD tools which automate this process.

AD tools can be divided into two categories [16]. Source transformation tools take as an input an existing code (usually written in FORTRAN) and generate a new code (in the same language) to perform either forward mode or reverse mode sensitivity calculations. In previous work in the field of Computational Fluid Dynamics [11], we have obtained excellent results using the Tapenade package developed by Hascoët and Pascual at INRIA [6], [18]. Other notable source transformation packages are TAMC/TAF [8], [9], [22] and ADIFOR [2]. However, all three of these packages currently treat only FORTRAN codes; work is in progress to extend them to C/C++ but there are technical challenges because of structures, pointers, operator overloading, classes and templates. Preliminary results are presented later for TAC++ which is the C/C++ version of TAF [9].

The alternative approach which is popular for C++ uses operator overloading. ADOL-C [17], [25] is one popular package, but in this paper we have used FADBAD++ [1], [23] which is particularly easy to use. Operator overloading for forward mode sensitivity calculation is fairly easy to understand. "Active" variables, those whose value will change when the input parameters of interest are varied, have their type changed from `double` to `F<double>`. The `F<double>` type for a variable x holds both the value of the variable, and one or more sensitivities $\dot{x}$. Operations involving either one or two `F<double>` variables are then defined in the natural way through operator overloading specifications in a C++ header file:

$$\begin{array}{lll}
\text{addition:} & x + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x+y \\ \dot{y} \end{pmatrix} & \begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x+y \\ \dot{x}+\dot{y} \end{pmatrix} \\
\text{multiplication:} & x * \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x*y \\ x*\dot{y} \end{pmatrix} & \begin{pmatrix} x \\ \dot{x} \end{pmatrix} * \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x*y \\ \dot{x}*y + x*\dot{y} \end{pmatrix} \\
\text{division:} & x/ \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x/y \\ -(x/y^2)*\dot{y} \end{pmatrix} & \begin{pmatrix} x \\ \dot{x} \end{pmatrix} / \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} x/y \\ \dot{x}/y - (x/y^2)*\dot{y} \end{pmatrix} \\
\text{exponentiation:} & \exp \begin{pmatrix} y \\ \dot{y} \end{pmatrix} = \begin{pmatrix} \exp(y) \\ \exp(y)*\dot{y} \end{pmatrix} &
\end{array}$$

Operator overloading for reverse mode computations is very much harder to understand. In essence, it stores the history of all operations performed in the forward pass, during which it computes the partial derivatives of each arithmetic operation. Then, when the adjoint sensitivities are desired, it goes backwards through this stored history, computing the required adjoint values [16], [17]. In using FADBAD++, this is accomplished by defining the active variables to have type `B<double>`, and defining the output variable of interest by assigning it a unit adjoint sensitivity.

Further information about AD tools and publications is available from the AD research community website `http://www.autodiff.org` which includes links to the major groups working in this field.

```
/* Monte Carlo LIBOR path calculation                        */
/* template enables use of AD through operator overloading */

template <typename ADdouble>
void path_calc(const int N, const int Nmat, const double delta,
               ADdouble L[], const double lambda[], const double z[])
{
  int      i, n;
  double   sqez, lam, con1;
  ADdouble v, vrat;

  for(n=0; n<Nmat; n++) {
    sqez = sqrt(delta)*z[n];

    v = 0.0;
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
  }
}
```

TABLE I
TEMPLATED C++ CODE TO PERFORM LIBOR PATH CALCULATION

D. Test application

The test problem is the same LIBOR market model example which was used in the previous paper by Giles and Glasserman [12]. Letting L_i^n denote the forward LIBOR rate for the time interval $[i\,\delta, (i{+}1)\,\delta)$ at time $n\,\delta \le i\,\delta$, then taking the timestep to be equal to the LIBOR interval δ, the evolution of the forward rates L_i^n for $n = 0, \ldots, N_{mat} - 1$ is approximated by the discrete equations

$$L_i^{n+1} = L_i^n \, \exp\left((\sigma_{i-n-1} S_i - \tfrac{1}{2}\sigma_{i-n-1}^2)\,\delta + \sigma_{i-n-1} Z^n \sqrt{\delta} \right), \quad i > n,$$

where Z^n is the unit Normal random variable for the n^{th} timestep, and

$$S_i^n = \sum_{j=n+1}^{i} \frac{\sigma_{j-n-1}\,\delta\, L_j^n}{1 + \delta\, L_j^n}, \quad i > n.$$

The model treats the volatility as being a function of time to maturity. Once a rate reaches its maturity it remains fixed, so we set $L_i^{n+1} = L_i^n$ if $i \le n$. A portfolio of N_{opt} different swaptions with swap rates swap_n and maturity mat_n has payoff

$$P = \left\{ \prod_{i=0}^{N_{mat}-1} \frac{1}{1 + \delta\, L_i} \right\} \left\{ \sum_{n=1}^{N_{opt}} 100\ (1 - B_{\text{mat}_n} - \text{swap}_n C_{\text{mat}_n})_+ \right\},$$

milliseconds/path	Gnu `g++`	Intel `icc`
original	0.37	0.10
hand-coded forward	0.97	0.52
hand-coded reverse	0.47	0.19
FADBAD++ forward	4.30	5.00
FADBAD++ reverse	6.20	4.86
hybrid forward	1.02	0.63
hybrid reverse	0.65	0.35
TAC++ forward	1.36	0.85
TAC++ reverse	1.28	0.45

TABLE II
TIMINGS FOR LIBOR CALCULATION IN MILLISECONDS PER PATH

where

$$B_m = \prod_{i=1}^{m} \frac{1}{1+\delta\, L_{N_{mat}+i-1}}, \qquad C_m = \sum_{i=1}^{m} \delta\, B_i.$$

The C++ implementation [10] requires about 10 lines of code for the path calculation, plus approximately 20 lines of code for the payoff evaluation. The hand-coded forward mode sensitivity calculation adds almost twice as much code, and the hand-coded reverse mode calculation adds about three times as much code. The forward mode programming was very straightforward, taking just a day or so, comparable to the time taken in writing the original code. The reverse mode programming was significantly more difficult, taking two or three days, despite the fact that I consider myself to be quite expert at writing reverse mode adjoint codes based on AD principles. A non-expert should expect to take at least twice as long, and it probably takes at least a year of developing adjoint codes to become an expert.

The FADBAD++ programming was very straightforward. Because I am still a novice C++ programmer, having used FORTRAN for most of my career it took one day for the forward mode, and another day for the reverse mode. However, with the benefit of that experience, the same task could be carried out in the future for a new application in an hour or two. The use of C++ templates is central to this ease-of-use. Table I presents the code for the path calculation. Three different "instances" of this routine are called by the main application code. In the first, the input variable `L` has type `double`; this is used for the original code. In the second and third versions, `L` has type `F<double>` and `B<double>`, respectively; these are used for the forward mode and reverse mode FADBAD++ calculations. Based on the calling sequences in each case, looking at the type of the input parameters, the compiler automatically generates each of these instances as needed. Thus the type `ADdouble` in the source code is a dummy type to be replaced by the actual type of the input variable `L`. This leads to very simple programming with a single copy of each key routine.

Table II gives execution times in milliseconds per path calculation. The timings are for a computation with 40 timesteps, and sensitivities with respect to each one of the 80 initial forward rates. There are two sets of timings, one for the Gnu compiler `g++` and the other for the Intel compiler `icc`; for each the compiler flags were set to achieve the highest possible level of code optimisation, and the tests were run on a machine with a single Intel Pentium 4 CPU. The results shows times for the original code and different versions of forward and reverse mode sensitivity codes. The hybrid versions use the hand-coded routine for the path calculation together with the FADBAD++ routine for the payoff evaluation, and the TAC++ results are obtained from code generated automatically by the source transformation tool TAC++ being developed by FastOpt [9].

The first point to make is that all of the sensitivity calculations produce identical values, to machine

precision. As one might expect, the most efficient code is the hand-coded reverse mode; the timings are similar to those reported in [12] with the valuation plus 80 sensitivities being obtained at a cost which is less than twice the cost of calculating the valuation on its own. The FADBAD++ implementations are significantly less efficient than the hand-coded routines, suggesting that the best use of FADBAD++ might be as a validation tool to check the correctness of the hand-coded routines. However, the hybrid results show that one gets almost as much efficiency when combining the hand-coded path calculation with the FADBAD++ payoff evaluation. This is because most of the computational effort is in the path calculation, and it could be very attractive in real-world applications because most of the programming effort is usually in the payoff evaluation.

The code generated by TAC++ is very efficient in both forward and reverse modes. The efficiency of the forward mode code relies heavily on compiler optimisation to eliminate a large number of repeated sub-expressions in the code produced by TAC++. Older versions of the Gnu compiler failed to make these optimisations and produced executable code which was much slower.

The interested reader is encouraged to download the source code for this testcase [10], investigate the performance with their preferred compiler, study the use of FADBAD++ and try to understand the hand-coded forward and reverse mode routines.

IV. "Vibrato" Monte Carlo and hybrid Greeks

A. Conditional expectation

The paper which introduced the adjoint approach to Monte Carlo sensitivity calculations [12] was well received by the finance community. They were excited by the computational savings it offered, but nevertheless had some reservations. The previous section has addressed some concerns over the practical implementation, but a more fundamental issue is that pathwise sensitivity is inapplicable in cases in which the payoff function is discontinuous. Also, even when the payoff function is continuous and piecewise differentiable, calculating the derivative can pose significant practical difficulties because of the flexible software framework which is often used to make it possible for traders to easily specify new financial products. This section addresses these concerns by developing a mathematical approach which only require payoff values, not their derivatives, through combining the best features of pathwise and LRM sensitivity calculations.

The Oxford English Dictionary describes "vibrato" as "a rapid slight variation in pitch in singing or playing some musical instruments". The analogy to Monte Carlo methods is the following; whereas a path simulation in a standard Monte Carlo calculation produces a precise value for the output values from the underlying stochastic process, in the vibrato Monte Carlo approach the output values have a narrow probability distribution.

This is a generalisation of the technique of conditional expectation discussed by Glasserman in section 7.2.3 of his book [14] as a solution to the problem of discontinuous payoffs. In his example, a path simulation is performed in the usual way for the first $N-1$ timesteps, at each timestep taking a value for the Wiener increment ΔW^n which is a sample from the appropriate Gaussian distribution, and then using (5) to update the solution. On the final timestep, one instead considers the full distribution of possible values for ΔW^N. This gives a Gaussian distribution for $\widehat{S}_N$ at time T,

$$p_S(\widehat{S}_N) = \frac{1}{\sqrt{2\pi}\,\sigma_W}\ \exp\left(-\frac{(\widehat{S}_N-\mu_W)^2}{2\,\sigma_W^2}\right) \tag{13}$$

where

$$\mu_W = \widehat{S}_{N-1} + a(\widehat{S}_{N-1}, T-h)\,h, \quad \sigma_W = b(\widehat{S}_{N-1}, T-h)\,\sqrt{h},$$

with $a(S,t)$ and $b(S,t)$ being the drift and volatility of the SDE described in (1). Hence, the conditional

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

$$f(S(T)) \;=\; H(S(T)-K) \equiv \left\{ \begin{array}{ll} 1, & S(T) > K \\ 0, & S(T) \leq K \end{array} \right.$$

is

$$\mathbb{E}[f(\widehat{S}_N) \,|\, \widehat{S}_{N-1}] \;=\; \int_{-\infty}^{\infty} H(\widehat{S}_N - K)\; p_S(\widehat{S}_N)\; \mathrm{d}\widehat{S}_N \;=\; \Phi\left(\frac{\mu_W - K}{\sigma_W}\right)$$

where $\Phi(\cdot)$ is the cumulative Normal distribution function.

The Monte Carlo estimator for the option value is now

$$\widehat{V} = M^{-1} \sum_m \mathbb{E}[f(\widehat{S}_N) \,|\, \widehat{S}_{N-1}^{(m)}],$$

and because the conditional expectation $\mathbb{E}[f(\widehat{S}_N) \,|\, \widehat{S}_{N-1}]$ is a differentiable function of the input parameters (provided $\sigma(S)$ is twice differentiable) the pathwise sensitivity approach can now be used.

There are two difficulties in using this form of conditional expectation in real-world applications. This 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 a change to the complex software framework used to specify payoffs.

The solution is to use a Monte Carlo estimate of the conditional expectation, and use LRM to obtain its sensitivity. Thus, the technique which is proposed combines pathwise sensitivity for the path calculation with LRM sensitivity of the payoff calculation. offering the computational efficiency of the pathwise approach, together with the generality and ease-of-implementation of LRM.

B. Vibrato Monte Carlo

The idea is very simple; adopting the idea of conditional expectation, each path simulation for a particular discrete set of Wiener increments W computes a Gaussian conditional probability distribution $p_S(\widehat{S}_N|W)$ for the state of the path approximation at time T. For a scalar SDE, if μ_W and σ_W are the mean and standard deviation for given W, then

$$\widehat{S}_N(W, Z) = \mu_W + \sigma_W Z,$$

where Z is a unit Normal random variable. The expected payoff can then be expressed as

$$V = \mathbb{E}_W\left[\mathbb{E}_Z[f(\widehat{S}_N) \,|\, W]\right] = \int \left\{ \int f(\widehat{S}_N)\; p_S(\widehat{S}_N|W)\; \mathrm{d}\widehat{S}_N \right\}\; p_W(W)\; \mathrm{d}W.$$

The outer expectation/integral is an average over the different values for the discrete Wiener increments, while the inner conditional expectation/integral is averaging over Z.

Now, to compute the sensitivity to an input parameter θ, the first step is to apply the pathwise sensitivity approach at fixed W to obtain $\partial\mu_W/\partial\theta, \partial\sigma_W/\partial\theta$. We then apply LRM to the inner conditional expectation to get

$$\frac{\partial V}{\partial \theta} \;=\; \mathbb{E}_W\left[\frac{\partial}{\partial\theta}\,\mathbb{E}_Z\left[f(\widehat{S}_N) \,|\, W\right]\right] \;=\; \mathbb{E}_W\left[\mathbb{E}_Z\left[f(\widehat{S}_N)\;\frac{\partial(\log p_S)}{\partial\theta} \;|\; W\right]\right],$$

where p_S is defined in (13) and

$$\frac{\partial(\log p_S)}{\partial\theta} = \frac{\partial(\log p_S)}{\partial\mu_W}\;\frac{\partial\mu_W}{\partial\theta} + \frac{\partial(\log p_S)}{\partial\sigma_W}\;\frac{\partial\sigma_W}{\partial\theta}.$$

The Monte Carlo estimators for V and $\partial V/\partial\theta$ have the form

$$\widehat{V} = M^{-1}\sum_j \mathrm{CEE}_f^{(j)}, \quad \widehat{\frac{\partial V}{\partial\theta}} = M^{-1}\sum_j \mathrm{CEE}_{\partial f/\partial\theta}^{(j)},$$

where $\mathrm{CEE}_f^{(j)}$ and $\mathrm{CEE}_{\partial f/\partial\theta}^{(j)}$ are Conditional Expectation Estimates for $\mathbb{E}_Z\left[f(\widehat{S}_N)\,|\,W\right]$ and $\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\theta}\,|\,W\right]$, respectively.

Although the discussion so far has had a single output spot price at the terminal term T, the idea extends very naturally to multiple outputs at the final time, producing a multivariate Gaussian distribution. If the payoff also depends on values at intermediate times, not just at maturity, these can be handled by omitting the simulation time closest to each measurement time, using for that time interval a timestep which is twice as big as the usual timestep. The distribution at the measurement time can then be obtained from Brownian interpolation [14].

C. Efficient estimators

It is important to have efficient estimators for $\mathbb{E}_Z[f(\widehat{S}_N)\,|\,W]$ and $\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\theta}\,|\,W\right]$. For a given W, using the identity $\mathbb{E}_Z[h(Z)] = \mathbb{E}_Z[h(-Z)]$ for all functions $h(Z)$ due to the symmetry of the probability distribution for Z, then

$$\begin{aligned}\mathbb{E}_Z\left[f(\widehat{S}_N)\right] &= f(\mu_W) + \mathbb{E}_Z\left[f(\mu_W+\sigma_W Z) - f(\mu_W)\right] \\ &= f(\mu_W) + \mathbb{E}_Z\left[\tfrac{1}{2}\Big(f(\mu_W+\sigma_W Z) - 2f(\mu_W) + f(\mu_W-\sigma_W Z)\Big)\right].\end{aligned}$$

If $f(S)$ is differentiable, the second term is the expectation of a quantity which has a very small mean and variance, and so a single sample will be sufficient for the estimator.

In the case of a one-dimensional SDE,

$$\log p_S = -\log\sigma_W - \frac{(\widehat{S}_N-\mu_W)^2}{2\sigma_W^2} - \tfrac{1}{2}\log(2\pi)$$

and for a given W,

$$\begin{aligned}\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\theta}\right] &= \frac{\partial\mu_W}{\partial\theta}\,\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\mu_W}\right] \\ &+ \frac{\partial\sigma_W}{\partial\theta}\,\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\sigma_W}\right].\end{aligned}$$

Looking at the first of the two expectations, then

$$\begin{aligned}\mathbb{E}_Z\left[f(\widehat{S}_N)\frac{\partial(\log p_S)}{\partial\mu_W}\right] &= \mathbb{E}_Z\left[\frac{\widehat{S}_N-\mu_W}{\sigma_W^2}\,f(\widehat{S}_N)\right] \\ &= \mathbb{E}_Z\left[\frac{Z}{\sigma_W}\,f(\mu_W+\sigma_W Z)\right] \\ &= \mathbb{E}_Z\left[\frac{Z}{2\sigma_W}\left(f(\mu_W+\sigma_W Z) - f(\mu_W-\sigma_W Z)\right)\right].\end{aligned}$$

If $f(S)$ is differentiable, this is the expectation of a quantity which is $O(1)$ in magnitude. One sample may be sufficient, but if the computational cost of evaluating the payoff is small compared to the path

calculation, it is probably better to use several samples. If $f(S)$ is discontinuous, then for paths near the discontinuity the expectation is of a quantity which is $O(\sigma_W^{-1}) = O(h^{-1/2})$ and multiple samples should definitely be used to estimate the expected value.

Similarly, using the additional result that $\mathbb{E}_Z[Z^2-1] = 0$,

$$\begin{aligned}
\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\sigma_W}\right] &= \mathbb{E}_Z\left[\left(-\frac{1}{\sigma_W}+\frac{(\widehat{S}_N-\mu_W)^2}{\sigma_W^3}\right)\,f(\widehat{S}_N)\right] \\
&= \mathbb{E}_Z\left[\frac{Z^2-1}{\sigma_W}\;f(\mu_W+\sigma_W Z)\right] \\
&= \mathbb{E}_Z\left[\frac{Z^2-1}{\sigma_W}\;\big(f(\mu_W+\sigma_W Z)-f(\mu_W)\big)\right] \\
&= \mathbb{E}_Z\left[\frac{Z^2-1}{2\,\sigma_W}\,\big(f(\mu_W+\sigma_W Z)-2f(\mu_W)+f(\mu_W-\sigma_W Z)\big)\right].
\end{aligned}$$

The expression within this expectation is in general no larger than for the previous expectation, and so the same set of samples will suffice.

These estimators can be generalised to the case of multiple assets with a multivariate Gaussian distribution conditional on the discrete set of Wiener increments. If μ_W is now the column vector of means, and Σ_W is the covariance matrix, then $\widehat{S}_N$ can be written as

$$\widehat{S}_N(W,Z) = \mu_W + CZ,$$

where Z is a vector of uncorrelated unit Normal variables (and hence its covariance matrix is the identity matrix) and C is any matrix such that

$$\Sigma_W = \mathbb{E}\left[(\widehat{S}_N-\mu_w)\,(\widehat{S}_N-\mu_w)^T\right] = \mathbb{E}\left[CZ\,Z^TC^T\right] = C\;\mathbb{E}\left[ZZ^T\right]C^T = C\,C^T.$$

Provided Σ_W is non-singular, which corresponds to the requirement that $d_1 = d_2$ in (1), then the joint probability density function for S is

$$\log p_S = -\tfrac{1}{2}\log|\Sigma| - \tfrac{1}{2}(\widehat{S}_N-\mu_W)^T\Sigma_W^{-1}(\widehat{S}_N-\mu_W) - \tfrac{1}{2}d\,\log(2\pi).$$

Differentiating this (see [7], [20]) gives

$$\frac{\partial\log p_S}{\partial\mu_W} \;=\; \Sigma_W^{-1}(\widehat{S}_N-\mu_W) \;=\; C^{-T}Z,$$

and

$$\frac{\partial\log p_S}{\partial\Sigma_W} \;=\; -\tfrac{1}{2}\Sigma^{-1}+\tfrac{1}{2}\Sigma^{-1}(\widehat{S}_N-\mu_W)(\widehat{S}_N-\mu_W)^T\Sigma^{-1} \;=\; \tfrac{1}{2}\,C^{-T}\left(ZZ^T-I\right)C^{-1},$$

and then for a given W

$$\begin{aligned}
\mathbb{E}_Z\left[f(\widehat{S}_N)\;\frac{\partial(\log p_S)}{\partial\theta}\right] &= \left(\frac{\partial\mu_W}{\partial\theta}\right)^T\mathbb{E}_Z\left[f(\widehat{S}_N)\;\frac{\partial(\log p_S)}{\partial\mu_W}\right] \\
&+ \;\text{Trace}\left(\frac{\partial\Sigma_W}{\partial\theta}\;\mathbb{E}_Z\left[f(\widehat{S}_N)\;\frac{\partial(\log p_S)}{\partial\Sigma_W}\right]\right),
\end{aligned}$$

where the trace of a matrix is the sum of its diagonal elements.

To obtain efficient estimators, we again use the symmetry property to obtain

$$\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\mu_W}\right] = \mathbb{E}_Z\left[\tfrac{1}{2}\left(f(\mu_W{+}CZ) - f(\mu_W{-}CZ)\right)C^{-T}Z\right].$$

and we use $\mathbb{E}_Z[ZZ^T{-}I\,] = 0$ to give

$$\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\Sigma_W}\right] = \mathbb{E}_Z\left[\tfrac{1}{2}\left(f(\mu_W{+}CZ) - 2f(\mu_W) + f(\mu_W{-}CZ)\right)C^{-T}(ZZ^T{-}I)\,C^{-1}\right].$$

D. Adjoint Greeks

This approach is completely compatible with an adjoint calculation of the path sensitivity. In the scalar case, one would first compute the path, and then estimate the quantities

$$\mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\mu_W}\right], \quad \mathbb{E}_Z\left[f(\widehat{S}_N)\,\frac{\partial(\log p_S)}{\partial\sigma_W}\right].$$

These values correspond to what AD terminology would call $\overline{\mu_W}$ and $\overline{\sigma_W}$, the sensitivity of the expected payoff for that path to changes in μ_W and σ_W. This is the starting information required for the reverse pass of the adjoint path calculation, following the algorithmic differentiation procedure described in Section III.

In the multi-dimensional case, the adjoint initialisation is

$$\overline{\mu_W} = \mathbb{E}_Z\left[\tfrac{1}{2}\left(f(\mu_W{+}CZ) - f(\mu_W{-}CZ)\right)C^{-T}Z\right],$$

and

$$\overline{\Sigma_W} = \mathbb{E}_Z\left[\tfrac{1}{2}\left(f(\mu_W{+}CZ) - 2f(\mu_W) + f(\mu_W{-}CZ)\right)C^{-T}(ZZ^T{-}I)\,C^{-1}\right].$$

V. Conclusions

In this paper we have given an overview of the Monte Carlo method for determining the value of financial options, and their sensitivity to changes in input parameters. The adjoint implementation of pathwise sensitivity calculations is a particularly efficient way to obtain the sensitivity with respect to a large set of input parameters, and we have shown the way in which adjoint codes can be developed through the principles of Algorithmic Differentiation, and with the aid of AD tools. Readers are encouraged to download the source code for the LIBOR testcase [10] to better understand the software development process.

Lastly, we have introduced a new idea of vibrato Monte Carlo calculations. This is a generalisation of the use of conditional expectation and leads to a hybrid method which applies pathwise sensitivity analysis to the path simulation, and Likelihood Ratio Method to the payoff evaluation. This offers the computational efficiency of the pathwise method, together with the greater generality and ease-of-implementation of LRM.

Acknowledgements

This research was funded in part by a research grant from Microsoft Corporation, and in part by a fellowship from the UK Engineering and Physical Sciences Research Council.

I am very grateful to Ole Stauning for providing FADBAD++ and patiently responding to my questions, and to Thomas Kaminski and Michael Vossbeck of FastOpt who generated the TAC++ code and timing results and gave very helpful feedback on the paper.

References

[1] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[2] C.H. Bischof, A. Carle, P.D. Hovland, P. Khademi, and A. Mauer. ADIFOR 2.0 User's Guide (Revison D). Technical Report 192, Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[3] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[4] I. Charpentier and M. Ghemires. Efficient adjoint derivatives: application to the meteorological model Meso-NH. *Opt. Meth. and Software*, 13(1):35–63, 2000.

[5] P. Courtier and O. Talagrand. Variational assimilation of meteorological observations with the adjoint vorticity equation, II, Numerical results. *Q. J. R. Meteorol. Soc.*, 113:1329–1347, 1987.

[6] F. Courty, A. Dervieux, B. Koobus, and L. Hascoet. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Opt. Meth. and Software*, 18(5):615–627, 2003.

[7] P.S. Dwyer. Some applications of matrix derivatives in multivariate analysis. *Journal of the American Statistical Association*, 62(318):607–625, 1967.

[8] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Software*, 24(4):437–474, 1998.

[9] R. Giering and T. Kaminski. TAF and TAC++ product information. `http://www.fastopt.com/topics/products.html`, FastOpt, 2007.

[10] M.B. Giles. Source code for LIBOR testcase for adjoint sensitivities generated both by hand-coded routines and FADBAD++ automatic differentiation. `http://www.comlab.ox.ac.uk/mike.giles/libor/`, Oxford University Computing Laboratory, 2007.

[11] M.B. Giles, D. Ghate, and M.C. Duta. Using automatic differentiation for adjoint CFD code development. In B. Uthup, S. Koruthu, R.K. Sharma, and P. Priyadarshi, editors, *Recent Trends in Aerospace Design and Optimization*, pages 426–434. Tata McGraw-Hill, New Delhi, 2006.

[12] M.B. Giles and P. Glasserman. Smoking adjoints: fast Monte Carlo Greeks. *RISK*, January 2006.

[13] M.B. Giles and N.A. Pierce. An introduction to the adjoint approach to design. *Flow, Turbulence and Control*, 65(3-4):393–415, 2000.

[14] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, 2004.

[15] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[16] A. Griewank. *Evaluating derivatives : principles and techniques of algorithmic differentiation*. SIAM, 2000.

[17] A. Griewank, D. Juedes, and J. Utke. ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.

[18] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. `http://www-sop.inria.fr/tropics/`, INRIA, 2004.

[19] P.E. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*. Springer-Verlag, Berlin, 1992.

[20] J.R. Magnus and H. Neudecker. *Matrix differential calculus with applications in statistics and econometrics*. John Wiley & Sons, 1988.

[21] E. Nielsen and W.K. Anderson. Aerodynamic design optimization on unstructured meshes using the Navier-Stokes equations. *AIAA J.*, 37(11):957–964, 1999.

[22] C. Othmer, T. Kaminski, and R. Giering. Computation of topological sensitivities in fluid dynamics: cost function versatility. In P. Wesseling, E. E. Oñate, and J. Périaux, editors, *ECCOMAS CFD 2006*. TU Delft, 2006.

[23] O. Stauning and C. Bendtsen. FADBAD++ online documentation. `http://www2.imm.dtu.dk/~km/FADBAD/`, Technical University of Denmark, 2007.

[24] O. Talagrand and P. Courtier. Variational assimilation of meteorological observations with the adjoint vorticity equation, I, Theory. *Q. J. R. Meteorol. Soc.*, 113:1311–1328, 1987.

[25] A. Walther and A. Griewank. ADOL-C online documentation. `http://www.math.tu-dresden.de/~adol-c/`, Technical University of Dresden, 2007.

[26] R.E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.