Adjoint methods for computing sensitivities in local volatility surfaces

Juraj Spilda

St. Edmund Hall

University of Oxford

A thesis submitted in partial fulfillment of the MSc in

Mathematical and Computational Finance

June 2010

Contents

Acknowledgements

I would like to thank Professor Mike Giles for all his support in the making of this project. He continued to provide me with the support and motivation I needed and was always helpful. I am especially thankful for all his early-morning replies to all my late-night e-mails. One could only hope everyone was lucky enough to have such a dedicated supervisor.

I would also like to thank the SPP Foundation for helping me secure the funding needed to undergo the studies that led to this project.

Finally, I would like to thank my family and friends for their continual love and support.

Abstract

In this paper we present the adjoint method of computing sensitivities of option prices with respect to nodes in the local volatility surface. We first introduce the concept of algorithmic differentiation and how it relates to path-wise sensitivity computations within a Monte Carlo framework. We explain the two approaches available: forward mode and adjoint mode. We illustrate these concepts on the simple example of a model with a geometric Brownian motion driving the underlying price process, for which we compute the Delta and Vega in forward and adjoint mode. We then go on to explain in full detail how to apply these ideas to a model where the underlying has a volatility term defined by a local volatility surface. We provide source codes for both the simple and the more complex case and analyze numerical results to show the strengths of the adjoint approach.

Chapter 1

An introduction to adjoint methods

1.1 Introduction

Monte Carlo methods are becoming an ever more ubiquitous approach to pricing complex derivative contracts in the financial industry due to the fact that they are easily adaptable to any type of complex contract and acquiring results of greater accuracy is just a matter of providing more computing power. But as these methods become so wide-spread, the total strain on computer centers in banks becomes problematic to handle. Therefore, an important challenge banks face is reducing computational costs involved in pricing derivatives. The greatest computational cost in dealing with derivatives is incurred when computing the sensitivities, the so-called Greeks, of derivative contracts. For that purpose, Giles and Glasserman [2006] propose an efficient way to compute a large number of Greeks simultaneously using the so-called adjoint method for path-wise sensitivity computations, making use of knowledge from the field of algorithmic differentiation. They demonstrate an application of the adjoint method on the example of a LIBOR model. Since the release of that article, many academic papers have endeavored into using adjoint methods in a Monte Carlo setting, such as Achdou and Pironneau [2005], Denson and Joshi [2009a], Denson and Joshi [2009b], Kaebe et al. [2009] and Leclerc et al. [2009].

In this paper, we shall analyze the application of adjoints to computing option price sensitivity to an entire grid of local volatility surface spline nodes. The paper is organized as follows: first, we shall explain the principles of the adjoint method and compare it to the standard forward derivative simulation technique. To demonstrate the adjoint method we will first show its application to computing Greeks on the simple example of geometric Brownian motion driving the underlying. We will then

move on to use the same adjoint technique on a model with a volatility determined by a local volatility surface. Here, we will compute the sensitivity of the payoff with respect to each spline node in the volatility surface obtained by Dupire's formula. Finally, we will analyze numerical results obtained from the algorithm.

1.2 Path-wise sensitivity

When computing option sensitivities, better known as the Greeks, in a Monte Carlo framework, we attempt to evaluate $\frac{\partial V}{\partial \theta} = \frac{\partial}{\partial \theta}\mathbb{E}[P(\theta)]$, where V is the current value of the contract, P is the payoff of the contract and θ is the parameter with respect to which we are computing the sensitivity. Within the literature, such as Glasserman [2003], there are three standard ways to compute Greeks: the finite difference 'bumping' method, the path-wise sensitivity method and the likelihood ratio method. We will be interested in the path-wise sensitivity method, which lends itself well for use in conjunction with adjoint algorithmic differentiation.We will here briefly establish path-wise sensitivity for further reference later on.

The fundamental idea of path-wise sensitivity lies within its name: we attempt to evaluate the sensitivity over each simulated path and then take the mean of these sensitivities. More specifically, it means transferring the evaluation of the derivative from the payoff function to the process of the underlying:

$$\frac{\partial V}{\partial \theta} = \frac{\partial}{\partial \theta}\mathbb{E}\left[P(\theta)\right] = \mathbb{E}\left[\frac{\partial P}{\partial S(T)}\frac{\partial S(T,\theta)}{\partial \theta}\right] \tag{1.1}$$

We must note, however, that for equation (1.1) to hold, we must require the payoff P to be continuous when computing first-order Greeks, and that it have a continuous first derivative for second-order Greeks. Otherwise, we cannot bring the derivative into the expectation function. A clear counter-example where this condition is not satisfied is a standard digital option, where $\frac{\partial P}{\partial S(T)} = 0$ and hence $\mathbb{E}\left[\frac{\partial P}{\partial S(T)}\frac{\partial S(T)}{\partial S_0}\right] = 0$, but the Delta $\frac{\partial V}{\partial S_0}$ is obviously not 0 everywhere (this limitation can be partially sidestepped by using payoff smoothing techniques).

To evaluate the expectation, we use the estimator $\frac{1}{N}\sum_{i=1}^{N}\frac{\partial P}{\partial S}(S^{(i)}(T))\frac{\partial S^{(i)}(T)}{\partial \theta}$, where $S^{(i)}(T)$ is the final value of the underlying obtained by simulating a single path. To find $\frac{\partial S^{(i)}}{\partial \theta}$, we differentiate the entire Euler-Maryuama recurrence that drives the underlying path evolution. Let the recurrence take the form

$$S_{n+1} = a(\theta; S_n, t_n)\Delta t + b(\theta; S_n, t_n)\Delta W_n.$$

Then by differentiating we obtain the corresponding recurrence relation for the derivative

$$\dot{S}_{n+1} \stackrel{\text{def}}{=} \frac{\partial S_{n+1}}{\partial \theta} = \frac{\partial a}{\partial \theta}\Delta t + \frac{\partial b}{\partial \theta}\Delta W_n + \left(\frac{\partial a}{\partial S_n}\Delta t + \frac{\partial b}{\partial S_n}\Delta W_n\right)\dot{S}_n, \tag{1.2}$$

which we initialise by the value $\dot{S}_0 = \frac{\partial S_0}{\partial \theta}$. Now we have everything needed to find the value of the estimator and hence compute the required sensitivity. We can easily obtain Monte Carlo estimates of second-order Greeks by differentiating the Euler scheme again, although the payoff derivative is rarely continuous. In the more common case, where the derivative of the payoff is discontinuous, the best we can do is use a smooth approximation of any discontinuities that may arise in the derivative. For these purposes, we use a smoothed Heaviside function, which is efficiently reproduced by a modified standard normal cumulative distribution function $\Phi\left(\frac{S-K}{\varepsilon}\right)$, where K is the stock price at which discontinuity occurs (since the discontinuity usually occurs at a strike price specified in the contract, we use K to denote the point of discontinuity in the derivative) and $\varepsilon \ll K$ is a small value. Figure 1.1 illustrates how well this approximates the true Heaviside function. Since we are using the smoothing function

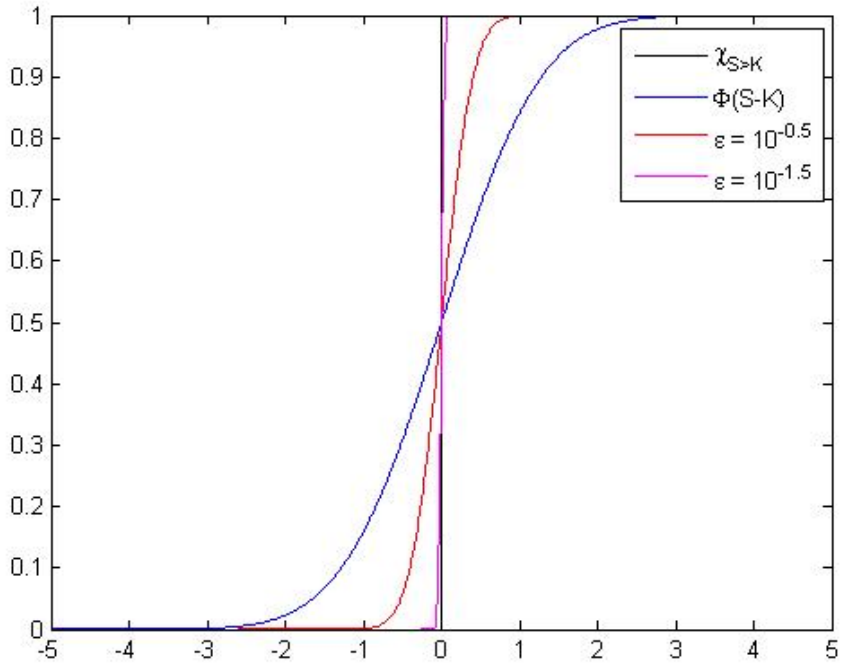

Figure 1.1: Comparison of the Heaviside function $H(S) = \chi_{S>K}$, the cumulative distribution function $\Phi(S-K)$ and the smoothed Heaviside function $\Phi\left(\frac{S-K}{\varepsilon}\right)$ for values $K = 0$ and $\varepsilon = 10^{-0.5}, 10^{-1.5}$.

to estimate the first derivative, we must integrate it to obtain the function we should use to estimate the payoff in the region of discontinuity. Working out the integration, we find that we use the function

$$f(S) = (S-K)\Phi\left(\frac{S-K}{\varepsilon}\right) + \frac{\varepsilon}{\sqrt{2\pi}}\exp\left(-\frac{(S-K)^2}{2\varepsilon^2}\right).$$

Now we can comfortably compute derivatives of any order, since our estimating function is infinitely differentiable. We must however keep in mind we have introduced a numerical imprecision into our computation and can expect this to become more pronounced the further we differentiate.

1.3 Forward and reverse mode

To understand the workings of the adjoint, or *reverse mode*, we will first look at how standard derivative computations are perceived from the perspective of algorithmic differentiation (AD). We will refer to this standard approach as *forward mode*. As is implicitly hinted in the names, the adjoint algorithm will be closely tied to that of the forward computation. The reason for the naming of the modes will later become apparent.

1.3.1 Forward mode

To illustrate the principles of forward mode differentiation, let us start by analyzing an example presented in Giles [2007]. We shall adopt the notation commonly used among AD researchers in literature such as Griewank and Walther [2008]. Let us consider a function $c = c(a, b)$,where a,b might depend on some initial parameter x_0. Then computing the derivative $\dot{c} = \frac{\partial c}{\partial x_0}$ naturally results in what is called a forward mode computation

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

If we include the trivial equations $\dot{a} = \dot{a}, \dot{b} = \dot{b}$, we can also write the equation in matrix form:

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

We thus have a system that preserves all previously computed derivatives and expands the output by an additional derivative. Griewank and Walther [2008] show that this forward computation is at most 3 times as costly as the original evaluation of

function $c(a, b)$, which more importantly means that computing the forward derivative is directly proportional to the cost of the function and thus differentiation cannot cause an algorithm to become overly computationally expensive as long as the original calculation is not.

We can now generalize this situation. If we have a d-dimensional vector $\mathbf{x}^0$ of input parameters for a composite vector function $\mathbf{f} = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \ldots \circ \mathbf{f}^1(\mathbf{x}^0)$ with intermediate vector functions $\mathbf{f}^n$ of the form

$$\mathbf{f}^n(\mathbf{x}^{n-1}) = \begin{pmatrix} \mathbf{x}^{n-1} \\ f^n(\mathbf{x}^{n-1}) \end{pmatrix},$$

(f^n is a function with a scalar output) resulting in $\mathbf{f}$ giving us a $(N+d)$-dimensional output vector $\mathbf{x}^N$, then finding the derivative of that output with respect to the k-th element x_k^0 of $\mathbf{x}^0$, noted as $\dot{\mathbf{x}}^N = \frac{\partial \mathbf{x}^N}{\partial x_k^0}$, can be seen as a recursive relation, where each step of the recursion is equivalent to the simple example previously presented. Specifically, we can write the relation as

$$\dot{\mathbf{x}}^n = \frac{\partial \mathbf{f}^n}{\partial \mathbf{x}^{n-1}} \dot{\mathbf{x}}^{n-1},$$

or if seen in matrix form:

$$\dot{\mathbf{x}}^n = D^n \dot{\mathbf{x}}^{n-1}, D^n \equiv \begin{pmatrix} I^{n-1} \\ \partial f^n / \partial \mathbf{x}^{n-1} \end{pmatrix}, \tag{1.3}$$

where I^{n-1} is an identity matrix of dimension $(d+n-1) \times (d+n-1)$. Stringing the recursion together, the resulting relation for computing the derivative is

$$\dot{\mathbf{x}}^N = D^N D^{N-1} \cdots D^1 \dot{\mathbf{x}}^0$$

where the value of $\dot{\mathbf{x}}^0$ is known, since all of its elements are 0 except for the k-th element, which has value 1.

Let us describe how this would generalize to a case where we want to compute the derivative with respect to all the initial inputs. For such a case, $\dot{\mathbf{x}}^n$ would become a matrix of the corresponding size, where every column of the matrix would correspond to the derivatives with respect to one of the input parameters. Thus, our recursion would require us to do N sparse matrix multiplications of size at least $d \times d$ and at most $(N+d) \times (N+d)$.

We can now put this into perspective with regards to path-wise sensitivity computations. There, the composite function whose derivative we are trying to evaluate is defined by the recursion for time-stepping; $S_{n+1} = f^n(S_n, \mathbf{x}^n)$, where the $(d+n)$-dimensional vector $\mathbf{x}^n$ consists of all the parameters that might come into the calculation, such as the risk-free interest rate, volatility, etc., and n intermediate derivatives

$\dot{S}_j, j = 1...n$. If we assume that all parameters are condensed into vector $\mathbf{x}^0$, then we can see our computation as a string of interconnected inputs and outputs:

$$\mathbf{x}^0 \to S_0 \to S_1 \to \cdots \to S_N \to P$$

Thus to compute the derivative of the payoff with respect to one parameter in the input, we must evaluate the expression

$$\dot{P} = \frac{\partial P}{\partial S}\dot{S} = \frac{\partial P}{\partial S}\frac{\partial S}{\partial \mathbf{x}^0}\dot{\mathbf{x}}^0 \tag{1.4}$$

Here, the recursive computation, which corresponds to recursion (1.2), is hidden within the expression $\frac{\partial S}{\partial \mathbf{x}^0}$. We can see that thus to evaluate the derivative, we have to compute dot terms in the same order as the original inputs and outputs were interconnected:

$$\dot{\mathbf{x}}^0 \to \dot{S}_0 \to \dot{S}_1 \to \cdots \to \dot{S}_N \to \dot{P}$$

In terms of computational complexity, since each recursive step is a vector operation as noted in (1.3), we end up doing N vector multiplications to evaluate the derivative with respect to one input parameter. If we were to use forward mode to compute the derivative with respect to each of the parameters, we would have to run the algorithm above for each parameter individually, or as presented previously, we would have to run it once but doing costly matrix multiplications at every point of the recursion.

1.3.2 Reverse mode

We will now take an analogical approach to algorithmic differentiation and work backwards to obtain derivatives with respect to all the initial input parameters simultaneously. For this, we again use the notation used in AD research:

$$\bar{x} \equiv \frac{\partial P}{\partial x}$$

Let us again first explore the properties of this operator on the simple scalar example $c = c(a, b)$. We know that $\bar{a} = \frac{\partial P}{\partial a} = \bar{c}\frac{\partial c}{\partial a}$ and that an analogous relation holds for b. But at this point we must consider the case that there are numerous other functional relationships in which a and b serve as input parameters. For example, if at a different point in the calculations we would know that $d = d(a)$, then the relationship for $\bar{a}$ alters to $\bar{a} = \frac{\partial P}{\partial a} = \bar{d}\frac{\partial d}{\partial a} + \bar{c}\frac{\partial c}{\partial a}$. Thus, when evaluating the bar values for one given relationship, it is better to view $\bar{a}, \bar{b}$ as cumulative variables:

$$\bar{a} := \bar{a} + \frac{\partial c}{\partial a}\bar{c}$$
$$\bar{b} := \bar{b} + \frac{\partial c}{\partial b}\bar{c}$$

or in matrix form

$$\begin{pmatrix} \bar{a} \\ \bar{b} \end{pmatrix} := \begin{pmatrix} 1 & 0 & \frac{\partial c}{\partial a} \\ 0 & 1 & \frac{\partial c}{\partial b} \end{pmatrix} \begin{pmatrix} \bar{a} \\ \bar{b} \\ \bar{c} \end{pmatrix}.$$

We can see the matrix representing the bar operation is simply a transpose of the matrix obtained in forward mode. Griewank and Walther [2008] show that the computational complexity of the reverse evaluation of derivatives is, just as in the case of forward mode, proportional to the complexity of evaluating the original function $c = c(a, b)$. Unlike forward mode, which is at most 3 times as costly, reverse mode has a slightly higher maximum cost and is at most 4 times as costly.

Again, if we view this as a building block to an entire sequence of functions $\mathbf{x}^n = \mathbf{f}^n(\mathbf{x}^{n-1})$ and string the recursion together, the resulting relation for computing the derivative is

$$\bar{\mathbf{x}}^0 = (D^0)^T (D^1)^T \cdots (D^N)^T \bar{\mathbf{x}}^N$$

where D^n are the matrices introduced in (1.3). Thus we can in one sweep compute the sensitivity of the final output $\mathbf{x}^N$ to all the input parameters $\mathbf{x}^0$. It is important to point out at this point the savings in computational costs. In this adjoint case, by doing a vector multiplication and updating the values of a vector, we finally obtain all the required sensitivities. This is in stark contrast with forward mode, where this same amount of computer operations would only yield us one of the sensitivities and we would have to resort to a computation involving matrix multiplications to obtain all the wanted sensitivities.

Now we can again look at how this reverse mode computation applies to path-wise sensitivity. We can write an analogical relation to (1.4) in terms of this new notation:

$$\bar{\mathbf{x}}^0 = \left(\frac{\partial S}{\partial \mathbf{x}^0}\right)^T \bar{S} = \left(\frac{\partial S}{\partial \mathbf{x}^0}\right)^T \left(\frac{\partial P}{\partial S}\right)^T \bar{P}$$

The informational flow is then as follows:

$$\bar{\mathbf{x}}^0, \bar{S}_0 \leftarrow \bar{S}_1 \leftarrow \cdots \leftarrow \bar{S}_N \leftarrow \bar{P}$$

Knowing that the initial value is $\bar{P} \equiv 1$, we can now execute this string of computations.

1.4 A simple example: geometric Brownian motion

To illustrate the ideas of forward and adjoint mode presented above, we shall first show their application on the simple example of a European call option with a geometric Brownian motion driving the price process of the underlying. In this case, the price evolution is described by the standard SDE

$$dS_t = rS_t dt + \sigma S_t dW_t.$$

For the discrete numerical simulation we will use the Euler-Maruyama approximation scheme

$$S_{n+1} = (1 + r\Delta t + \sigma \Delta W_n)\, S_n, \Delta W_n \sim N(0, \Delta S).$$

We shall compute the Delta of the contract by applying ideas presented in Giles and Glasserman [2006], first in the standard forward mode for path-wise sensitivity, then in the reverse, adjoint mode.

1.4.1 Computing the Delta - forward and reverse mode

When calculating the Delta of a contract in forward mode, we want to obtain $\dot{P} = \frac{\partial P}{\partial S_0}$, which we then use in the Monte Carlo estimator. For this, we use relation (1.4), where our initial parameter $\mathbf{x}^0$ is simply the current stock price S_0. From (1.4) we see that the value we are after is $\dot{S}_N$, since we already know $\frac{\partial P}{\partial S_N}$, the derivative of the payoff at the time of maturity t_N. To obtain $\dot{S}_N$, we differentiate the Euler scheme to obtain the recurrence

$$\dot{S}_{n+1} = (1 + r\Delta t + \sigma \Delta W_n)\, \dot{S}_n = D(n)\dot{S}_n.$$

The initial condition $\dot{S}_0 = 1$ then easily allows us to find $\dot{S}_N$, which in turn allows us to compute $\dot{P} = \frac{\partial P}{\partial S_0} = \frac{\partial P}{\partial S_N}\dot{S}_N$.

In the adjoint approach, we will strive to find the value of $\bar{S}_0 = \frac{\partial P}{\partial S_0} = \dot{P}$. For this, we will use the recursion

$$\bar{S}_n = \left(\frac{\partial P}{\partial S_n}\right)^T = \left(\frac{\partial P}{\partial S_{n+1}}\frac{\partial S_{n+1}}{\partial S_n}\right)^T = \left(\frac{\partial S_{n+1}}{\partial S_n}\right)^T \bar{S}_{n+1} = (D(n))^T\, \bar{S}_{n+1}. \qquad (1.5)$$

We work through this recursion backwards, starting with the terminal condition $\bar{S}_N = \frac{\partial P}{\partial S_N}$, which is known and sufficiently differentiable in the case of a standard call option.

When comparing the two approaches in practical terms, we have to generate the time-stepping of the stock price process in both forward and adjoint mode. In forward mode, we simultaneously have to generate the corresponding process for $\dot{S}_n$. In adjoint mode, we do not need to do so, but we do need to store the intermediate variables $D(n)$ from the stock price generation. Then we back out the sensitivity with respect to S_0 from a backward recursion. In this case, when we are only computing a single sensitivity, there are no computational savings to be made by using adjoint mode, it is potentially even more costly. But it is an easy extension to compute further sensitivities without the need for additional path simulations that makes the adjoint mode so useful. We shall illustrate this by extending the computations to calculate the option's Vega.

1.4.2 Computing the Vega - forward and reverse mode

The forward mode for computing the Vega is much like that for the Delta. The thing that changes is the calculation of $\dot{S}_N$, which is now obtained through the altered recursion

$$\dot{S}_{n+1} = (1 + r\Delta t + \sigma\Delta W_n)\,\dot{S}_n + S_n\Delta W_n = D(n)\dot{S}_n + B(n)$$

with the initial condition $\dot{S}_0 = 0$. The remainder of the analysis remains the same for the case of computing the Vega in forward mode - analogically to the Delta, we evaluate $\dot{P} = \frac{\partial P}{\partial \sigma} = \frac{\partial P}{\partial S_N}\dot{S}_N$.

In adjoint mode, however, things change a bit more. Our goal is to obtain $\bar{\sigma}$. For that we use the relation $\bar{\sigma} = \left(\frac{\partial P}{\partial \sigma}\right)^T = \left(\frac{\partial S_N}{\partial \sigma}\right)^T \bar{S}_N$. We know the relationship for $\frac{\partial S_N}{\partial \sigma}$ from the forward computation, as it is the same as computing the dot value. But we do not need to compute the dot value to find $\bar{\sigma}$ in adjoint mode. If we simply string the $\dot{S}$ recurrence relation together and plug into our expression for $\bar{\sigma}$, we obtain an expression in the form

$$\begin{aligned}\bar{\sigma} &= \left(\frac{\partial S_N}{\partial \sigma}\right)^T \bar{S}_N = \left(D(N-1)\frac{\partial S_{N-1}}{\partial \sigma} + B(N-1)\right)^T \bar{S}_N \\ &= \left[B(N-1) + D(N-1)\left(B(N-2) + D(N-2)(\cdots)\cdots\right)\right]^T \bar{S}_N.\end{aligned}$$

If we look back at (1.5) we notice that our expression for $\bar{\sigma}$ simplifies significantly and can be rewritten as a sum

$$\bar{\sigma} = \sum_{n=0}^{N-1} B(n)^T \bar{S}_{n+1}$$

Let us note then that we cannot really compute the Vega $\bar{\sigma}$ without also calculating the Delta $\bar{S}_0$. This shows just how interconnected the processes for computing various sensitivities in adjoint mode really are. Computing the Vega only requires very little extra computational effort. Specifically, we need to store the intermediate variables $B(n)$ that are generated within the price process simulation and run the computationally inexpensive summation shown above. From our analysis of first-order Greeks, we can see how this would generalize to computing second-order derivatives.

1.4.3 Debugging and code validation

We have established algorithms to compute adjoint and forward mode results and expect these to give us the same results. But how can we be sure that either of these are giving us correct results? To validate the results of our algorithms and make sure that both forward and adjoint mode are giving us the true values of the derivatives, we use the so-called complex variable trick (CVT) established in Squire and Trapp [1998]. The basic idea of this technique is to introduce a complex element to the variable we want to differentiate with respect to, which then propagates through the recursive calculations and allows us to compute the derivative of the final output by use of a Taylor expansion approximation. Specifically, let us assume the final result of all the forward path calculations is a composite function $f(x_0)$, where x_0 is the initial parameters with respect to which we want to differentiate (f may have other inputs as well). Then by Taylor expansion we know that

$$f(x_0 + ih) = f(x_0) + ihf'(x_0) - h^2\frac{f''(x_0)}{2} + \mathcal{O}(h^3); \qquad i = \sqrt{-1}$$

and thus to compute a good estimate of the derivative, we can use the expression

$$f'(x_0) = \frac{Im(f(x_0 + ih))}{h} + \mathcal{O}(h^3).$$

The advantage of this calculation is that we side-step the problem of subtracting two very similar values $f(x_0 + h), f(x_0 - h)$ that arises in finite difference methods and that causes these to be inaccurate for very small values of h. With the CVT, we can take a step as small as $h = 10^{-20}$ and thus obtain the derivative up to machine precision.

In terms of our path-wise Delta and Vega calculations, the CVT asks us to introduce a complex part to the parameter with respect to which we want to differentiate, i.e. $S_0 := S_0 + ih$ and $\sigma := \sigma + ih$ respectively. If we do so and follow the propagation of the complex part over the time-stepping recursion for stock price and then transform

that according to the payoff, we finally obtain the value we require for the Delta and Vega: $\dot{P} = \bar{S}_0 = \frac{Im(P)}{h} + \mathcal{O}(h^3)$ and $\dot{P} = \bar{\sigma} = \frac{Im(P)}{h} + \mathcal{O}(h^3)$ respectively. We use this relation to check that we compute the correct dot and bar values for every individual simulated path. The maximum error we obtained across all the simulations done was of order 10^{-16}. Thus we consider the adjoint and forward mode computations to be exact and reliable.

1.4.4 Source code

To better illustrate the exact workings of the forward and adjoint mode on the example of geometric Brownian motion, we include MATLAB source code for doing the calculations:

```
%
% Calculating sensitivites of GBM on a EU call using adjoint technique
% Use Euler-Maruyama method and Pathwise Sensitivity Method to
% estimate Vega and Delta for European call option based on usual GBM
%
% problem parameters
%
eps = 1e-20; %for CVT
r   = 0.05;
%sig = 0.5 + 1i*eps; %use to compute Vega through CVT
sig = 0.5;
T   = 1;
S0  = 100+1i*eps; %inserting complex part for use in CVT
K   = 110;
%
% Monte Carlo simulation parameters
%
M  = 1e+5;   % total number of Monte Carlo paths
M2 = 1e+4;   % number of paths at a time
N = 256;
h = T/N;

%setting up estimator variables
sum1 = 0;
```

```
sum2 = 0;
sigbarsum1 = 0;
sigbarsum2 = 0;
err = 0; %for CVT

for m = 1:M2:M
    m2 = min(M2,M-m+1);

    S = S0*ones(1,m2);
    sdot = ones(1,m2);
    D = zeros(N,m2);
    B = zeros(N,m2);
    for n = 1:N
      dW = sqrt(h)*randn(1,m2);
      %storing variables for adjoint mode
      D(n,:) = 1+r*h+sig*dW;
      B(n,:) = S.*dW;
      %forward mode
      sdot  = sdot.*D(n,:);
      S  = S.*D(n,:);
    end

    %setting sbar(N)
    sbar = zeros(1,m2);
    for i = 1:m2
       if(real(S(i))>K)
           sbar(i) = exp(-r*T);
           S(i) = exp(-r*T)*S(i);
       else
           sbar(i) = 0;
           S(i) = 0;
       end
    end

    %adjoint recursion
    sigbar = zeros(1,m2);
```

```
    for n=0:(N-1)
       sigbar = sigbar + sbar.*B(N-n,:);
       sbar = D(N-n,:).*sbar;
    end

    % Check our error in dS_N\dS_0:
    % sdot,sbar will have gained an imag part because of our CVT
    %- we only want real part to give us dS_N\dS_0.
    sbar = real(sbar); sdot = real(sdot);
    % imag(S)/eps will give us CVT value for dS_N\dS_0
    err = max(err,max(abs(sbar - imag(S)/eps),abs(sdot - imag(S)/eps)));

    %using the adjoint mode results
    sum1 = sum1 + sum(sbar);
    sum2 = sum2 + sum(sbar.^2);
    sigbarsum1 = sigbarsum1 + sum(sigbar);
    sigbarsum2 = sigbarsum2 + sum(sigbar.^2);
end

delta_approx  = sum1/M; %delta estimator
sig2 = (sum2/M - (sum1/M)^2); %variance
disp(sprintf('numerical delta: %f  +/- %f',delta_approx,3*sqrt(sig2/M)))

vega_approx  = sigbarsum1/M; %vega estimator
sig3 = (sigbarsum2/M - (sigbarsum1/M)^2); %variance
disp(sprintf('numerical vega: %f  +/- %f',vega_approx,3*sqrt(sig3/M)))
disp(['error in dS_N\dS_0 vs CVT result: ' num2str(err)]);
```

1.5 The black box view

Having seen a concrete example of how to utilize the adjoint method, let us now look back at the algorithm from the entirely abstract perspective of a computer scientist. Instead of interconnected vector functions being our basic building blocks, let us just consider any set of programming routines or just individual lines of code that have inputs and outputs that are interconnected, as depicted in figure 1.2.

Each of those building blocks is a black box to us, and it is not important for us to

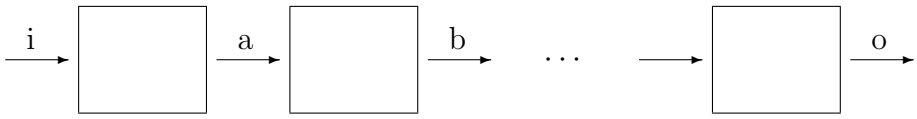

Figure 1.2: Dataflow between individual computer codes.

know what goes on in it to be able to deduce the structure of the forward and adjoint algorithm. Each of these black boxes has an equivalent black box containing the dot or bar algorithm respectively, using the respective dot values $\dot{i}, \dot{a}, \dot{b}, \dot{o}$ or bar values $\bar{i}, \bar{a}, \bar{b}, \bar{o}$ as input and output, just as we saw illustrated on the case of geometric Brownian motion. We can assume these black boxes are available, there are even automated programs such as FADBAD++ that will generate the source code for these black boxes for you. Now that we know these black boxes and the obvious fact that $\dot{i} = \frac{\partial i}{\partial i} = 1$, we are able to compute the dot values $\dot{a}, \dot{b}, \dot{o}$. Equally, we know that $\bar{o} = 1$ and hence can compute all the bar values $\bar{a}, \bar{b}, \bar{i}$.

Moreover, the black box view gives us an excellent way to debug our code and check that our calculations are correct. If we consider one of these boxes, with input a and output b and corresponding dot and bar inputs and outputs $\dot{a}, \dot{b}, \bar{a}, \bar{b}$, then we can verify that the dot and bar algorithms are producing the correct outputs by checking the relation

$$(\bar{a})^T \dot{a} = \frac{\partial o}{\partial a}\frac{\partial a}{\partial i} = \frac{\partial o}{\partial b}\frac{\partial b}{\partial i} = \left(\bar{b}\right)^T \dot{b}.$$

If this relation holds, then we are sure that both the forward and adjoint mode algorithm for the specific box are working correctly, otherwise either one or the other is malfunctioning. This relation holds for all the boxes in the chain, thus ultimately giving the relation $\bar{i} = \dot{o} = (\bar{a})^T \dot{a} = \left(\bar{b}\right)^T \dot{b}$.

1.6 The need for 'binning'

In terms of program flow, we must also consider a situation where there is a costly preprocessing procedure, e.g. a Cholesky matrix factorization, that comes before the Monte Carlo simulations on our main recursively evaluated composite function $\mathbf{f}$ introduced earlier in this chapter. This costly procedure alters the input vector $\mathbf{x}^0$ and we subsequently use the altered $\tilde{\mathbf{x}}^0$ to initialize the recursion for $\mathbf{f}$. We now want to compute the sensitivity of the final output with respect to the parameters that enter

into this preprocessing procedure and do not appear in any of the later computations. The preprocessing does not pose any problem in this sensitivity computation in forward mode, since we only need to run the corresponding dot version of the preprocessing procedure once and then we run as many Monte Carlo simulations of the dot algorithm as needed, obtaining an estimator and its confidence interval. However, it poses a challenge to implement an adjoint mode algorithm, as is pointed out in Capriotti and Giles [2010] on the example of introducing correlation into the input vector $\mathbf{x}^0$, since we have difficulty obtaining a confidence interval for the estimator of the sensitivities. In the natural adjoint approach we work backwards, first running the adjoint version of the Monte Carlo simulations, which result in an estimator and confidence interval for the bar values of all the input variables $\tilde{\mathbf{x}}^0$. At this point however, we use this estimator, and not the individual simulation results, as input into the adjoint version of the preprocessing procedure. This gives us the required average sensitivities with respect to input parameters in the preprocessing procedure. But since we have only run this adjoint procedure once, we only obtain the average sensitivities, but no confidence interval for these and we have no way to transfer the confidence interval across the adjoint computation as well.

The alternative we have is to run the adjoint preprocessing code for every Monte Carlo path simulation, and thus obtain an estimate and confidence interval for the sensitivities with respect to parameters defining the preprocessing procedure. However, this means we have to run the procedure repeatedly, which is most likely extremely computationally costly, since adjoint computations are at most 4 times as costly as the original procedure (for proof see Griewank and Walther [2008]). To find a feasible solution we must strike a compromise between the two extremes. We divide our N_{MC} Monte Carlo simulations into N_B equal sets, or 'bins', of $B = \frac{N_{MC}}{N_B}$ simulations. For each bin, we only run the adjoint preprocessing procedure once, the input into it being an average of the intermediate adjoint variables obtained from the B simulations. We thus obtain an average preprocessing sensitivity, but no confidence interval. Finally, since we do this N_B times, we obtain the standard estimator and confidence interval we require for the average preprocessing sensitivities.

For illustrative purposes, let us demonstrate this on the example in Capriotti and Giles [2010]. Let us aim to obtain the sensitivities of a payoff to a correlation matrix $\rho = CC^T$ (CC^T is the Cholesky matrix factorization), which is used to produce correlation in the input variables $\mathbf{x}^0$, i.e. the composite function takes $\tilde{\mathbf{x}}^0 = C\mathbf{x}^0$ as input. If we run our adjoint algorithm for one path, we gradually obtain $\bar{\tilde{\mathbf{x}}}^0$ and consequently $\bar{C}$. Doing this B times within one 'bin' eventually gives us an average value for $\bar{C}$ with

a confidence interval. However, there is no way for us to maintain that confidence interval when using the adjoint computation $\bar{\rho}_{i,j} = \sum_{l,m=1}^{N} \frac{\partial C_{l,m}}{\partial \rho_{i,j}} \bar{C}_{l,m}$ for the correlation matrix ρ, since we only run this computation once within one 'bin' and we have no way to back out a confidence interval from one sample. Thus, although the simulated sensitivities we have are averages and are quite possibly good estimates of the true sensitivities, they are of little use to us without knowing their confidence intervals. Therefore, we have to run N_B 'bins' of simulations, each of these bins computing their own estimate for $\bar{\rho}_{i,j}$. We can now use these to obtain an estimate with a confidence interval.

In essence, our problem is analogous to the one we face in Quasi Monte Carlo methods. Although the use of low discrepancy sequences of quasi-random numbers with useful properties give us results that are very similar to those obtained by standard Monte Carlo number generation, these results lack a confidence interval, since we use completely deterministic variables in the computations. Therefore, we use randomized Quasi Monte Carlo to introduce an element of randomness and thus to regain a confidence interval in our computations. Analogically to the 'binning' technique, we generate a certain number of random offsets, denoted e.g. N_B, that is much lower than the total number of quasi-random numbers in the low discrepancy sequence, and divide the sequence into N_B sets corresponding to each of the random offsets. This consequently allows us to use Quasi Monte Carlo methods to obtain estimators with regular confidence intervals.

Chapter 2

The adjoint method and the local volatility surface

Now that we have seen how simple it is to deduce computations needed for the adjoint mode, especially if we already have an idea about forward mode calculations, we will try to apply these ideas to an application with a much greater scale. We will simultaneously compute the sensitivity of an option with respect to the values of all the nodes that are used as calibration points in generating a local volatility surface through interpolation techniques.

2.1 The local volatility surface

The concept of the local volatility surface was first introduced in 1994 from three different and independent perspectives simultaneously. Two of these approaches, presented in Derman and Kani [1994] and Rubinstein [1994], are based on a discrete binary tree modeling framework. The third approach, based on a continuous time framework of option pricing, was introduced around the same time in Dupire [1994]. It is the latter that has become an industry standard on which to build procedures for computing local volatility surfaces. It allows us to extract information about the market view on short-term volatility from currently quoted option prices on a range of strike prices and maturity dates $V(T, K; S_0)$. By Dupire's approach, assuming that the stock pays zero dividend, the formula for local volatility is then simply given by

$$\sigma^2(S, t; S_0)_{|S=K, t=T} = \frac{2\left(\frac{\partial V}{\partial T} + rK\frac{\partial V}{\partial K}\right)}{K^2\frac{\partial^2 V}{\partial K^2}}.$$

This formula requires a whole continuum of option prices so we turn to finite difference methods to use this formula for data that we have in reality. We thus obtain

a discrete set of nodes at quoted strikes and maturities that form the foundation for an entire surface. As we can see, the formula depends on the current stock price S_0 (it is implicitly hidden in the option prices and is important in the derivation of the equation) and hence local volatility is only useful for analyzing short-term market views and must be updated frequently. It complements stochastic volatility models well, since those have trouble capturing volatility behaviour for short-term contracts. A problematic feature of Dupire's local volatility computation is its instability when estimated through finite difference methods. We can see that the formula can become hard to evaluate when the contract price exhibits linear growth in the strike price, making the denominator of the equation approach 0. The level of instability can vary across various choices of interpolation techniques and boundary conditions, and numerous ways to stabilize the computations have been proposed in literature (such as by Tikhonov regularization in Crepey [2003] and Hanke and Rosler [2005] or entropy minimization in Avellaneda et al. [1997] and Samperi [2002]). Due to all this instability, we may be quite interested to know how a possible error in the volatility surface calibration influences the pricing of an option within this framework. For this, we need to compute the sensitivity of the option price with respect to each of the calibration nodes. In view of our previous analysis of the adjoint method, we can see that this application is well suited for adjoint differentiation - a large number of inputs in the form of volatility surface nodes, and just a single output in the form of an option price.

2.2 Our setup

We shall consider a situation where we are given a grid of computed local volatility nodes $\sigma_{i,j} = \sigma(T_i, S_j), i = 1...I, j = 1...J$. To price an option as a risk-neutral expectation $\mathbb{E}[P(S(T))]$, we consider the underlying to be a geometric Brownian motion with a volatility term defined by the local volatility:

$$dS_t = rS_t dt + \sigma(S_t, t)S_t dW_t.$$

We again use the Euler-Maruyama scheme to discretize the SDE into

$$S_{n+1} = S_n \left(1 + r\Delta t + \sigma(S_n, t_n)\Delta W_n\right)$$

We can now see that we are faced with the problem of obtaining a value for $\sigma_n = \sigma(S_n, t_n)$ when the Monte Carlo time steps t_n do not coincide with the local volatility grid times T_i. To obtain this value, we first linearly interpolate over time and then

similarly interpolate over the stock price. Interpolation over time gives us intermediate values

$$\sigma_j^{(n)} = \sigma_{i,j} + \frac{t_n - T_i}{T_{i+1} - T_i} \left(\sigma_{i+1,j} - \sigma_{i,j}\right) = \sigma_{i,j} + \beta(n) \left(\sigma_{i+1,j} - \sigma_{i,j}\right).$$

$\sigma_j^{(n)}$ is a value that lies on the local volatility grid in terms of stock price but lies at a time determined by our Euler scheme time-stepping (see figure 2.1). For this volatility value $\sigma_j^{(n)}$ there is a corresponding stock price S_j for which $S_n \in (S_j, S_{j+1})$ and a time T_i for which $t_n \in (T_i, T_{i+1})$. We can now similarly interpolate to obtain σ_n at the simulated stock price S_n:

$$\sigma_n = \sigma(S_n, t_n) = \sigma_j^{(n)} + \frac{S_n - S_j}{S_{j+1} - S_j} \left(\sigma_{j+1}^{(n)} - \sigma_j^{(n)}\right) = \sigma_j^{(n)} + \alpha(S_n) \left(\sigma_{j+1}^{(n)} - \sigma_j^{(n)}\right).$$

Figure 2.1 provides us with a visualization of our grid setup and the process of interpolation. Let us point out again that there is a slight abuse in notation, since S_n refers to a path value at time-step t_n, but S_j refers to a fixed grid coordinate in the local volatility surface (represented by the horizontal lines in figure 2.1). Since we are working on a finite local volatility grid, we have to handle the issue of obtaining a volatility in a situation where the Euler scheme generates a stock price value that lies

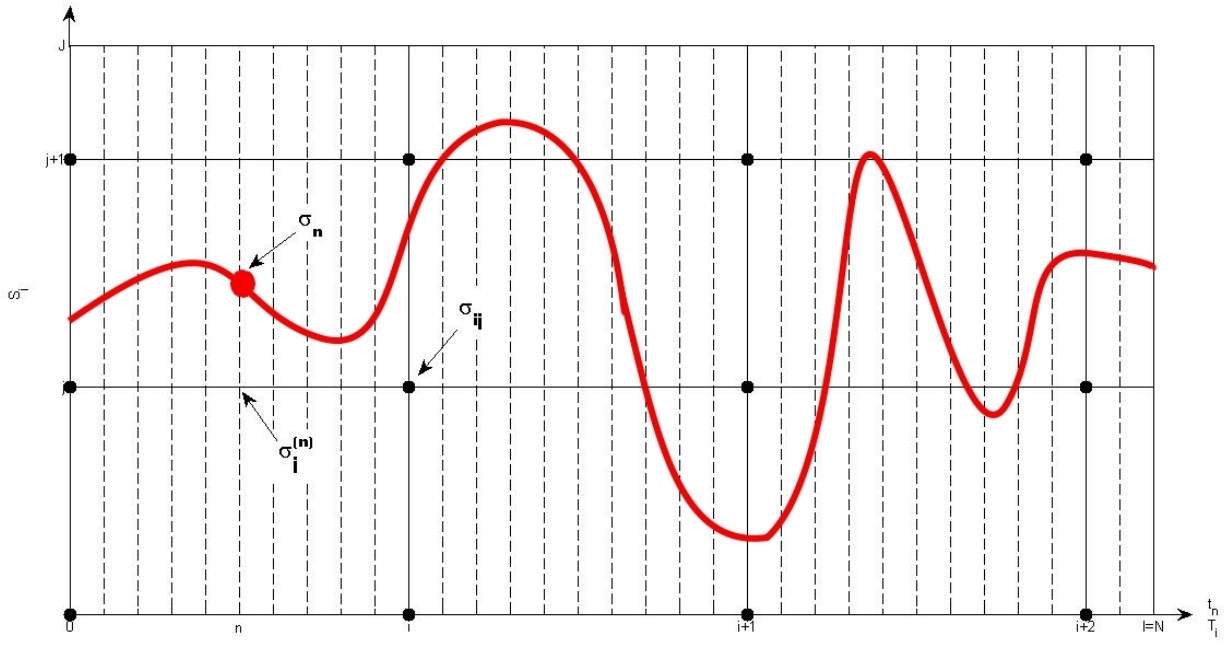

Figure 2.1: An illustration of the grid setup. The red curve represents a sample path simulation. The arrows indicate what volatility values correspond to the specific values of S and t.

outside this grid. We deal with the issue by considering the volatility surface to be constant for all stock prices outside our grid interval $[S_{min}, S_{max}]$, namely if at time t_n the stock price $S_n > S_{max}$ then $\sigma_n = \sigma(S_n, t_n) = \sigma(S_{max}, t_n) = \sigma_J^{(n)}$. This means that our interpolation relationships over stock price also hold for any points simulated outside the grid, since $\sigma_{J+1}^{(n)} = \sigma_J^{(n)}$, which in return means we do not have to deal with the boundaries as a special case within our source code. If we consider that our grid encompasses stock prices that are up to 3 standard deviations away from our current stock price, we can justify taking a constant volatility for stock price values outside the grid by claiming that the market does not have a strong view on volatility in areas so far away from today's price due to the fact that such a situation occurring is highly unlikely. Therefore the market will assume that the volatility will be approximately the same as the closest known volatility value. This argument is even more pertinent when we consider that a local volatility model in itself is used to make short-term forecasts and thus the probability of a large swing in price in the short-term is even less probable.

2.3 Forward mode

Now that we have set up our grid, we can move on to doing a forward mode computation of the sensitivities. Again using the standard notation in AD research, our goal is to obtain the value for $\dot{P} = \frac{\partial P}{\partial \sigma_{i,j}}$ for $i = 1...I, j = 1...J$. Let us compute the sensitivity with respect to one of the nodes $\sigma_{i,j}$. To do so, we will need to compute a chain of values that will eventually lead us to finding $\dot{P}$:

$$\dot{\sigma}_{i,j} \rightarrow \dot{\sigma}_j^{(n)} \rightarrow \dot{\sigma}_n \rightarrow \dot{S}_n \rightarrow \dot{P}$$

To deduce the form of this chain for the dot values, we must look at the functional relations we are given for the original variables. Since we know what variable we want to find the derivative of, we can work backwards by investigating what inputs are required to compute the original variable, and then continue to look further back at functional relations between variables until we come across a value whose derivative we can evaluate without any further information. So e.g. we know that to compute the payoff, we need to know the final stock price, but to obtain it, we need to know the volatility term driving the SDE and so on.
Let us work through the computations. To start, we know that obviously $\dot{\sigma}_{i,j} = 1$. We then continue by looking at $\dot{\sigma}_j^{(n)}$ at every timestep. We can deduce this by

differentiating the interpolation relation to obtain

$$\dot{\sigma}_j^{(n)} = \dot{\sigma}_{i,j} + \frac{t_n - T_i}{T_{i+1} - T_i} \left(\dot{\sigma}_{i+1,j} - \dot{\sigma}_{i,j}\right) = \dot{\sigma}_{i,j} + \beta(n) \left(0 - \dot{\sigma}_{i,j}\right) = (1 - \beta(n))\dot{\sigma}_{i,j}.$$

Now that we have $\dot{\sigma}_j^{(n)}$, we can compute $\dot{\sigma}_n$. Again we use the interpolation relation, but this time we must be careful since the interpolating coefficient depends on stock price S_n, which has a non-zero derivative with respect to $\sigma_{i,j}$. Hence we obtain the relation

$$\begin{aligned} \dot{\sigma}_n &= \dot{\sigma}_j^{(n)} + \alpha(S_n)\left(\dot{\sigma}_{j+1}^{(n)} - \dot{\sigma}_j^{(n)}\right) + \dot{\alpha}(S_n)\left(\sigma_{j+1}^{(n)} - \sigma_j^{(n)}\right) \\ &= \dot{\sigma}_j^{(n)} + \alpha(S_n)\left(\dot{\sigma}_{j+1}^{(n)} - \dot{\sigma}_j^{(n)}\right) + \frac{\dot{S}_n}{\Delta S}\left(\sigma_{j+1}^{(n)} - \sigma_j^{(n)}\right), \end{aligned}$$

where $\Delta S = S_{j+1} - S_j$. Finally, since we know that $\dot{P} = \frac{\partial P}{\partial S_N}\dot{S}_N$ and we know the derivative of the payoff with respect to the stock price at maturity, all we need to find is a recurrence relationship for $\dot{S}_n$ to obtain its value at time t_N. This only requires the standard step of differentiating the Euler scheme, paying attention to the fact that $\sigma_n = \sigma(S_n, t_n)$ depends on the stock price:

$$\dot{S}_{n+1} = \dot{S}_n \left(1 + r\Delta t + \sigma(S_n, t_n)\Delta W_n\right) + S_n \Delta W_n \dot{\sigma}(S_n, t_n).$$

Now we have all we need to run the algorithm in forward mode and compute the sensitivity of the option price $V = e^{-rT}\mathbb{E}\left[P(S(T))\right]$ with respect to the value of a single local volatility node $\sigma_{i,j}$. The problem here lies in the fact that we want to compute the sensitivity with respect to all the nodes, and this would prove rather costly, if for each sensitivity we would run e.g. 10 000 Monte Carlo path simulations to get an estimate with a reasonable variance. For a grid of size 20×20 this would lead to having to simulate 4 million paths. Therefore we shall look for a more effective way to find the sensitivities.

2.4 Adjoint mode

For the adjoint algorithm, we will go through the flow of information in reverse order, calculating all the intermediate variables in the order

$$\bar{\sigma}_{i,j} \leftarrow \bar{\sigma}_j^{(n)} \leftarrow \bar{\sigma}_n \leftarrow \bar{S}_n \leftarrow \bar{P},$$

starting with the value we know, $\bar{P} = \frac{\partial P}{\partial P} = 1$. We then compute a recursive relationship allowing us to evaluate $\bar{S}_n$ similarly as in the simple case of standard geometric

Brownian motion:

$$\begin{aligned}\bar{S}_n &= \left(\frac{\partial P}{\partial S_n}\right)^T = \left(\frac{\partial S_{n+1}}{\partial S_n}\right)^T \bar{S}_{n+1} \\ &= \left((1 + rh + \sigma_n \Delta W_n) + S_n \Delta W_n \frac{1}{\Delta S}(\sigma_{j+1}^{(n)} - \sigma_j^{(n)})\right)^T \bar{S}_{n+1} \\ &= (D(n) + B(n)A(n))^T \bar{S}_{n+1},\end{aligned}$$

where $D(n) = 1 + rh + \sigma_n \Delta W_n$ and $B(n) = \Delta W_n S_n$ as in the case of geometric Brownian motion, and $A(n) = \frac{1}{\Delta S}\left(\sigma_{j+1}^{(n)} - \sigma_j^{(n)}\right)$. We initialize this recursion with a value we know: $\bar{S}_N = \left(\frac{\partial P}{\partial S_N}\right)^T$, since this is just the derivative of the payoff function. From this we can compute $\bar{\sigma}_n$, which will then give us access to the bar values in the interpolation relationships:

$$\bar{\sigma}_n = \left(\frac{\partial P}{\partial \sigma_n}\right)^T = \left(\frac{\partial S_{n+1}}{\partial \sigma_n}\right)^T \bar{S}_{n+1} = (S_n \Delta W_n)^T \bar{S}_{n+1}.$$

Now we can move on to computing derivatives with respect to the partially interpolated values $\bar{\sigma}_j^{(n)}$. We must be careful, however to update the respective values for both intermediate spline nodes at stock prices S_j, S_{j+1}, since both of these are used to compute the final σ_n. We derive the relationship analogically to the previous ones to find that

$$\begin{aligned}\bar{\sigma}_j^{(n)} &= \left(\frac{\partial P}{\partial \sigma_j^{(n)}}\right)^T = \left(\frac{\partial \sigma_n}{\partial \sigma_j^{(n)}}\right)^T \bar{\sigma}_n = (1 - \alpha(S_n))^T \bar{\sigma}_n, \\ \bar{\sigma}_{j+1}^{(n)} &= \left(\frac{\partial P}{\partial \sigma_{j+1}^{(n)}}\right)^T = \left(\frac{\partial \sigma_n}{\partial \sigma_{j+1}^{(n)}}\right)^T \bar{\sigma}_n = \alpha(S_n)^T \bar{\sigma}_n.\end{aligned}$$

We must individually handle the special case when the stock price drifts beyond our grid, i.e. $j > J$. In that case we know that $\sigma_n = \sigma_J^{(n)}$, since σ_n is constant beyond the grid, and so the corresponding bar relationship is

$$\bar{\sigma}_J^{(n)} = \bar{\sigma}_n$$

Having these intermediate interpolation sensitivities, we can finally compute all the node sensitivities we want. At this point we must be careful, because unlike all the previous sensitivities, those at the nodes have a cumulative character. Up to now, each of the sensitivities was unique for the given time-step t_n and its value didn't change at any other time-step before or after t_n. But the volatility surface node $\sigma_{i,j}$ is needed for interpolating the values $\sigma_j^{(n)}$ for all time-steps $t_n \in [T_i, T_{i+1}]$. Moreover, at every

time step we compute both $\sigma_j^{(n)}$ and $\sigma_{j+1}^{(n)}$, but each of these requires two spline nodes $\sigma_{i,j}, \sigma_{i+1,j}$ and $\sigma_{i,j+1}, \sigma_{i+1,j+1}$ respectively, so then in the adjoint computations, all four node sensitivities will be influenced. To simplify notation, we shall use notation used in programming for incrementing values: $a = a + b \Leftrightarrow a{+}{=}b$. With that established, we can compute the final values we are looking for:

$$\bar{\sigma}_{i,j} += (1 - \beta(n))^T \bar{\sigma_j}^{(n)} \qquad \bar{\sigma}_{i,j+1} += (1 - \beta(n))^T \bar{\sigma}_{j+1}^{(n)}$$
$$\bar{\sigma}_{i+1,j} += \beta(n)^T \bar{\sigma_j}^{(n)} \qquad \bar{\sigma}_{i+1,j+1} += \beta(n)^T \bar{\sigma}_{j+1}^{(n)}$$

Thus we have, through a series of backward calculations, obtained the sensitivities we are looking for. Let us note that we do not need to loop through all $j = 1 \dots J$ and $i = 1 \dots I$ for each time-step but only need to update the value of those indices i, j to which the simulated stock price S_n belongs. These indices are uniquely defined, i.e. there exists no ambiguity as to which part of the local volatility surface a given stock price belongs to.

In comparison to forward mode, we only need one set of e.g. 10 000 Monte Carlo path simulations to obtain estimates of all the sensitivities, not just one, with a reasonable variance. That results in a considerable saving in computational effort. On the other hand, we make greater demands on the amount of available memory: for the computations we need to store all the intermediate variables that are generated in the path simulation and are then later required for the adjoint sensitivity computation. Specifically, we need to save all the interpolation coefficients $\beta(n), \alpha(S_n)$; the local volatility indices i, j that correspond to the stock price at time t_n; the coefficients $D(n)$ and $B(n)$, which remain unchanged from our analysis of standard geometric Brownian motion; and finally, we need to store coefficient $A(n)$, which is also to be found in the recursion for determining $\bar{S}_n$.

2.5 Handling other types of interpolation using 'binning'

We have deduced the results for linear interpolation in time and stock price, but we must also discuss the generalization to other types of interpolation methods, since this is not straightforward. Let us consider the example of a linear interpolation over time, but a natural cubic spline interpolation over stock price. To create the cubic spline over stock price at each time-step t_n, we have to use all the time-interpolated volatility values $\sigma_j^{(n)}, j = 1...J$, not just those adjacent to the currently simulated stock price. More importantly, doing the cubic spline interpolation is quite costly computationally,

since it involves solving a potentially large system of linear equations, and thus we want to avoid having to solve its adjoint counterpart, which is potentially 4 times as costly, for every path simulation. Viewed generally, it is a costly preprocessing procedure that has to be done before time-stepping that we want to avoid doing repeatedly in adjoint mode. Therefore, at this point it is optimal for us to use the 'binning' technique described in Chapter 1. For each of the N_B bins of path simulations, we compute the average value for $\bar{\sigma}_n, n = 1...N$, then we use the adjoint cubic spline algorithm over these average values and continue on to do the standard bar computations for the time interpolation. Doing so, we obtain N_B estimates of average local volatility surface sensitivities, which thus provide us with an estimator and confidence interval for the local volatility surface sensitivities we are looking for.

Chapter 3

Numerical results and conclusions

In the previous chapter we fully described how to implement the forward and adjoint mode algorithm to compute sensitivities of a local volatility surface. The MATLAB code in the appendix provides an implementation of this algorithm. Having the algorithm at hand allows us to analyze numerical results and verify the scale of the savings and any potential pitfalls the adjoint mode might have. All the numerical tests were done on a computer with a 2.2GHz Intel Core2Duo processor.

3.1 Numerical results

We firstly compare the thing we are most interested in - are we truly saving computational power? If so, how much? We hope for results similar to Giles and Glasserman [2006]. And indeed, if we run the analysis, we obtain such results. As we can see in figure 3.1, adjoint mode provides enormous savings as we scale the problem being solved. Whereas the time it takes to do computations in forward mode is linearly proportional to the number of nodes being processed, the adjoint method is near to constant in comparison. The adjoint method provides speed that must satisfy even the most demanding and impatient of users: even when scaled to unrealistic dimensions, with a grid size of 100×100 and $n = 1000$ time-steps for each of the $M = 10000$ simulated paths, the adjoint algorithm managed to do the computations in 22 seconds, which is still reasonable. For any realistic size of the problem, the adjoint method manages to handle the computations within under 5 seconds.

It is now important to make sure that the results we are getting fully correspond to forward mode and that we do not obtain any instabilities in the results. We will test both modes on the local volatility surface shown in figure 3.2, a surface that drops away to zero exponentially in time. This surface exhibits the properties of real-world local volatility surfaces that are obtained after lengthy calibration and smoothing. We

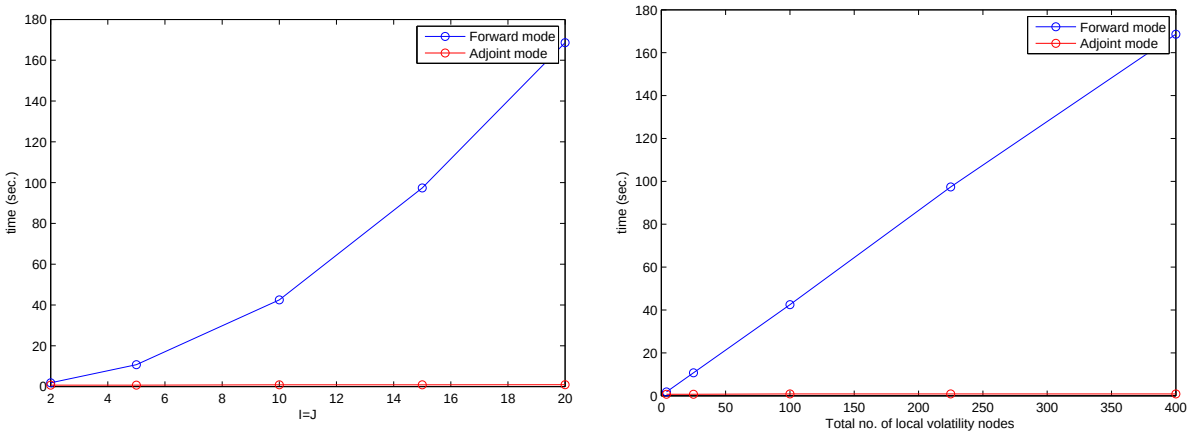

Figure 3.1: Comparison of time complexity of adjoint and forward mode. The left figure compares computation times with the size of the square grids with variable grid size $I = J$. The right figure compares the times with the total number of local volatility nodes. Run with 10 000 sample paths and 100 time-steps for the Monte Carlo simulation.

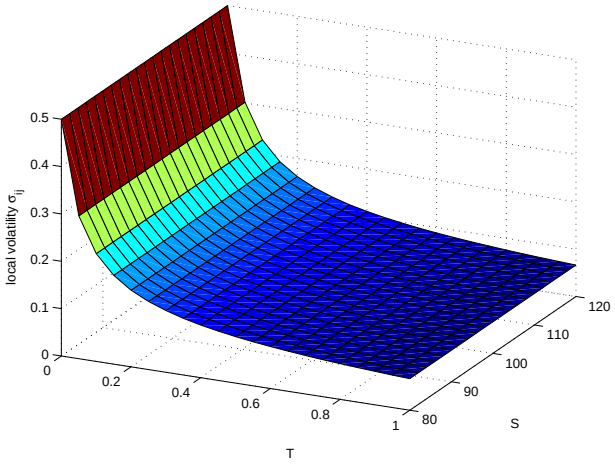

Figure 3.2: The testing local volatility surface.

also tested our source code on different volatility surfaces, but the results presented here relate to that in figure 3.2.

Figure 3.3 shows us that we are obtaining surfaces and confidence intervals that are extremely similar in appearance, as we would hope. The spike in the confidence interval at the upper stock price boundary can be attributed to the fact that many sample paths drift beyond the upper boundary and hence the confidence interval is of a cumulative nature.

To make sure that these surfaces indeed have the same values, we can look at the error surface in figure 3.4 showing us the difference between the forward and adjoint method results. With the error between the two methods being of order 10^{-16}, we see that the error is not only definitely within reasonable bounds in comparison to the Monte Carlo estimator confidence interval, but also barely detectable in terms of machine accuracy for 64-bit double precision. Thus adjoint mode can fully replace any forward mode calculations.

We also ran the adjoint algorithm with a much larger number of sample paths (1 million of them) to investigate the improvement obtained on the Monte Carlo confidence interval. Comparing results for the adjoint method from figure 3.3 for 10 000 paths

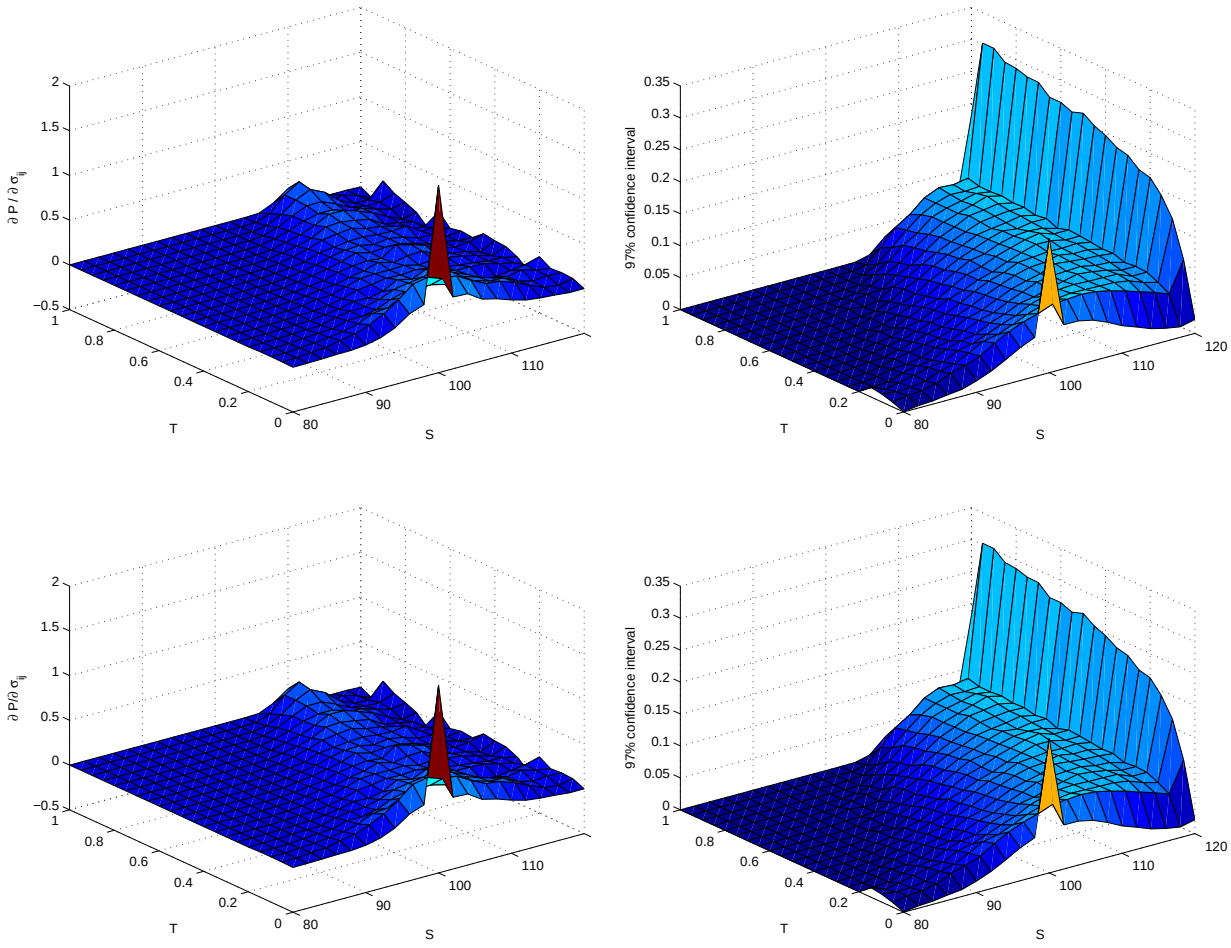

Figure 3.3: Forward and adjoint mode results for $n = 100$ time-steps, $M = 10000$ paths, grid size $I = J = 20$, risk-free rate $r = 5\%$. The top results are for forward mode, the lower for adjoint mode. The figures on the right depict the confidence intervals of the estimators. Sensitivities computed for a local volatility surface depicted in 3.2.

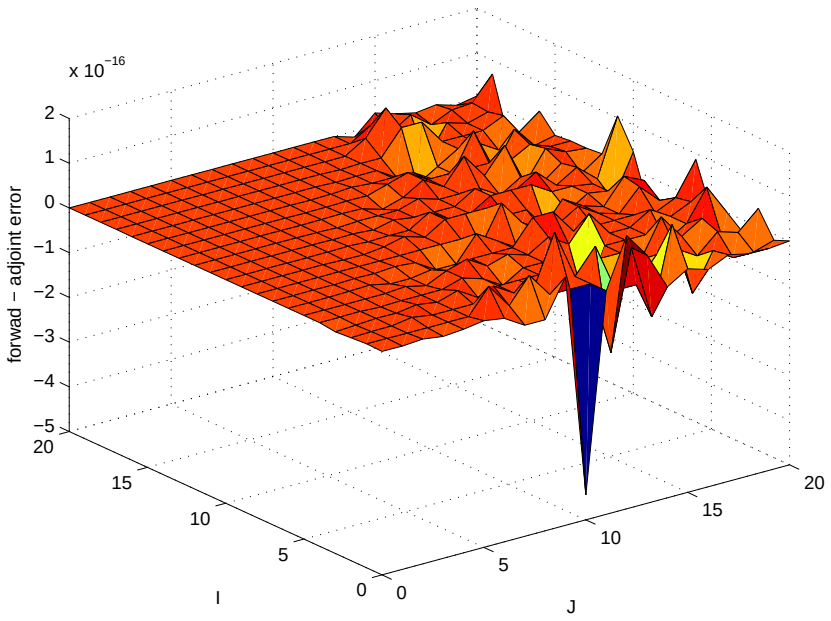

Figure 3.4: Error between adjoint method and forward mode surfaces. $n = 100$, $M = 10000$, $I = J = 20$

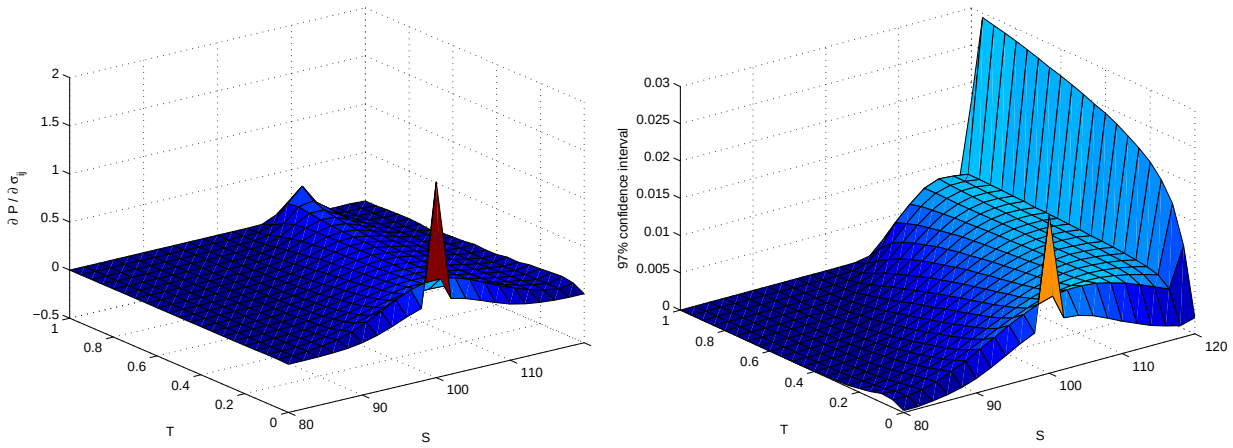

Figure 3.5: Adjoint method results for $n = 100$, $M = 10^6$, $I = J = 20$

and the results in figure 3.5 for 1 000 000 paths, we can observe that increasing the number of paths 100-fold only results in confidence interval improvements of factor 10, moreover at a considerable cost, since computational effort grows linearly in the number of paths and thus even computations that originally took less than a second can take considerable time to do a hundred times over. It is only advisable to use an extreme amount of samples in cases where superior precision is absolutely necessary.

3.2 Code validation

We have established that adjoint and forward mode give us the same results. To validate these results and make sure that both forward and adjoint mode are giving us the true values of the derivatives, we use the complex variable trick (CVT) introduced in Chapter 1. In terms of our path-wise sensitivity calculations, the CVT asks us to introduce a complex part to the local volatility node with respect to which we want to differentiate, i.e. $\sigma_{ij} := \sigma_{ij} + ih$. If we do so and follow the propagation of the complex part over the time-stepping recursion for stock price and then transform that according to the payoff, we finally obtain the value we require: $\dot{P} = \bar{\sigma}_{ij} = \frac{Im(P)}{h} + \mathcal{O}(h^3)$. We use this relation to check that we compute the correct dot and bar values for every individual simulated path. The maximum error we obtained across all the simulations done was of order 10^{-16}. Thus we consider the adjoint and forward mode computations to be exact and reliable.

3.3 Conclusions

From our application on a local volatility surface, we conclude that the adjoint method confirmed its immense efficiency when it comes to computing the sensitivities of one output with respect to a large number of inputs. More importantly, the application shows us how intertwined the forward and adjoint mode computations are and thus how straightforward and simple it is to derive the bar relations once we know the dot relations. We believe this is a general property of adjoint computations and in our specific application applies to any interpolating technique we might choose, not only to simple linear interpolation used in our case. This is important in the light of the fact that there exists software that automatically generates the required adjoint code, but results in Giles [2007] show that this is computationally significantly less efficient than when coded by hand. Therefore the simplicity of deriving the adjoint algorithm by hand for a single application gives us incentive to do so and save us time and computational effort later. Considering the black box handling described in Chapter 1, we believe this upholds even in the case of the modular data and algorithm infrastructure that computational systems in the derivatives industry incline towards.

Appendix A

The source codes

Here we provide the full source code to both the forward mode and adjoint mode computations of the local volatility surface sensitivities. When possible, the dot variables are denoted by $d<VarName>$, in cases where this would be confusing the notation $<VarName>dot$ is used. The notation for bar variables is simply noted as $<VarName>bar$.

A.1 Forward mode code

```
% Calculating sensitivites of local vol. surface of a EU call using forward
% technique
% Use Euler-Maruyama method and Pathwise Sensitivity Method to
% estimate sensitivity of call option to spline points of local vol.
% surface
% Test problem:   dS   = r*S dt + sig*S dW
% Indices iT, jT select with respect to which spline node
% we calculate sensitivity
function [Pdot,PdotVar]=forward_mode(iT,jT)

% problem parameters
r    = 0.05;
sig0 = 0.5;
T    = 1;
S0   = 100;
K    = 110;

%parameters for local vol grid
```

```
J = 20;
I = 20; %I < N

% Monte Carlo simulation parameters
M  = 1e+4;   % total number of Monte Carlo paths
M2 = 1e+3;   % number of paths at a time
N  = 100;
h  = T/N;

% local vol surface grid setup
Smin = 0.8*S0;
Smax = 1.2*S0;
dS   = (Smax - Smin)/J;
dT   = T/I;

%our testing local vol. surface
LocalVolSurf = ones(I+1,J+1)*sig0;
for i=1:I+1
    LocalVolSurf(i,:) = LocalVolSurf(i,:)*((I+1)*((i^(-1)+i^(-0.5))/2)/(I+1));
end

%set up derivative operator matrix
dLocalVolSurf = zeros(I+1,J+1);
dLocalVolSurf(iT,jT) = 1;

%iIndex - integer from interval [0,I]
iIndex = floor([1:N]*h./dT);
%beta - time interpolation parameter
beta = ([1:N].*h-iIndex([1:N]).*dT)./dT;
iIndex = [0,iIndex];
beta = [0,beta];

%setting up interpolation over time
sigJ = zeros(N+1,J+1);
dSigJ = zeros(N+1,J+1);
for n=0:N
```

```
    sigJ(n+1,:) = LocalVolSurf(1+iIndex(1+n),:) + beta(1+n) * (...
    LocalVolSurf(min(1+iIndex(1+n)+1,I+1),:) - LocalVolSurf(1+iIndex(1+n),:));
end
%the corresponding dot spline
for n=0:N
    dSigJ(n+1,:) = dLocalVolSurf(1+iIndex(1+n),:) + beta(1+n) * (...
    dLocalVolSurf(min(1+iIndex(1+n)+1,I+1),:) - dLocalVolSurf(1+iIndex(1+n),:));
end

Pdot = 0;
PdotVar = 0;
%main MC sampling loop
for m = 1:M2:M
    m2 = min(M2,M-m+1);

    S = S0*ones(1,m2);
    D = zeros(N,m2);
    B = zeros(N,m2);
    alpha = zeros(N+1,m2);
    jIndex = zeros(N+1,m2);
    A = zeros(N+1,m2);
    Sdot = zeros(1,m2);
    for n = 1:N
      dW = sqrt(h)*randn(1,m2);
      %jIndex - integer from interval [0,J] - indicates which block S is in
      jIndex(n,:) = max( min(J,floor(real((S-Smin)./dS))) ,0);
      %alpha - stock price interpolation parameter
      alpha(n,:) = (S - (Smin + jIndex(n,:).*dS) )./dS;
      %linear interpolation in stock price
      sig = sigJ(n,1+jIndex(n,:)) + ...
      alpha(n,:).*(sigJ(n,min(J+1,1+jIndex(n,:)+1)) - sigJ(n,1+jIndex(n,:)) );
      D(n,:) = 1+r*h+sig.*dW;
      %forward mode
      A(n,:) = (1/dS).*(sigJ(n,min(J+1,1+jIndex(n,:)+1)) -...
      sigJ(n,1+jIndex(n,:)) );
      %linear interpolation of dSigJ
```

```
      dSig = dSigJ(n,1+jIndex(n,:)) + ...
      alpha(n,:).*(dSigJ(n,min(J+1,1+jIndex(n,:)+1)) -...
       dSigJ(n,1+jIndex(n,:)) ) + A(n,:).*s;
      %forward mode iteration
      B(n,:) = S.*dW;
      Sdot = D(n,:).*Sdot + B(n,:).*dSig;
      %timestep
      S  = S.*D(n,:);
    end
    for i = 1:m2
       if(S(i)>K)
           Sdot(i) = exp(-r*T)*s(i);
       else
           Sdot(i) = 0;
       end
    end
    Pdot = Pdot + sum(Sdot);
    PdotVar = PdotVar + sum(Sdot.^2);
end

Pdot = Pdot./M;
PdotVar = 3*sqrt( (M/(M-1))*(PdotVar./M - Pdot.^2) )/sqrt(M);
end
```

A.2 Adjoint mode code

```
%
% Calculating sensitivites of local vol. surface of a EU call using adjoint
% technique
% Use Euler-Maruyama method and Pathwise Sensitivity Method to
% estimate sensitivity of call option to spline points of local vol.
% surface
% Test problem:   dS   = r*S dt + sig*S dW
%
function [LocalSens,LocalSensConf]=adjoint_mode
```

```
% problem parameters
r = 0.05;
sig0 = 0.5;
T    = 1;
S0   = 100;
K    = 110;

%parameters for local vol grid
J = 20;
I = 20; %I < N

% Monte Carlo simulation parameters
M  = 1e+6;   % total number of Monte Carlo paths
M2 = 1e+4;   % number of paths at a time
N  = 100;
h  = T/N;

% local vol surface grid setup
Smin = 0.8*S0;
Smax = 1.2*S0;
dS   = (Smax - Smin)/J;
dT   = T/I;
%set up our testing local vol surface
LocalVolSurf = ones(I+1,J+1)*sig0;
for i=1:I+1
    LocalVolSurf(i,:) = LocalVolSurf(i,:)*((I+1)*((i^(-1)+i^(-0.5))/2)/(I+1));
end

%set up variable for storing sensitivities
LocalSens = zeros(I+1,J+1); %sensitivities
LocalSensConf = zeros(I+1,J+1); %confidence interval for MC sampling

%iIndex - integer from interval [0,I]
%beta - time interpolation parameter
iIndex = floor([1:N]*h./dT);
beta = ([1:N].*h-iIndex([1:N]).*dT)./dT;
```

```
iIndex = [0,iIndex];
beta = [0,beta];

%setting up interpolation over time
sigJ = zeros(N+1,J+1);
for n=0:N
    sigJ(n+1,:) = LocalVolSurf(1+iIndex(1+n),:) + beta(1+n) * (...
    LocalVolSurf(min(1+iIndex(1+n)+1,I+1),:) - LocalVolSurf(1+iIndex(1+n),:));
end

%main MC sampling loop
for m = 1:M2:M
    m2 = min(M2,M-m+1);

    S = S0*ones(1,m2);
    D = zeros(N,m2);
    B = zeros(N,m2);
    alpha = zeros(N+1,m2);
    jIndex = zeros(N+1,m2);
    A = zeros(N+1,m2);

    for n = 1:N
      dW = sqrt(h)*randn(1,m2);
      %jIndex - integer from interval [0,J] - indicates which block S is in
      jIndex(n,:) = max( min(J,floor(real((S-Smin)./dS))) ,0);
      %alpha - stock price interpolation parameter
      alpha(n,:) = (S - (Smin + jIndex(n,:).*dS) )./dS;
      %linear interpolation in stock price
      sig = sigJ(n,1+jIndex(n,:)) + ...
      alpha(n,:).*(sigJ(n,min(J+1,1+jIndex(n,:)+1)) - sigJ(n,1+jIndex(n,:)) );
      %storing necessary variables
      D(n,:) = 1+r*h+sig.*dW;
      A(n,:) = (1/dS).*(sigJ(n,min(J+1,1+jIndex(n,:)+1)) - ...
      sigJ(n,1+jIndex(n,:)) );
      B(n,:) = S.*dW;
      %timestep
```

```
  S  = S.*D(n,:);
end
jIndex(N+1,:) = max( min(J,floor(real((S-Smin)./dS))) ,0);
alpha(N+1,:) = (S - (Smin + jIndex(N+1,:).*dS) )./dS;

%-------------------------ADJOINT METHOD-----------------------------%
%setting Sbar(N) - recursion for calculating dP/dS0
Sbar = zeros(1,m2);
for i = 1:m2
   if(S(i)>K)
       Sbar(i) = exp(-r*T);
   else
       Sbar(i) = 0;
   end
end

sigbarJ = zeros(N+1,J+1,m2);
sigbarIJ = zeros(I+1,J+1,m2);
for n=0:(N-1)
   %recurrence for sigbar, Sbar
   sigbar = Sbar.*B(N-n,:);
   Sbar = (D(N-n,:)+B(N-n,:).*A(N-n,:)).*Sbar;

   %sigbarJs
   for k=1:m2
       j = 1 + jIndex(N-n,k);%in [1,J+1]
       if(j<J+1)
          sigbarJ(N-n,j,k)= (1-alpha(N-n,k))*sigbar(k);
          sigbarJ(N-n,j+1,k) = alpha(N-n,k)*sigbar(k);
       else
           sigbarJ(N-n,j,k) = sigbar(k);
       end
   end

   %sigbarIJs
   for k=1:m2
```

```
            i = 1 + iIndex(N-n);
            j = 1+jIndex(N-n,k);
            sigbarIJ(i,j,k) = sigbarIJ(i,j,k) + (1-beta(N-n)).*sigbarJ(N-n,j,k);
            if(j<J+1)
              sigbarIJ(i,j+1,k) = ...
              sigbarIJ(i,j+1,k) + (1-beta(N-n)).*sigbarJ(N-n,j+1,k);
                if(i<I+1)
                    sigbarIJ(i+1,j+1,k) = ...
                    sigbarIJ(i+1,j+1,k) + beta(N-n).*sigbarJ(N-n,j+1,k);
                end
            end
            if(i<I+1)
              sigbarIJ(i+1,j,k) = ...
              sigbarIJ(i+1,j,k) + beta(N-n).*sigbarJ(N-n,j,k);
            end
        end
    end

    LocalSens = LocalSens + sum(sigbarIJ,3);
    LocalSensConf = LocalSensConf + sum(sigbarIJ.^2,3);

end

LocalSens = LocalSens./M;
LocalSensConf = (LocalSensConf./M - LocalSens.^2);
LocalSensConf = 3*sqrt((M/(M-1)) .* LocalSensConf)./sqrt(M);
end
```

References

Y. Achdou and O. Pironneau. *Computational Methods for Option Pricing*. Society for Industrial and Applied Mathematics, 1st edition, 2005.

M. Avellaneda, C. Friedman, R. Holmes, and D. Samperi. Calibrating volatility surfaces via relative-entropy minimization. *Applied mathematical finance*, 4(1):37–64, 1997.

L. Capriotti and M.B. Giles. Fast correlation greeks by adjoint algorithmic differentiation. *RISK*, March 2010.

S. Crepey. Calibration of the local volatility in a trinomial tree using tikhonov regularization. *Journal on Mathematical Analysis*, 34(5):1183–1206, 2003.

N. Denson and M. Joshi. Fast and accurate greeks for the libor market model. Technical report, University of Melbourne, August 2009a. URL `http://ssrn.com/paper=1448333`.

N. Denson and M. Joshi. Flaming logs. *Wilmott Journal*, 1:259–262, May 2009b. URL `http://www.markjoshi.com/downloads/index.htm`.

E. Derman and I. Kani. The volatility smile and its implied tree. Technical report, Goldman Sachs, January 1994. Global Derivatives Quarterly Review.

B. Dupire. Pricing with a smile. *RISK*, January 1994.

M.B. Giles. Monte carlo evaluation of sensitivities in computational finance. Technical report, University of Oxford, 2007.

M.B. Giles and P. Glasserman. Smoking adjoints: Fast monte carlo greeks. *RISK*, January 2006.

P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Stochastic Modelling and Applied Probability, Vol.53. Springer-Verlag, 2003.

A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, 2nd edition, 2008.

M. Hanke and E. Rosler. Computation of local volatilities from regularized dupire equations. *International Journal of Theoretical and Applied Finance*, 8(2):207–222, 2005.

C. Kaebe, J.H. Maruhn, and E.W. Sachs. Adjoint-based monte carlo calibration of financial market models. *Finance and Stochastics*, 13(3):351–379, September 2009.

M. Leclerc, Q. Liang, and I. Schneider. Fast monte carlo bermudan greeks. *RISK*, July 2009.

M. Rubinstein. Implied binomial trees. *Journal of Finance*, 49(3):771–818, January 1994.

D. Samperi. Calibrating a diffusion pricing model with uncertain volatility: Regularization and stability. *Mathematical finance*, 12(1):71–87, 2002.

W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40:110–112, 1998.