Imperial College of Science, Technology
and Medicine

Department of Mathematics

Proxy Scheme and Automatic Differentiation: Computing faster Greeks in Monte Carlo simulations

Blandine Stehlé
CID: 00613966

September 2010

Submitted to Imperial College London in fulfilment of the requirements for the
Degree of Master of Science in Mathematics and Finance

Abstract

Most, if not all, major financial institutions have trading operations which stand prepared to write complex contingent claims, usually termed exotic options, with almost arbitrary payoffs, on multiple underlying asset classes on demand for their customers.

With the aim being to minimise the time taken to quote prices to customers, the pricing methodology usually consists of an object-orientated implementation of a Monte Carlo simulation together with an option payoff specified by a scripting language which uses parsing technology to allow the trader to specify an (almost) arbitrary payoff external to the Monte Carlo simulation engine. This results in a very generic and flexible methodology for pricing the exotic option in question.

However, in practice, obtaining good quality Greeks (ie sensitivities of the price of the exotic option with respect to input parameters) for hedging and risk-management purposes is at least as important as obtaining an accurate price. Having an extremely generic and flexible pricing methodology is not useful if there is not a generic and flexible methodology for computing Greeks. Obtaining Greeks by a "bump-and-revalue" methodology is generic and simple to implement but is well-known to be slow and often yields inaccurate Greeks.

In this dissertation, we consider and analyse two other potentially very generic methodologies for obtaining Greeks. These methodologies are the "partial proxy scheme" and the "pathwise derivative method" combined with Automatic Differentiation. The latter methodology can be implemented either in "forward mode" or in "adjoint mode". We describe these methodologies in detail and give numerical examples which compare their performances (from the point of view of both timing and accuracy) with a bump-and-revalue methodology. We discuss the use of Automatic Differentiation software to compute generic pathwise derivatives. This approach yields very real advantages since it obviates the need to write any new code whatsoever to obtain Greeks. We conclude that the pathwise derivative method, combined with Automatic Differentiation, seems to be the best methodology for obtaining Greeks taking into account computation times, accuracy and ease of generic implementation.

Acknowledgements

Special thanks to John Crosby, my project supervisor, for his dedicated guidance, support and invaluable suggestions throughout this entire project. I am deeply grateful to him for his involvement and his help in both the implementation and this dissertation.

Sincere thanks to Mark Davis, my project supervisor at Imperial College, for his advice.

Many thanks to all my Imperial College lecturers, from whom I gained invaluable knowledge and without which this thesis would not have been possible.

Many thanks to all the members of the front-office quantitative analytics team at UBS for their welcome.

Contents

1 Introduction

This dissertation examines and compares different methodologies for obtaining Greeks (ie partial derivatives) in Monte Carlo simulations.

The vast majority of major financial institutions have trading and sales operations which stand prepared to write complex contingent claims, usually termed exotic options, with almost arbitrary payoffs, on multiple underlying asset classes. The exotic options might have payoffs reflecting a very specific view that an investor has on the future price performance of, for example, several underlying assets such as stock indices (which may, additionally, for example, be quoted in different currencies).

Alternatively, they might be linked to the desire of a corporation to buy assets or securities which closely match its future liabilities or which provide a (possibly, partial) hedge against the cost of the corporation's raw materials moving in an adverse fashion. Clearly, well-known financial options such as asian (average-rate) options, chooser options, "best-of" options, spread options on the relative performance of the prices of two assets or commodities all fall into the general class of what might be termed exotic options.

However, once one allows for different payoffs and different underliers such as stocks, currencies, bonds, rates on interest-rate swaps, default events, rates on Credit Default Swaps, commodities, measures of realised variance and indices linked to property prices or measures of inflation, the range of possible exotic options is infinite.

From the view of, for example, an investor, when she has determined a particular view on the future price performance of particular assets, she may be very keen to act on this view as soon as reasonably possible else the market may move against her.
Taking this into account and taking into account the competitive nature of investment banking, it is clear that investment banks, to be successful in their options' trading operations, must minimise the time taken to quote, to their customers, the prices of exotic options.

With this in mind, investment banks have typically made a large investment in building a sophisticated general-purpose Monte Carlo engine, using object-orientated programming techniques. The engine is designed to be very flexible and can simulate, for example, multiple underlying assets of every conceivably possible type. Note that the engine is, necessarily, a Monte Carlo simulation since, for example, PDE solvers are not usually able to solve problems in the high-dimensionality that is implicit within many types of exotic options. The Monte Carlo simulation is typically implemented in C++. Given the sophistication of the implementation, new releases of the executable program containing the Monte Carlo engine may only be possible on a time-frame of several days to several weeks (to allow time to, not only, change the underlying source-code but also to build and run test-harnesses, link in with wider IT infrastructure, etc).

The Monte Carlo simulation engine links with a separate mechanism for specifying the payoff of the option. Traders need to be able to specify the payoff of the option within, perhaps, a few minutes. This is done by specifying the payoff using a scripting language (essentially, a mini-programming language - some such languages - for example, BOOST spirit - are widely available as freeware downloadable from the internet).

The Monte Carlo engine then links in the specified payoff at run-time, using parsing technology to interpret the exotic option payoff as a mathematical formula. This architecture has the desired effect of allowing for the pricing of exotic options, with almost arbitrary payoffs, at minimal cost in elapsed time.

However, once a particular trade has been done with a customer, the question immediately arises of obtaining Greeks (ie sensitivities of the price of the exotic option with respect to input parameters) for hedging and risk-management purposes - and this is typically a difficult question. Since the pricing methodology is optimised for flexibility and generality, any methodology for obtaining Greeks must be equally flexible.

Broadly speaking, methodologies for obtaining Greeks from Monte Carlo simulations fall into three categories: "bump-and-revalue", likelihood ratio methods and pathwise differentiation methods. All three of these methods are described in Glasserman (2004) [11] and in chapters 3 and 4 so we simply give the briefest description here.

"Bump-and-revalue" follows the naive but simple strategy of repeating the same simulation with different input parameters. A forward-finite difference for N_G Greeks would require performing the simulation $N_G + 1$ times - once to get the original price and once each with a perturbed input parameter.
A central finite-difference raises this to performing the simulation $2N_G + 1$ times. Clearly, this method will be slow. There are potentially other drawbacks to this method (see Glasserman (2004) [11]) when the exotic option has a discontinuous payoff.
On the positive side, bump-and-revalue is clearly simple to implement.

Likelihood ratio methods rely on differentiating the transition probability density function of the quantities underlying the simulation (such as log stock prices or log forward LIBOR rates). The likelihood ratio method can be a powerful and general method (for example, it does not require any assumption on the option payoff such as continuity) but it has two potential drawbacks. Firstly, the variance of the estimate of the Greek may be very high (see section 7.3.2 of Glasserman (2004) [11]) and, secondly, it requires the transition probability density function to be known in closed-form. In practice, the latter is often not known analytically. This lead Fries and Kampen (2006) [8] and Fries and Joshi (2006) [7] to suggest a variant on the likelihood ratio method, called the "partial proxy scheme", which does not need an analytical transition probability density function. We discuss this method at greater length in chapter 3. But briefly, the partial proxy scheme essentially replaces an analytical differentiation of the transition probability density function with a finite-difference differentiation which allows the possibility to combine the flexibility of finite-differences with some of the benefits of the likelihood ratio method.

Pathwise differentiation methods rely on differentiating (the discounted value of) the option payoff with respect to some input parameter (as opposed to likelihood ratio methods which essentially differentiate the transition probability density function).
Then the method essentially computes (by Monte Carlo simulation) the expectation of the derivative (of the discounted value) of the option payoff.
In order to be differentiable, we require that the option has a continuous payoff. In practice, many exotic options have discontinuous payoffs which appears to slightly limit the

applicability of the method. However, this is less of a restriction than it seems because it is nearly always possible to slightly change the option payoff so that it is continuous.

For example, consider a binary cash-or-nothing call option with strike K. Let the price of the underlying asset at maturity be S.
The option has payoff: $\mathbb{I}_{S \geq K}$ where $\mathbb{I}$ denotes the indicator function.
This payoff is clearly discontinuous. However, it can be replaced, for some small $h > 0$, by

$$\begin{cases} 1 & \text{if} \quad S \geq K + \frac{h}{2}, \\ 0 & \text{if} \quad S \leq K - \frac{h}{2}, \quad \text{and} \\ \frac{1}{h}\left(S - \left(K - \frac{h}{2}\right)\right) & \text{otherwise.} \end{cases}$$

This idea is, generically, referred to as a "ramp" and the value of h is referred to as the "ramp width". Clearly, there are potential drawbacks here: For any strictly positive h, we do change the option payoff (and therefore also the option price) and it leaves open the choice of a suitable value of h.

Nonetheless, this idea of replacing a discontinuous payoff by a continuous payoff is very generic (see Glasserman (2004) [11] and Giles (2007) [9] for examples for other types of options apart from binary cash-or-nothing options) and can easily be built into the payoff definition in the scripting language.
Since, it is very generic, we will, unless otherwise explicitly specified, assume that it has been done and assume that all payoffs are (or have been modified to be) continuous.

As part of the pathwise differentiation method, we need to differentiate the option payoff. For the case of, for example, a Vanilla call option, this is easily done analytically - it is the indicator function (see section 7.2 of Glasserman (2004) [11] for details).

For very complicated option payoffs, analytical differentiation is still possible but becomes tedious in extreme cases as it will likely involve a quantitative analyst mathematically differentiating the payoff and then coding it manually. Of course, we are motivated to find methods which are highly generic and require minimal implementation time. Fortunately, there is an alternative. This uses Automatic Differentiation (we will often abbreviate this to AD) software. This is software [18] which can be downloaded from the internet which can automatically compute analytically the derivative of any function (including an option payoff) whose functional form is specified (for example, in C++ or via a scripting language). We refer the reader to Giles (2007) [9] and Capriotti (2010) [4] for more background information on Automatic Differentiation.

Allied with ramps and Automatic Differentiation software, pathwise differentiation becomes a very flexible and powerful tool for computing Greeks. However, there is potentially even better news. In practice, a trader using a LIBOR Market Model with semi-annual LIBORs to price, say, a 30 year exotic interest-rate option may need to compute a delta (partial derivative) with respect to each LIBOR up to the maturity of the option (implying that there are 60 delta calculations) as well as a vega (partial derivative) with respect to the volatility of each LIBOR (implying that there are 60 vega calculations). In total, therefore, in this example (which is by no means untypical) the trader needs 120 partial derivatives. Giles and Glasserman (2006) [10] show how the pathwise differentiation method can be used either in "forward mode" or in "adjoint mode".

Very briefly, "forward mode" corresponds to the brief description above, i.e. it corresponds to the idea of computing derivatives via a chain rule from input parameters towards the output sensitiviy. With the aid of Automatic Differentiation software, all of this can be done automatically -the user does not need to add any new code.

The essence of the "adjoint mode" (see Giles and Glasserman (2006)[10], Capriotti and Giles (2010) [5], Capriotti (2010) [4] and Leclerc et al. (2009) [14]) is that it is a variant on the "forward mode" which turns out to be an incredibly efficient way of computing the partial derivatives of an option price with respect to large numbers of input parameters. Hence, it is extremely effective for the example given above for the case where it is necessary to compute 120 partial derivatives.

Automatic Differentiation can also be used with the adjoint mode. Again, the user does not need to add any new code. This is especially useful if someone was using the approach of Giles and Glasserman (2006) [10] without Automatic Differentiation. Giles and Glasserman (2006) [10] used a standard LIBOR market model without displaced diffusion. Suppose one wished to extend one's model to include displaced diffusion. Without Automatic Differentiation, this would involve writing, by hand, significant amounts of new code which would take time and possibly be a source of error. With Automatic Differentiation, the whole process is completely automated which saves development time and minimises the possibility of software bugs.

We give a full description of "forward mode" and "adjoint mode" in chapter 4. The rest of this dissertation is structured as follows:

In chapter 2, we briefly introduce the LIBOR Market Model.
In chapter 3, we discuss the partial proxy scheme.
In chapter 4, we discuss the use of Automatic Differentiation together with pathwise differentiation as well as describe the "forward mode" and "adjoint mode".
In chapter 5, we discuss implementational issues.
In chapter 6, we illustrate with a number of numerical examples.
Chapter 7 concludes.

2 The LIBOR Market Model

In this chapter, we briefly introduce the LIBOR market model (referred to as LMM) that we will use in almost all our future numerical examples.
For the presentation of this model, we will rely on Glasserman (2004) [11], chapter 3, p.166 and succeeding.

We have chosen the LMM to be the model principally used in our tests for several reasons: it is commonly used in practice, it has a non-trivial drift term and is, in general, a high dimensional model which makes Monte Carlo simulation the natural pricing methodology.

The LMM or BGM model was introduced by Brace, Gatarek, and Musiela in 1997 (Brace et al. (1997) [2]). LIBOR stands for London Inter-Bank Offered Rate and is calculated daily through an average of rates offered by banks in London. Separate rates are quoted for different maturities (e.g., three months and six months) and different currencies. LIBOR rates are based on *simple* interest. If L denotes the rate for an accrual period of length δ (usually expressed in years), then the interest earned on one unit of currency over the accrual period is δL.

A *forward* LIBOR rate works similarly. We fix δ and consider a maturity T. The forward rate $L(0,T)$ is the rate set at time 0 for the interval $[T, T+\delta]$. If we enter into a contract at time 0 to borrow 1 at time T and repay it with interest at time $T+\delta$, the interest due will be $\delta L(0,T)$.

We now consider a class of models in which a finite set of maturities or *tenor dates*

$$0 = T_0 < T_1 < \cdots < T_n < T_{n+1}$$

are fixed in advance. Many derivative securities tied to LIBOR and swap rates are sensitive only to a finite set of maturities and it should not be necessary to introduce a continuum to price and hedge these securities. Let

$$\delta_i = T_{i+1} - T_i, \quad i = 0, \ldots, n,$$

denote the lengths of the intervals between tenor dates. Often, these would all be equal to a nominally fixed interval of a quarter or half year; but even in this case, day-count conventions would produce slightly different values for the fractions δ_i.

For each date T_i, we let $B_{i-1}(t)$ denote the time t price of a zero-coupon bond maturing at T_i, $\quad 0 \leq t \leq T_i$ and $i \in 1, 2, \ldots, n+1$. Similarly, we write $L_i(t)$ for the forward rate as of time t for the accrual period $[T_{i+1},\ T_{i+2}]$;

(Note: we use this notation to fit in with the indexing of the vectors we use in our C++ program, with $L_0(t)$ being the first forward LIBOR at time t, over the accrual period $[T_1,\ T_2]$, which fixes at time T_1).

The relation defining the forward LIBORs is given in terms of the bond prices by

$$L_i(t) = \frac{B_i(t) - B_{i+1}(t)}{\delta_{i+1} B_{i+1}(t)}, \quad 0 \le t \le T_{i+1}, \quad i = 0, 1, \ldots, n-1. \tag{2.1}$$

After T_{i+1}, the forward rate L_i becomes meaningless but it simplifies notation to extend the definition of $L_i(t)$ beyond T_{i+1} by setting $L_i(t) = L_i(T_{i+1})$ for all $t \ge T_{i+1}$.

2.1 Stochastic Differential Equation

The LMM, as described in Glasserman (2004) [11], is a model in which the evolution of the forward LIBOR rates is described by a system of stochastic differential equations of the form (with the usual notations):

$$\frac{dL_i(t)}{L_i(t)} = \mu_i(t, \mathbf{L}(t))dt + \sigma_i(t)dW_i(t), \quad 0 \le t \le T_{i+1}, \quad i = 0, \ldots, n-1. \tag{2.2}$$

with W, Brownian motion verifying $dW_i(t) \cdot dW_j(t) = \rho_{ij}\, dt$ and $\{\rho_{ij}\}_{(i,j)\in\{0,\ldots,n-1\}^2}$, correlation matrix of the LIBORs.
Moreover, $\Sigma \equiv (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$ defines the vector of volatilities of the forward LIBORs.

The drift term $\mu_i(t, \mathbf{L}(t))$ is a function of the LIBORs' vector $\mathbf{L}(t)$ and we state its formula in the following section.

2.2 Drift in the forward measure

We consider the forward measure $\mathbb{P}_{n+1}$ for maturity T_{n+1} and take the bond B_n as numeraire asset. We define the deflated bond prices to be ratios $B_i(t)/B_{n+1}(t)$, which simplifies to

$$\prod_{j=i+1}^{n-1} (1 + \delta_j L_j(t)). \tag{2.3}$$

We can derive the expression for the drift term starting from the requirement that the deflated ratios be martingales and proceed by induction (backwards from $i = n-1$) to derive restrictions on the evolution of the n LIBORs.

The drift term is then given by (Glasserman (2004) [11]):

$$\mu_i(t, \mathbf{L}(t)) = -\sum_{j=i+1}^{n-1} \frac{\delta_j L_j(t) \sigma_i(t) \sigma_j(t) \rho_{ij}}{1 + \delta_j L_j(t)}, \quad 0 \le t \le T_{i+1},\ i = 0, \ldots, n-1. \tag{2.4}$$

We finally find that the arbitrage-free dynamics of the n LIBORs $L_i,\ i = 0, \ldots, n-1$, under the forward measure $\mathbb{P}_{n+1}$ are given by

$$\frac{dL_i(t)}{L_i(t)} = -\sum_{j=i+1}^{n-1} \frac{\delta_j L_j(t) \sigma_n(t) \sigma_j(t) \rho_{ij}}{1 + \delta_j L_j(t)} dt + \sigma_i(t) dW_{n+1}(t), \tag{2.5}$$

$$\text{for } 0 \le t \le T_{i+1}, \text{ and } i = 0, \ldots, n-1.$$

2.3 Log-Coordinates

The log-coordinates of the vector $\mathbf{L}$ are given by the vector $\mathbf{K} = \log(\mathbf{L})$. We rewrite equation (2.2) with vector notations to obtain

$$d\mathbf{K}(t) = \mu^{\mathbf{K}} dt + \Sigma \cdot dW, \tag{2.6}$$

where $\Sigma \equiv (\sigma_0, \sigma_1, \ldots, \sigma_{n-1})$, $\mu^{\mathbf{K}} \equiv (\mu_0^{\mathbf{K}}, \ldots, \mu_{n-1}^{\mathbf{K}})$ and $\mu_i^{\mathbf{K}} \equiv \mu_i^{\mathbf{L}} - \frac{1}{2}\sigma_i^2$ by Itô's Lemma.

Log-coordinates will be used for our implementation of the LIBOR market model.

3 Partial Proxy Scheme

In this chapter, we present the methodology of the partial proxy scheme, introduced by Fries and Kampen (2006) [8].

The idea is to improve the likelihood ratio method in Monte Carlo simulations by generating paths from a scheme, referred to as the proxy scheme, which is simpler than the considered target (or original) scheme, but not too far from it, and by adjusting the proxy measure obtained, that means to introduce weights (via the likelihood ratio method) in the Monte Carlo sum which approximates the expectation of the option price.

To explain in details what is behind this idea, we start by redefining a Monte Carlo simulation and the computation of sensitivities, afterwards we will recall the notion of scheme and finally, we will develop the methodology of the proxy scheme of Fries and Kampen [8], Fries and Joshi [7] and Fries [6].

3.1 Monte Carlo simulations

Let Σ be a volatility matrix and Γ be the Cholesky decomposition of the correlation matrix $\{\rho_{ij}\}_{(i,j)\in\{0,\dots,n-1\}^2}$ (with n being the dimension of the SDE), i.e. Γ satisfies the equation $(\Gamma\cdot\Gamma^\top)_{ij}=\rho_{ij}$ for all $(i,j)\in\{0,\dots,n-1\}^2$. Define a filtered probability space $(\Omega,\mathbb{Q},\mathcal{F},\{\mathcal{F}_t\})$, fulfilling the usual conditions, and let U be an n-dimensional $\mathbb{Q}$-Brownian motion with mutually uncorrelated components.

A standard Monte Carlo simulation of a stochastic differential equation, e.g. an Îto process satisfying the stochastic differential equation

$$dK=\mu^K dt+\Sigma\cdot\Gamma\cdot dU,\quad K(0)=K_0, \tag{3.1}$$

defined over $(\Omega,\mathbb{Q},\mathcal{F},\{\mathcal{F}_t\})$, is given by generating sample paths $\omega_1,\dots,\omega_n$ of time-discrete realizations of the equation:

$$K(t+\Delta t)=K(t)+\int_t^{t+\Delta t}\mu^K(\tau)d\tau+\int_t^{t+\Delta t}\Sigma(\tau)\cdot\Gamma(\tau)\cdot dU(\tau). \tag{3.2}$$

Since the integrals in (3.2) are usually not available in closed form, the time-discrete process is approximated, for example by an Euler scheme:

$$K^*(t+\Delta t)=K^*(t)+\mu^{K^*}\Delta t+\Sigma\cdot\Gamma\cdot\Delta U\,. \tag{3.3}$$

The Monte Carlo approximation of the expectation $\mathbb{E}[f(K(T))|\mathcal{F}_0]$ of a function f of a realization $K(T)$ is given by

$$\begin{aligned}\mathbb{E}[f(K(T))|\mathcal{F}_0]&=\int f(\kappa)\phi^K(\kappa-K_0)d\kappa\\&\approx\int f(\kappa)\phi^{K^*}(\kappa-K_0)d\kappa\\&\approx\frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}}f(K^*(T,\omega_i))\end{aligned} \tag{3.4}$$

where ϕ^K and ϕ^{K^*} denote the probability density functions of $K(T)$ and $K^*(T)$ respectively and n_{MC} denotes the number of simulations. To shorten notation we will drop the conditioning on $\mathcal{F}_0$ in the expectation and the K_0, implicitly viewing the probabilities as transition probabilities depending on K_0 as a parameter. We assume that the time discretization error is small, i.e. that the densities ϕ^K and ϕ^{K^*} are close.
The whole procedure involves two approximation errors: the time discretization error and the Monte Carlo error, i.e. the error introduced by the approximation of the last integral in equation (3.4) through a sum.

3.2 Sensitivities in Monte Carlo simulations

We now present how to compute sensitivities in Monte Carlo simulations, as described in Fries and Kampen (2006) [8].

Let θ denote any model parameter (e.g. K_0, Σ, Γ) and let us assume that the densities ϕ^K and ϕ^{K^*} depend smoothly on θ and are $\mathcal{C}^1$ close to each other, which means that

$$\sup_\theta \left|\phi^K(\theta) - \phi^{K^*}(\theta)\right| + \sup_\theta \left|\frac{\partial}{\partial\theta}\phi^K(\theta) - \frac{\partial}{\partial\theta}\phi^{K^*}(\theta)\right| < \epsilon, \text{ with } \epsilon > 0. \tag{3.5}$$

Then one might differentiate the above approximation to get partial derivatives of the expectation $\mathbb{E}[f(K(T))]$ with respect to θ (giving the risk measure):

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}[f(K(T))] &= \int f(\kappa)\frac{\partial\phi^K}{\partial\theta}(\kappa)d\kappa \\ &\approx \int f(\kappa)\frac{\partial\phi^{K^*}}{\partial\theta}(\kappa)d\kappa \\ &\approx^{?} \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f'(K^*(T,\omega_i))\cdot\frac{\partial K^*}{\partial\theta}(T,\omega_i)\end{aligned} \tag{3.6}$$

In applications the partial derivative is numerically replaced by finite differences.
The last step in equation (3.6) holds only in a weak sense and might not even be an "approximation", e.g. if f is not smooth, say even discontinuous, the last term in (3.4) is discontinuous too, thus not differentiable. This is the reason why finite differences applied to Monte Carlo simulation has poor convergence rates for non-smooth functions f.

As Fries and Kampen (2006) [8] explain, this problem can be solved by using the Monte Carlo approximation of the differentiated integral rather than differentiating the Monte Carlo approximation, i.e. we consider

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}[f(K(T))] &= \int f(\kappa)\frac{\partial\phi^K}{\partial\theta}(\kappa)d\kappa \\ &\approx \int f(\kappa)\frac{\partial\phi^{K^*}}{\partial\theta}(\kappa)d\kappa = \int f(\kappa)\frac{\frac{\partial}{\partial\theta}\phi^{K^*}(\kappa)}{\phi^{K^*}(\kappa)}\phi^{K^*}(\kappa)d\kappa \\ &\approx^{!} \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^*(T,\omega_i))\cdot\frac{\frac{\partial}{\partial\theta}\phi^{K^*}(K^*(T,\omega_i))}{\phi^{K^*}(K^*(T,\omega_i))}\end{aligned} \tag{3.7}$$

If we compare the last term in (3.7) with the one in (3.4) we see that the partial derivative is just the expectation of a weighted payoff function $f \cdot w$ where the weight is given by

$$w = \frac{\frac{\partial}{\partial\theta}\phi^{K^*}}{\phi^{K^*}} = \frac{\partial}{\partial\theta}\log\left(\phi^{K^*}\right) \tag{3.8}$$

Thus the sensitivity has a similar approximation error than the price. This is essentially the likelihood ratio approach of Broadie and Glasserman (1996) [3] or the application of a Malliavin weight (see Nualart (1995) [16]).

It should be noted that (3.7) already exhibits a slight difference to the way the likelihood ratio or Malliavin weight is usually considered, namely that we consider the weight to be derived from the scheme K^* and not from the original scheme K, in other words: we first apply a time discretization to the scheme and then apply the likelihood ratio. This is the key idea behind the notion of proxy scheme that we will see in more details in the following.

We present in the next section the concept of scheme that we have already used in the introduction of this chapter.

3.3 Simulation scheme

To define a simulation scheme, we need to consider a discretization $0 \equiv t_0 < t_1 < t_2 < \dots$ of simulation time. We want to generate samples $K(t_i, \omega_j)$ of the time t_i-realizations $K(t_i)$ of the stochastic process K.
The simulation scheme is usually a time-discrete process $K^o(t_i),\ i = 0, 1, 2, \dots$, which approximates $K(t_i)$.

There exists a large number of different schemes, like the Predictor-Corrector scheme or the Trapezoidal Average Drift scheme, but we will only consider the well-known Euler (or log-Euler) scheme in this dissertation.

Starting from the equation (2.2) of the LIBOR market model,

$$\begin{aligned} dK(t) &= \mu^K dt + \Sigma \cdot dW \\ &= \mu^K dt + \Sigma \cdot \Gamma \cdot dU, \end{aligned} \tag{3.9}$$

where W is an n-dimensional $\mathbb{Q}$-Brownian motion with correlated components and U is an n-dimensional $\mathbb{Q}$-Brownian motion with mutually uncorrelated components, the *Euler Scheme* is given by

$$K^e(t_{i+1}) = K^e(t_i) + \mu^K(t_i, K^e(t_i)) \cdot (t_{i+1} - t_i) + \Sigma \cdot \Gamma \cdot (U(t_{i+1}) - U(t_i)). \tag{3.10}$$

The Euler scheme realizations $K^e(t_i)$ or the log-Euler scheme realizations $L^e(t_i)$ are basic approximations of the true realizations $K(t_i)$ and $L(t_i)$ respectively.

If we are under a non stochastic volatility model, i.e. if $\Sigma \cdot \Gamma$ is constant over $[t_i, t_{i+1}]$, we can evaluate the discretization error of the scheme which corresponds to the inaccurate integration of the drift term:

$$\int_{t_i}^{t_{i+1}} \mu^K(t, K(t))dt \approx \mu^K(t_i, K^e(t_i)) \cdot (t_{i+1} - t_i). \tag{3.11}$$

We now possess all the necessary tools to fully understand the idea behind the proxy scheme methodology. This is the subject of the next section.

3.4 Proxy Scheme

We take up the result of the section 3.2, in equation (3.6) and modify the approach towards a more generic framework to which we may apply finite differences by shifting input parameters while retaining the smoothness and convergence properties of a likelihood ratio method.

We follow the idea introduced by Fries and Kampen (2006) [8] and consider a second scheme, K^o, referred to as the proxy scheme with probability density function ϕ^o. We recall that we have the stochastic process K of probability density function ϕ already discretized with a target scheme K^* of probability density function ϕ^*.
ϕ^o should be close to ϕ but need not to be a very accurate approximation.

We consider the following equations to illustrate the proxy scheme innovation:

$$\begin{aligned}\mathbb{E}[f(K(T))] &= \int f(\kappa)\phi^K(\kappa)d\kappa \\ &\approx \int f(\kappa)\phi^{K^*}(\kappa)d\kappa = \int f(\kappa)\frac{\phi^{K^*}(\kappa)}{\phi^{K^o}(\kappa)}\phi^{K^o}(\kappa)d\kappa \\ &\approx \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\phi^{K^*}(K^o(T,\omega_i))}{\phi^{K^o}(K^o(T,\omega_i))}\,.\end{aligned} \tag{3.12}$$

Here, we only need to generate paths via the proxy scheme, which is supposed to be simpler than the original one, and correct the approximation error by adding weights in the Monte Carlo sum in order to get the expectation of the (discounted) payoff.

We continue with the analysis of Fries and Kampen (2006) [8]. For the sensitivity with respect to a model parameter θ, we take the proxy scheme K^o and its density ϕ^{K^o} fixed, i.e. that does not depend on θ, and may therefore differentiate this approximation when ϕ^{K^*} is $\mathcal{C}^2$ close to ϕ^K to obtain the likelihood ratio weighted Monte Carlo:

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}[f(K(T))] &= \int f(\kappa)\frac{\partial\phi^K}{\partial\theta}(\kappa)d\kappa \\ &\approx \int f(\kappa)\frac{\partial\phi^{K^*}}{\partial\theta}(\kappa)d\kappa = \int f(\kappa)\frac{\frac{\partial}{\partial\theta}\phi^{K^*}(\kappa)}{\phi^{K^o}(\kappa)}\phi^{K^o}(\kappa)d\kappa \\ &\approx \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\frac{\partial\phi^{K^*}}{\partial\theta}(K^o)}{\phi^{K^o}(K^o)} \\ &= \frac{\partial}{\partial\theta}\left(\frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\phi^{K^*}(K^o)}{\phi^{K^o}(K^o)}\right)\,.\end{aligned} \tag{3.13}$$

Remarks The differential operator only acts on ϕ^{K^*} since ϕ^{K^o} is assumed fixed. For the implementation, the realizations are generated by one scheme which will be used for both the price and the sensitivities. The model parameter θ steps in at only one place, in the

transition probability ϕ^{K^*}, i.e. θ is present in the Monte Carlo weights and nowhere else. Thus, the sensitivities may be calculated generically by applying finite differences to the numerical implementation of the model. This constitutes the main advantage of the proxy scheme methodology.

The second approach that we consider in this dissertation is the Automatic Differentiation methodology. This is the purpose of the next chapter.

4 Automatic Differentiation

In this chapter, we present the notion of Automatic Differentiation and its application to the computation of accurate Greeks in Monte Carlo simulations.

The concept of Automatic Differentiation (referred to as AD) applied to Greeks computation has been recently introduced by Giles and Glasserman (2006) [10] and works are still in progress, especially by Capriotti (2010) [4] and Capriotti and Giles (2010) [5].

In the context of AD, derivatives are computed by using the very well known chain rule for composite functions, in a clever way since the evaluation of a function and its derivatives are calculated simultaneously, using the same code and common temporary values. If the code for the evaluation is optimised, then the computation of the derivatives will automatically be optimised. One very important point concerning this methodology is that it is only suitable for continuous payouts.

4.1 First approach

The analysis that we present in this section is very closely based on Giles (2007) [9]. AD concerns the computation of sensitivity information from an algorithm or computer program.

Consider a computer program which starts with a number of input variables u_i, $i = 1, \ldots, n_u$, represented by the vector $\mathbf{u}^0$. Each step in the execution of the computer program computes a new value as a function of two previous values. Appending this new value to the vector of active variables, the k^{th} execution step can be expressed as

$$\mathbf{u}^k = \mathbf{f}^k\left(\mathbf{u}^{k-1}\right) \equiv \begin{pmatrix} \mathbf{u}^{k-1} \\ f_k\left(\mathbf{u}^{k-1}\right) \end{pmatrix}, \tag{4.1}$$

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

$$\mathbf{u}^N = \mathbf{f}^N \circ \mathbf{f}^{N-1} \circ \ldots \circ \mathbf{f}^2 \circ \mathbf{f}^1\left(\mathbf{u}^0\right). \tag{4.2}$$

Defining $\dot{\mathbf{u}}^k$ to be the derivative of the vector $\mathbf{u}^k$ with respect to one particular element of $\mathbf{u}^0$, differentiating (4.1) gives

$$\dot{\mathbf{u}}^k = D^n \dot{\mathbf{u}}^{k-1}, \text{ where } D^k \equiv \begin{pmatrix} I^{k-1} \\ \partial f_k / \partial \mathbf{u}^{k-1} \end{pmatrix}, \tag{4.3}$$

and with I^{k-1} being the identity matrix with dimension equal to the length of the vector $\mathbf{u}^{k-1}$. The derivative of (4.2) then gives

$$\dot{\mathbf{u}}^N = D^N D^{N-1} \ldots D^2 D^1 \dot{\mathbf{u}}^0, \tag{4.4}$$

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++. 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 now introduce the following notations to explain in details the two ways of using AD, i.e. the "forward" and "adjoint" modes:

Let P denote the price of a derivative, θ denote the vector of the model parameters, of length n_θ, which we wish to differentiate the price P with respect to, and let S be the state vector of length n_S that represents, for example, in the context of the LIBOR market model, the values of (log of) the forward LIBORs at each time step of a Monte Carlo path.

To obtain the value of the price P, we start from the model parameters vector θ in input and generate the state vector S from which we can obtain P via the chain rule:

$$\theta \longrightarrow S \longrightarrow P.$$

We want to compute the derivative of P with respect to each of the elements of θ, holding fixed the randomly generated Brownian path increments for this particular path calculation.

We will consider two different approaches to this problem: the forward Automatic Differentiation and the backward Automatic Differentiation, that we present in the following sections.

4.2 Tangent or Forward mode

Adopting the notation used in the Algorithmic Differentiation research community, let $\dot{\theta}$, $\dot{S}$, $\dot{P}$ denote the derivative with respect to one particular component of θ. Straightforward differentiation gives

$$\dot{S} = \frac{\partial S}{\partial \theta}\,\dot{\theta}, \quad \dot{P} = \frac{\partial P}{\partial S}\,\dot{S}, \tag{4.5}$$

and hence

$$\dot{P} = \frac{\partial P}{\partial S}\,\frac{\partial S}{\partial \theta}\,\dot{\theta}\,. \tag{4.6}$$

The standard pathwise sensitivity analysis proceeds forwards through the process (this is referred to as "forward mode" or "tangent mode" in AD terminology) and can be illustrated by the following figure:

$$\dot{\theta} \longrightarrow \dot{S} \longrightarrow \dot{P}\,.$$

We note that the forward mode operates from input parameters towards the output price.

4.3 Adjoint or Backward mode

Again, following the notation used in the AD community the adjoint quantities $\overline{\theta}, \overline{S}, \overline{P}$ denote the derivatives of P with respect to θ, S, P, respectively, with $\overline{P} = 1$ by definition. Differentiating again, one obtains

$$\overline{\theta} \triangleq \left(\frac{\partial P}{\partial \theta}\right)^T = \left(\frac{\partial P}{\partial S}\,\frac{\partial S}{\partial \theta}\right)^T = \left(\frac{\partial S}{\partial \theta}\right)^T \overline{S}, \tag{4.7}$$

and similarly

$$\overline{S} = \left(\frac{\partial P}{\partial S}\right)^T \overline{P}, \tag{4.8}$$

giving

$$\overline{\theta} = \left(\frac{\partial S}{\partial \theta}\right)^T \left(\frac{\partial P}{\partial S}\right)^T \overline{P}. \tag{4.9}$$

The adjoint analysis proceeds backwards ("adjoint mode" or sometimes called "reverse mode" in AD terminology) and can be illustrated by the following chain rule, with arrows in the opposite sense:

$$\overline{\theta} \longleftarrow \overline{S} \longleftarrow \overline{P}.$$

The sensitivities are computed via the backward mode, from the option price back towards the initial input parameters.

Remarks The forward and reverse (or backward or adjoint) modes compute exactly the same payoff sensitivities since $\dot{P} = \overline{\theta}$. The only difference is in 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.

To give a more concrete example, suppose we have a Monte Carlo with N time steps and m underlying assets (or more generally, state variables which could be (log) stock prices or (log) forward LIBOR rates in a LIBOR market model). The vector of input parameters θ is a vector with m elements. It corresponds to the initial i.e. time zero value of the underliers (e.g. (log) forward LIBOR rates). If we include the initial i.e. time zero value of the underliers, then the state vector S has $n_S \equiv m(N+1)$ elements (1 for the time zero and N for each time-step, for each of m underliers).

Then $\partial S/\partial\theta$ is a matrix which has $m^2(N+1)$ elements. Computing this matrix will typically be time-consuming. Clearly the elements which correspond to time zero will be trivial to compute (they are one or zero) and in a simple Black-Scholes model many other elements will be zero.
However, in a model with state-dependent drifts (e.g. LIBOR market model) or state-dependent volatilities (e.g. a local volatility model), computing $\partial S/\partial\theta$ will involve many non-trivial elements and so will be time-consuming, perhaps $\mathcal{O}(Nm^2)$ in a worst-case scenario.

The vector $\partial P/\partial S$ will be assumed to be a vector with $m(N+1)$ elements, i.e. the payoff can depend on any of the m underliers at each of $N+1$ times (including time zero). Computing equation (4.6) will have overhead roughly

$$\mathcal{O}(m(N+1)) \cdot \mathcal{O}(Nm^2) \sim \mathcal{O}(N(N+1)m^3)$$

and will get one element of the m element vector $\dot{P}$.
Computing equation (4.9) will have overhead roughly

$$\mathcal{O}(m(N+1)) \cdot \mathcal{O}(Nm^2) \sim \mathcal{O}(N(N+1)m^3)$$

i.e. the same as in equation (4.6) but it will get all m elements of the m element vector $\overline{\theta}$.

Hence, independent of N, equation (4.9) is approximately m times more efficient in that it computes all m elements of the vector $\overline{\theta}$ in the same time as equation (4.6) computes one of the m elements of $\dot{P}$.
Note that this conclusion holds whatever the value of N and whatever the actual computational overhead of calculating $\partial S/\partial\theta$.

We stress again that $\dot{P} = \overline{\theta}$. In other words, the benefit of using the reverse (or backward or adjoint) mode is that it is computationally faster -it, in no way, changes the accuracy of the estimates of the partial derivatives $\partial P/\partial\theta$.

Allied with these explanations concerning both the partial proxy scheme and the Automatic Differentiation applied to the pathwise method, we continue this dissertation with the most important parts, i.e. the implementation of these methodologies in a C++ program and their illustration with some numerical examples.

5 Implementation

In this chapter, we discuss the implementation of the different methodologies considered in this project. The central part of the work has been to create classes able to generate one single path according to the LIBOR market model SDE. This approach raises several problems, and the main ones are listed below:

- How to generate independent, normally distributed random numbers.
- How to include the Automatic Differentiation software into C++ files in order to keep the whole code as generic as possible.

The first point concerning the random numbers is very important. Indeed, all our results are based on simulations driven by random numbers. As a consequence, if the numbers are not following the required law, all our results will be irrelevant.

We need to generate independent standard normal random numbers, i.e. numbers following the law $\sim \mathcal{N}(0,1)$. As discussed in Glasserman (2004) [11], the C++ function "$rand()$" is not a good solution, especially when one wants to generate a very large number of random numbers. Other methods such as Linear Congruential Generators are also not ideal because the periodicity of the resulting numbers is typically too small.
We have chosen to use the Mersenne Twister algorithm for the generation of Gaussian random numbers (see Matsumoto and Nishimura (1998) [15]). This algorithm presents a high periodicity and is known to pass tests of "randomness". Files, written in C++, have been provided by John Crosby at UBS which were included into our project. The implementation of the Mersenne Twister algorithm allows us to change the "seed" to generate different streams of random numbers. Uniform random numbers were converted into normal random numbers using the inverse cumulative normal methodology.

The second point concerning the implementation of the Automatic Differentiation methodology is also an important one. Indeed, the beauty of this methodology lies in the fact that it is totally generic and very simple to implement. However, after having included header files coded by the FADBAD team into the project's code, we have been obliged to replace some simple functions by template functions and to modify the way we were using the Monte Carlo simulations.
We will illustrate this point by presenting short extracts of pseudo-code written in C++ style.

We now present the three different methods that have been implemented during this project for pricing options on the LMM:

- "Bump-and-revalue"
- Proxy Scheme
- Automatic Differentiation.

5.1 Bump-and-revalue

The first method, that we will henceforth call "bump-and-revalue", consists of evaluating the price of an option utilising Monte Carlo simulation and a two-sided finite difference for each bumped model parameter.

We consider the input parameter vector $[\theta_1, \ldots, \theta_{n_\theta}]$ which can be any relevant parameters used in the model (for example, the initial values of the LIBORs or their volatilities). We are interested in computing the partial derivative of the price $P(\theta)$ of an option with respect to θ_l, i.e. the quantity $\partial P(\theta)/\partial\theta_l$, for $l = 1, \ldots, n_\theta$.

The "bump-and-revalue" methodology consists in computing $P(\theta_1, \ldots, \theta_l + h_l, \ldots, \theta_{n_\theta})$ and $P(\theta_1, \ldots, \theta_l - h_l, \ldots, \theta_{n_\theta})$ to approximate

$$\frac{\partial P(\theta)}{\partial \theta_l} \text{ by } \quad \frac{P(\theta_1, \ldots, \theta_l + h_l, \ldots, \theta_{n_\theta}) - P(\theta_1, \ldots, \theta_l - h_l, \ldots, \theta_{n_\theta})}{2h_l} \tag{5.1}$$

for each $l \in \{1, \ldots, n_\theta\}$.

This approach is computationally very expensive since it requires $2n_\theta$ Monte Carlo simulations to estimate all the sensitivities.
However, it is completely generic and will provide a good benchmark to evaluate the competitiveness of our two new methods in our future tests.

In our implementation, we always used a proportional bump size of 1% on each model parameter. In other words, the bump size on the l^{th} parameter (denoted by h_l in equation (5.1)) is always equal to $0.01\,\theta_l \equiv h_l$.

The most naive implementation of bump-and-revalue essentially repeats a Monte Carlo simulation with different input parameters.
In other words, in pseudo-code, with central finite-differencing, the program would look like:

```
for(int k=0; k<=(2*n_theta); k++){
    // bump input parameters if k !=0
    for(int i=0; i<nMC; i++){
        // do simulation ...
    }
}
```

In the above, the $k = 0$ loop would compute the (unbumped) price and the remaining loops would recompute the price with bumped inputs. The resulting prices are stored and then one computes the appropriate Greeks via equation (5.1).

This is precisely how we have cast bump-and-revalue thus far. However, in our implementation, we actually did things slightly differently. In pseudo-code, our program looks like:

```
for(int i=0; i<nMC; i++){
```

```
    // do simulation ...
    for(int k=0; k<=(2*n_theta); k++){
        // code which implements that which changes for
        // each required partial derivative
    }
}
```

In other words, we wrote the program so that the bumping occurs inside the Monte Carlo loop i and not outside.
The reason for this is that if, for example, we are computing a delta with respect to the initial value of a forward LIBOR rate, only the initial starting point of the simulation changes. Everything else about the simulation is the same (at least in a simple LIBOR market model).

Clearly, there can be considerable computational savings for delta calculations by writing the code this way. One can do likewise for vega calculations although since changing volatilities will change entire paths the computational savings will probably be quite modest and limited to not having to redraw random numbers, for example.
Neverthess, the result is that our implementation is somewhat optimised and is, therefore, faster than the most naive implementation of bump-and-revalue and the reader should bear this in mind when comparing calculation times.

However, we stress that in real-world applications the scope for this sort of optimisation is very modest. For example, the code will become very cluttered and hard to maintain. More pertinently, if we had instead wished to compute a partial derivative with respect to a shift in a swap rate of a specific tenor or with respect to a shift in the continuously-compounded spot interest-rate to all maturities or with respect to a shift in the Black volatility of caps of all maturities up to 5 years, say, then our supposed code optimisation would quickly be more of a hindrance rather than a help.

5.2 Proxy Scheme

In this section, we explain how we have implemented the proxy scheme methodology.
The price of the option is computed by equation (3.12):

$$\mathbb{E}[f(K(T))] \approx \frac{1}{n_{MC}} \sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i)) \cdot \frac{\phi^{K^*}(K^o(T,\omega_i))}{\phi^{K^o}(K^o(T,\omega_i))} \,,$$

where the weights are expressed in terms of the transition probabilities:

$$\frac{\phi^{K^*}(K^o(T,\omega_i))}{\phi^{K^o}(K^o(T,\omega_i))} = \prod_{j=0}^{n_{Step}-1} \frac{\phi^{K^*}(t_j, K^o(t_j,\omega_i); t_{j+1}, K^o(t_{j+1},\omega_i))}{\phi^{K^o}(t_j, K^o(t_j,\omega_i); t_{j+1}, K^o(t_{j+1},\omega_i))}. \tag{5.2}$$

Firstly, we present the pseudo-code that corresponds to this method:

```
for(int i=0; i<nMC; i++){ //loop on the number of Monte Carlo simulations
    for(int j=0; j<nStep; j++){ //loop on the time step
        generate_random_numbers();
```

```
        generate_one_step_via_proxy_scheme();
        compute_weight_for_one_step();
    }
    //at the end of this loop, we have computed one entire path with
    //the proxy scheme and we get the specific weight to apply in the
    //future Monte Carlo sum

    price=take_weighted_average_over_all_MC_simulations();
}
```

In this pseudo-code, we already know how to generate the random numbers, i.e. via the Mersenne Twister generator, and we also know how to generate one step via an Euler scheme but we have not presented yet the computation of the Monte Carlo weights.

The weights are computed using the transition probability of an Euler scheme (or a log-Euler scheme if we deal with L and not directly with the vector of the logarithm of the LIBORs K).
For a given path ω, we have the equation (using the notation previously introduced):

$$K^e(t_{i+1},\omega) = K^e(t_i,\omega) + \mu^K(t_i)\cdot(t_{i+1}-t_i) + \Sigma\cdot\Gamma\cdot(U(t_{i+1},\omega)-U(t_i,\omega)). \tag{5.3}$$

Recall that U is an n-dimensional $\mathbb{Q}$-Brownian motion with uncorrelated components. Equation (5.3) can be solved for $\Delta U(t_i) = U(t_{i+1}) - U(t_i)$ by:

$$\Delta U(t_i) = \Gamma^{-1}\cdot\Sigma^{-1}(\Delta K^e - \mu^K(t_i)\Delta t_i). \tag{5.4}$$

Using the transition probability of $\Delta U(t_i,\omega)$

$$\phi(t_i,U(t_i,\omega);t_{i+1},U(t_{i+1},\omega)) = \frac{1}{(2\pi\Delta t_i)^{n/2}}\exp\left(-\frac{(\Delta U(t_i))^2}{2\Delta t_i}\right) \tag{5.5}$$

we get:

$$\phi^{K^e}(t_i,K_i;t_{i+1},K_{i+1}) = \frac{1}{(2\pi\Delta t_i)^{n/2}}\exp\left(-\frac{1}{2\Delta t_i}\left(\Gamma^{-1}\cdot\Sigma^{-1}(\Delta K^e-\mu^K(t_i)\Delta t_i)\right)^2\right). \tag{5.6}$$

For the sensitivities, recall the equation (3.13) from the chapter on the partial proxy scheme, with θ being a model parameter:

$$\begin{aligned}\frac{\partial}{\partial\theta}\mathbb{E}[f(K(T))] &\approx \frac{\partial}{\partial\theta}\left(\frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\phi^{K^*}(K^o)}{\phi^{K^o}(K^o)}\right)\\ &= \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\frac{\partial}{\partial\theta}\phi^{K^*}(K^o)}{\phi^{K^o}(K^o)}.\end{aligned} \tag{5.7}$$

We now apply the two-sided finite difference to the last equation to get:

$$\frac{\partial}{\partial\theta}\mathbb{E}[f(K(T))] \approx \frac{1}{n_{MC}}\sum_{i=1}^{n_{MC}} f(K^o(T,\omega_i))\cdot\frac{\phi^{K^*}(K^o,\theta+h)-\phi^{K^*}(K^o,\theta-h)}{2h\,\phi^{K^o}(K^o)}. \tag{5.8}$$

From this approach, we get the following pseudo-code:

```
double h; //we choose the shift in the two-sided finite differences

for(int i=0; i<nMC; i++){ //loop on the number of Monte Carlo simulations
    for(int j=0; j<nStep; j++){ //loop on the time step
        generate_random_numbers();
        generate_one_step_via_proxy_scheme();
        w1=compute_weight_for_one_step(theta+h);
        w2=compute_weight_for_one_step(theta-h);
        sensitivity_weight=(w1-w2)/(2*h);
    }
    //at the end of this loop, we have computed one entire path with the
    //proxy scheme and the specific weight of one sensitivity

    dprice_dtheta=take_weighted_average_over_all_MC_simulations();
}
```

We see that we only need to compute paths once via the proxy scheme since we only use the finite differences on the weights.

Then we can compute the sensitivities with respect to any model parameter in the same time as we are computing the price of one path. The proxy scheme method gives several advantages that we list in the following bullet points.

5.2.1 Advantages

- The method may be used for weak schemes where one does have an analytic formula ϕ^{K^*} but does not have efficient method for drawing realizations of K^*.
- The method may be used to compute sensitivities of discontinuous functions f.
- The method is efficient in terms of memory consumption.

5.2.2 Drawbacks

- The method will fail to correct the transition density ϕ^o if the condition that ϕ^o be close and $\mathcal{C}^2$ close to ϕ is not verified.

5.3 Automatic Differentiation

In this section, we present the implementation of the Automatic Differentiation via the FADBAD software.

FADBAD is a C++ program package (downloadable from the internet) which combines the two basic ways of applying the chain rule, namely forward and backward (or adjoint) Automatic Differentiation. Both the forward and the backward differentiation methods use operator overloading to redefine the arithmetic operations, so that the program is capable of calculating first order derivatives.

The only thing a user has to provide is the C++ program that performs the evaluation of the function. Since the computation of the derivatives is itself a C++ program we can obtain higher order derivatives by building the forward and the backward Automatic Differentiation classes on top of each other.

FADBAD needs to classify variables according to their use, which is done as follows:

- A dependent variable is a variable which has been assigned to another dependent variable or to a non-constant expression.
- An independent variable is a variable which has not been used yet, or which has been assigned to a constant, another independent variable or to an expression where no variables appeared.

As already explained in chapter 4 the forward and backward algorithm are structurally very different.

We present here how the forward and backward algorithms have been implemented in the FADBAD software [18], as described in the software documentation from Stauning and Bendtsen (1996) [1].

In the forward algorithm, the differentiation is carried out alongside of the function evaluation and when differentiating, it is convenient to store the partial derivatives in the classes of the dependent variables.
For the backward algorithm, the operator overloading is used to form a directed acyclic graph during the forward sweep, i.e. alongside of the function evaluation. During the reverse sweep the graph is traversed backwards and the class of each occurring variable stores the partial derivative with respect to itself, of the dependent variables that one wishes to differentiate.

Thus, at the end of the backward sweep the partial derivatives with respect to the independent variables are stored in the classes corresponding to the independent variables.

Another significant difference between the forward and backward method is the use of storage. In the forward algorithm there is no need to store temporary objects when they are no longer used in the function evaluation. This is however not the case for the backward method where the recording requires the storage of all temporaries until the differentiation is taking place and since most programs lead to quite a few temporary objects, the storage cost can be high.

All this implementation of the management of dependent and independent variables to lead to the computation of derivatives is provided in the three C++ header files of the FADBAD software and need not be altered.

This last point is the main advantage of the Automatic Differentiation methodology, i.e. the user does not need to implement code specific to the partial derivatives of a new option payoff, for example, he just needs to call functions already implemented. This offers significant flexibility to the user, especially when one wants to be able to change quickly

and easily the option payoffs and the underlying models. We will see in chapter 6 how Greeks can be computed in a displaced diffusion model with almost no change in our code.

We now present the specific implementations of the different ways to obtain accurate Greeks in a Monte Carlo simulation, including several extracts of pseudo-code.

5.3.1 Forward mode

The Forward Automatic Differentiation (FAD) methodology can be implemented in two different ways, either on the payoff or on the whole simulation.

The first one consists of using the Automatic Differentiation software on each payoff computed by a single Monte Carlo simulation, i.e. we run the Automatic Differentiation software on a single function, which takes as input the state vector at the final time (this is essentially the payout function), and before that, the final state vector has been computed by another function, independently of the Automatic Differentiation software (i.e. this long function does not need to be a template function able to support the FAD template type "F<double>"). This first way is very fast for specific models where we know the derivatives of the final state vector with respect to the input parameters (which are the parameters we want to obtain the derivatives with respect to).

Indeed, as described in Glasserman (2004) [11], the chain rule for differentiation applied on a payoff P, a state vector S, and a vector of input parameters $\theta = S(0)$ gives:

$$\frac{dP}{d\theta} = \frac{dP}{dS(0)} = \frac{dP}{dS(T)} \frac{dS(T)}{dS(0)}, \tag{5.9}$$

with T being the maturity of the option and 0 the initial time.

As a consequence, we can use the FAD methodology to get $dP/dS(T)$ for any arbitrary payoff but use a hand-coded formula to get $dS(T)/dS(0)$ for simple cases where we know analytical formulae for the derivatives of the final state vector with respect to the initial input parameters. We illustrate this idea by the following pseudo-code:

FAD on each payoff

```
for(int i=0; i<n_MC; i++){
    vector<double> ST=function(theta); //operates on doubles
    dST_dtheta=Hand_coded_formula(theta,ST);

    //we create the F<double> objects for the FAD methodology
    vector<F<double> > F_ST(ST);
    dP_dST=FAD_on_each_payoff(F_ST); //operates on F<double>'s

    //we compute the sensitivities by the chain rule
    dP_dtheta=dP_dST * dST_dtheta;
}
//we are out of the Monte Carlo loop:
//we can take the average over all simulations to obtain
//the final price and the final sensitivities.
```

The second way of using the FAD methodology consists of calling the Automatic Differentiation software on one entire simulation of the price. This way is totally generic and need not know anything about the computation of the derivatives of the option payoff or the computation of $dS(T)/dS(0)$.

We illustrate this idea with the following pseudo-code that displays in particular the precise implementation of the Forward Automatic Differentiation (and the management of the new template type "F<double>" of the FADBAD software):

FAD on the whole simulation

```
double price; //price of the option
vector<double> dprice_dtheta(n_theta); //vector of future sensitivities
vector<F<double> > F_theta(n_theta); //vector of parameters

for(int j=0; j<n_alpha; j++){
    F_theta[j].diff(j,n_theta);
    //declare the parameters with respect to which
    //we want to compute the derivative of the price
}

F<double> F_payoff;

for(int i=0; i<n_MC; i++)){ //loop on the Monte Carlo simulations
    F_payoff=long_function(F_theta,other);
    //computes the price associated to one path

    price=F_payoff.x();
    //evaluates the value of the price and converts it into a double

    for(int j=0; j<n_theta; j++){
        dprice_dtheta[j]=F_payoff.d(j);
        //computes the sensitivities associated to one path
    }
}
```

5.3.2 Adjoint mode

In this subsection, we present the implementation of the Adjoint (or Backward) Automatic Differentiation (BAD) methodology.
We essentially consider the two different ways, as described in the case of the FAD methodology, to implement the BAD methodology "on each payoff" and "on each simulation".

Adjoint on each payoff

This method uses the BAD methodology to compute the derivatives of the payoff with respect to the initial input parameters via the chain rule. This implies that we compute the derivatives of the final state vector with respect to the initial input parameters by hand-coded formulae and we call the FADBAD software in an adjoint mode for computing the

derivatives of the payoff with respect to the final state vector.

This idea will be very fast in cases where we know explicit formulae for the derivatives $dS(T)/d\theta$, since we do not need to create too many temporary objects useful for the backward sweep of the adjoint mode. We will create temporary objects only for the last function that computes the derivatives of the payoff with respect to the final state vector $S(T)$.

Adjoint on the whole simulation

By contrast with the previous method, the BAD methodology applied on the whole simulation computes the derivatives of the payoff P with respect to the initial input parameters θ completely by the adjoint mode, i.e. the Automatic Differentiation software is called on the complete function that computes the payoff from the initial input parameters, and not only from the final state vector $S(T)$.

We illustrate this method with the following pseudo-code that displays in particular the precise implementation of the Backward (or adjoint) Automatic Differentiation (and the use of the new template type "B<double>" of the FADBAD software).

```
double price; //price of the option
vector<double> dprice_dtheta(n_theta); //vector of future sensitivities
vector<B<double> > B_theta(n_theta); //vector of parameters

B<double> B_payoff;

for(int i=0; i<n_MC; i++)){ //loop on the Monte Carlo simulations
    B_payoff=long_function(B_theta,other);
    //computes the price associated to one path

    B_payoff.diff(0,1);
    price=B_payoff.x();
    //evaluates the value of the price and converts it into a double

    for(int j=0; j<n_theta; j++){
        dprice_dtheta[j]=B_theta[j].d(0);
        //computes the sensitivities associated to one path
    }
}
//we are out of the Monte Carlo loop:
//we can take the average over all simulations to obtain
//the final price and the final sensitivities.
```

We stress again that all the methods presented in this sub-section work in an automated fashion and do not require the user to write new code if the option payoff changes (nor if the model changes for the "Forward on the whole simulation" and "Adjoint on the whole simulation" methodologies). In the next chapter, we will present different numerical examples to illustrate the two different methodologies introduced in chapters 3 and 4 of this dissertation.

6 Numerical Examples

In this chapter, we illustrate the partial proxy scheme and the Automatic Differentiation methodologies with several numerical examples. In particular, we stress the case of a Vanilla Caplet in the LIBOR market model for which price and sensitivities can be benchmarked by the results computed from the Black formula. We also look at more complicated options such as Binary Cash or Nothing basket options and Forward Starting Digital Caplets. Finally, we test the Automatic Differentiation methodology with a Vanilla Caplet in a displaced diffusion version of the LIBOR market model.

All these examples have a large number of parameters, such as the number of forward LIBORs, the number of Monte Carlo simulations, the initial values of the forward LIBORs and their volatilities, the presence of a discount factor or not and the ramp width (as defined in section 1). We try to explore their respective impacts on our results in order to determine the strengths and weaknesses of the methodologies.

In this context, our reference or base-case for both the accuracy of the results and the computational time will always be the "bump-and-revalue" methodology, as described in section 5.1.

6.1 First tests of the methods

In this section, we test the forward Automatic Differentiation methodology in the case of a Vanilla call option under the Black Scholes model.
We have chosen this basic example to be able to verify the results and to demonstrate the correctness of the algorithms that we have developed since we know analytical formulae for both the price and the sensitivities. Several methods are displayed here:

1. Analytical: Analytical price, analytical Greeks (Black formula).
2. Anal.+FD: Analytical price, Greeks obtained by two-sided finite difference.
3. Simple MC: Simple Monte Carlo simulation (price and Greeks determined by hand-coded formulae in a log-Euler scheme)
4. MC+ramp: Monte Carlo simulation with a ramp applied on the payoff to obtain an estimate of the gamma.
5. FAD Simu: Forward Automatic Differentiation on the whole Monte Carlo simulation.
6. FAD Payoff: Forward Automatic Differentiation on the payoff.

We have used the following parameters:

Risk-free interest rate $r = 0.05$
Dividend yield $q = 0.0$
Volatility $\sigma = 0.25$
Maturity $T = 1$ year
Spot $S = 100$

Strike $K = 95$
Ramp width $h = 0.003$
Proportional bump size in finite differences: $shift = 1\%$
Number of Monte Carlo simulations: $n_{MC} = 10\,000$
Number of time-steps: $n_{Step} = 5$.

The results are displayed in table (1).

Table 1: Results of the test for a Vanilla Call Option

Method	1 Analytical	2 Anal.+FD	3 Simple MC	4 MC+ramp	5 FAD Simu	6 FAD Payoff
Price	15.047058	15.047058	15.046813	15.046813	15.046813	15.046813
Delta	0.702004	0.701935	0.701980	0.701980	0.701980	0.701980
Gamma	0.0138654	0.0138639	-	0.014	-	-
Vega	34.663590	34.663210	34.665089	34.665089	34.665089	34.665089
Rho	55.143784	55.143652	-	-	55.1512	55.1512

Comments on the price Compared to the analytical result, the prices computed by Monte Carlo simulation or Forward Automatic Differentiation have an absolute error of

$$(15.047058 - 15.046813) = 2.45 \cdot 10^{-4}$$

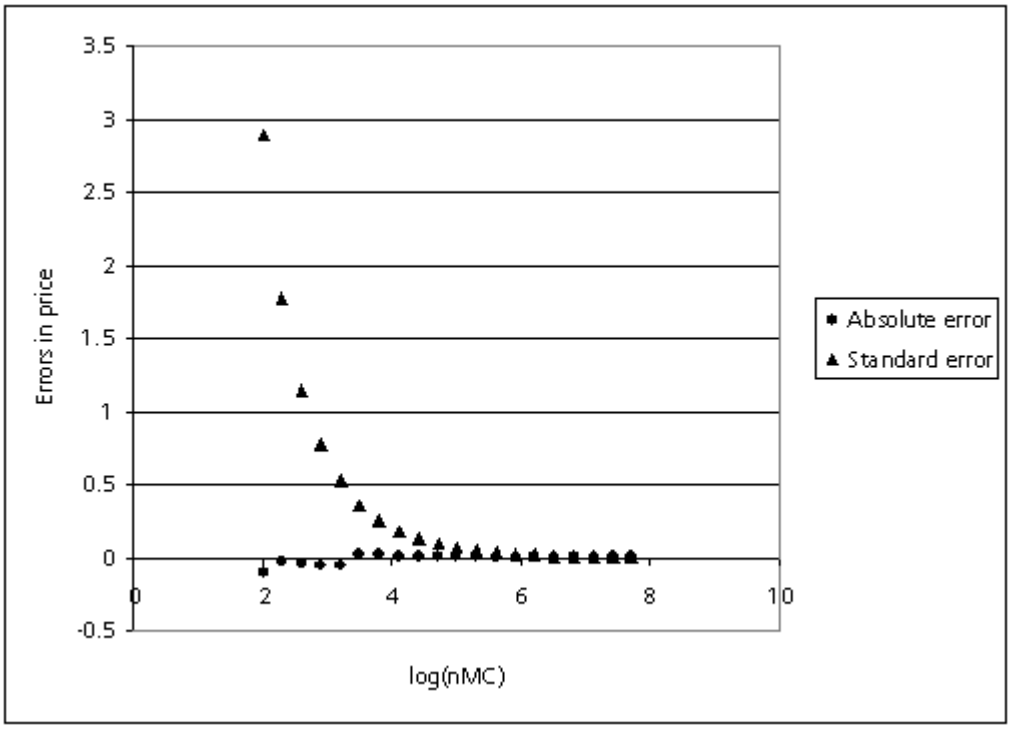
for 10 000 paths compared with the exact price of the analytical method. We are interested to see the impact of the number of simulations on the error in the Monte Carlo simulation estimate of the price of the option.

We plot in figure (1) the absolute error $\epsilon = P_{\text{Anal}} - P_{\text{MC}}$, between the analytical price and the (simple) Monte Carlo simulation estimate of the price, and the standard error of the Monte Carlo estimate of the price, as a function of $\log_{10}(n_{MC})$, all other parameters fixed.

We take values for n_{MC} going from 10^2 to 10^8. The absolute error is represented by dots. We note that the standard error, represented by triangles, decreases with the logarithm of the number of simulations. In fact, we see that the standard errors in figure (1) are consistent with being proportional to $1/\sqrt{n_{MC}}$.

Comments on the sensitivities All the available sensitivities are very accurate. We note in particular that we have obtained Rho (the sensitivity of the option price with respect to changes in the risk-free interest rate) very easily thanks to the AD software. AD software offers the flexibility to allow the user to implement very quickly an extra sensitivity.

Figure 1: Absolute and Standard Errors in Price



6.2 Particular case of a Vanilla Caplet

In this section, we present several tests done on Vanilla caplets using our implementation of the LIBOR market model.

6.2.1 Test on 12 LIBORs

Description of the test

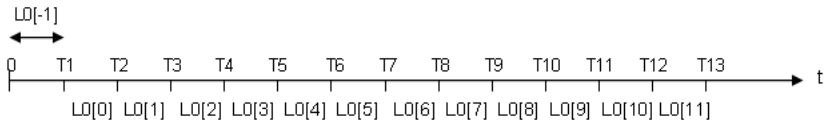
We consider 12 LIBORs spanning a time period from time zero (today) out to 13 years. Each LIBOR is assumed to be one year in length (in practice, typical LIBOR periods are 3 months or six months but our aim, in this dissertation, is not to replicate every market convention but to produce illustrative results for both typical and untypical cases).

The LIBOR which sets (or fixes) at time zero (today) is denoted by $L_0[-1]$ and is assumed to be known. It is the known interest rate applicable to the period from time $T_0 \equiv 0$ until the time one year from now i.e. from time T_0 until T_1.

We let $L_0[i-1]$, for $i \geq 1$, be the forward LIBOR rate, as observed at time zero, for the period $[T_i, T_{i+1}]$. So, for example, $L_0[0]$ is the forward LIBOR rate, as observed at time zero, applicable from one year from now until two years from now i.e. from time T_1 until T_2, $L_0[1]$ is the forward LIBOR rate, as observed at time zero, applicable from two years from now until three years from now i.e. from time T_2 until T_3 and $L_0[11]$ is the forward LIBOR rate, as observed at time zero, applicable from twelve years from now until thirteen years from now i.e. from time T_{12} until T_{13}. In a similar fashion, We let $L_t[i-1]$, for $i \geq 1$, be

the forward LIBOR rate, as observed at time t (with $0 \leq t \leq T_i$), for the period $[T_i, T_{i+1}]$. Clearly, at time T_i, the LIBOR rate for the period $[T_i, T_{i+1}]$ fixes.

Here is a diagram with the time axis (in years) and the time-line of the forward LIBOR rates observed at time 0: We consider a caplet written on the final LIBOR i.e. written



on the LIBOR rate $L_{T_{12}}[11]$ for the period from twelve years from now until thirteen years from now which fixes twelve years from now i.e. the LIBOR applicable rate from time T_{12} until T_{13} which fixes at T_{12}. The payoff of the caplet is $\max(0, L_{T_{12}}[11] - K)$, where K is the strike. In practice, a vanilla caplet on the LIBOR rate $L_{T_{12}}[11]$ would make the payoff of $\max(0, L_{T_{12}}[11]$ at time T_{13}. A Monte Carlo simulation, pricing this caplet would then discount the payoff from time T_{13} back to time $T_0 \equiv 0$. In general terms, the presence (or absence) of the discounting term is not central to our analysis (i.e. to comparing caplet prices and Greeks obtained by different methods). Therefore, in some of our numerical examples, we will omit all discounting. We will do this in this particular example. When we do this, we will indicate this to the reader by referring to a flag "Discounting present" which will be set to "no".

Parameters used for the test

Number of LIBORs $n = 12$
First LIBOR $L_0[-1] = 0.05$
Initial forward LIBORs $L_0 = [0.05, ..., 0.05]$
$vol = [0.2, ..., 0.2]$
Strike $K = 0.05445$
Discounting present: no
Number of Monte Carlo simulations: 5000
Seed of the Mersenne Twister Random Numbers Generator: 6

Results

We have obtained our results displayed in table 2 by two different methods: the Black formula applied to a Vanilla caplet (with no discounting) and a Monte Carlo simulation using the log-Euler approximation of the LMM SDE.

Table 2: Prices of the option

Method	Price	Std error
Exact	0.01373	0
Approx	0.01362	0.00044

Table 3: Computational times

Method	Time (ms)
Exact	16
Approx	7078

Comments We can see that the price obtained from the log-Euler scheme is consistent with the analytical value, taking into account the standard error.
The computational time of the Black formula (referred to as "Exact" in tables 2 and 3) is unbeatable since we have a simple analytical formula whereas the Monte Carlo simulation (referred to as "Approx") used 5 000 paths with many random numbers being generated per path. Of course, Monte Carlo simulation is much more flexible and can be used to price options for stochastic processes and non-standard payoffs for which no analytical formula is available.

6.2.2 Test on 4 LIBORs

In this sub-section, we consider the computation of Greeks -specifically the deltas and the vegas. We stay in the context of Vanilla caplets in order to be able to evaluate the accuracy of our Monte Carlo results compared with analytical formulae.

We test our program in the case of 4 LIBORS spanning time periods of one year each. We consider a caplet written on the final LIBOR i.e. written on the LIBOR rate $L_{T_4}[3]$ for the period from four years from now until five years from now which fixes four years from now. The payoff of the caplet is $\max(0, L_{T_4}[3] - K)$, where K is the strike. As in the previous sub-section, we will omit all discounting i.e. the flag "Discounting present" will be set to "no". We compute the deltas and vegas of the caplet using all the different methods that we consider in this dissertation (AD stands for Automatic Differentiation) and which have been presented in chapter 5:

- *Fwd Payoff*: AD in forward mode applied on each simulated payoff.
- *Fwd simu*: AD in forward mode applied on the whole simulation.
- *Adj Payoff*: AD in adjoint mode on each payoff.
- *Adj simu*: AD in adjoint mode on the whole simulation.
- *Bumping*: A two-sided finite difference with a proportional bump size of 1% as described in section 5.1.

Henceforth, we will denote by "Delta i" the sensitivity of the price with respect to the LIBOR $L_0[i]$.

Parameters used for the test

Number of LIBORs $n = 4$
First LIBOR $L_0[-1] = 0.1$
Initial forward LIBORs $L_0 = [0.08, 0.018, 0.04, 0.055]$

$Vol = [0.25, 0.02, 0.44, 0.2]$
Strike $K = 0.05445$
Discounting present: no
Number of simulations: 100000

We remark that the initial forward LIBORs and their volatilities are not at all typical of those that might be observed in practice. Nonetheless, we should still be able to see close agreement between the prices and Greeks computed by the different possible methods.

Results of the simulation

Table 4: Price of the caplet

Method	Price	Std error
Black	0.008953	0
Proxy	0.008954	5.26E-05
Fwd payoff	0.008954	5.26E-05
Fwd simu	0.008954	5.26E-05
Adj payoff	0.008954	5.26E-05
Adj simu	0.008954	5.26E-05
Bumping	0.008954	5.26E-05

Table 5: Deltas of the caplet

Method	Delta 0	Delta 1	Delta 2	Delta 3
Black	zero	zero	zero	0.589059
Proxy	0.001922	0.003645	-0.00193	0.589183
Fwd payoff	0	0	0	0.588435
Fwd simu	0	0	0	0.588435
Adj payoff	0	0	0	0.589175
Adj simu	0	0	0	0.589175
Bumping	0	0	0	0.588321

In table 5, we present the deltas of the caplet, i.e. the estimates of the partial derivatives of the caplet with respect to $L_0[0], L_0[1]$, $L_0[2]$ and $L_0[3]$.

We remark that it is clear that the price of the caplet has zero sensitivity to $L_0[0], L_0[1]$ and $L_0[2]$ since $L_0[0], L_0[1]$ and $L_0[2]$ play no role in determining its price (given that we are omitting discounting in this example). This is reflected in the results using the Black formula and in all four methods (Fwd payoff, Fwd simu, Adj payoff, Adj simu) which use pathwise differentiation. The Proxy method gives non-zero values for the deltas with respect to $L_0[0], L_0[1]$ and $L_0[2]$. The values are very small but nonetheless non-zero. This is, of course, an unwelcome drawback of the Proxy scheme methodology.

All six methods produce very, very close agreement for the delta (i.e. Delta 3) with respect to $L_0[3]$.
We note that, in fact, the value of Delta 3 obtained by the two adjoint methods should be the same as the value of Delta 3 obtained by the two forward methods (see section 4.3). We see from table 5 that this is not the case. The prices are in agreement and the discrepancies in Delta 3 are small but, nevertheless, the fact is that the values of Delta 3 should also be in perfect agrement. We investigated this and the only explanation we could determine is that there may be a bug in the Automatic Differentiation software. The computational

Table 6: Elapsed times during the simulations

Method	time (ms)
Black	16
Proxy	7468
Fwd payoff	32578
Fwd simu	32484
Adj payoff	59484
Adj simu	60672
Bumping	4812

times displayed in table 6 can be interpreted as if the bump-and-revalue method is the best method but we need to bear in mind that it is an optimised version. Nevertheless, the proxy scheme methodology seems to be very fast whereas the AD software introduces a great slowdown in our program. Moreover, the adjoint mode is slower than the forward mode, certainly due to the storage of the temporary variables needed for the backward sweep. We also note that using AD only on the payoff or on the whole simulation gives almost the same results for either the FAD or the BAD methodologies.

6.3 Binary Cash or Nothing basket option

We now consider an option with the following payoff, defined in a context of n LIBORs:

$$\mathbb{I}\left(\frac{1}{n}\sum_{i=0}^{n-1} L_{T_n}[i] - K > 0\right),$$

where $\mathbb{I}$ is the indicator function, K the strike and $L_{T_n}[i]$ is the i^{th} forward LIBOR that sets at time T_{i+1} but as seen at time T_n. We recall that $L_t[i] = L_{T_{i+1}}[i]$ if $t \geq T_{i+1}$ so that $L_{T_n}[i] = L_{T_{i+1}}[i]$ for each $i \in \{0, \ldots, n-1\}$.
The payoff is paid at time T_n, the time at which the last forward LIBOR sets.
This option could be described as a Binary Cash or Nothing option on an equally weighted basket of n LIBORs.

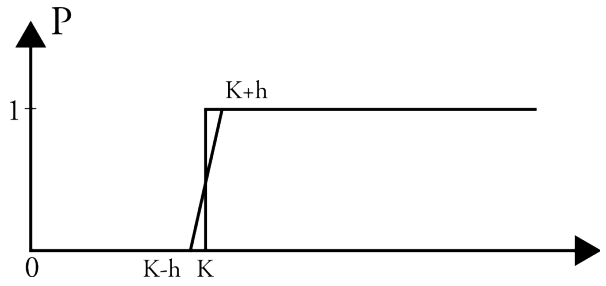
We test our program in the case of 3 LIBORs. We compute the deltas and the vegas via all the possible methods, that we recall here:

- *Fwd Payoff*: AD in forward mode applied on each simulated payoff.

- *Fwd simu*: AD in forward mode applied on the whole simulation.
- *Adj Payoff*: AD in adjoint mode on each payoff.
- *Adj simu*: AD in adjoint mode on the whole simulation.
- *Bumping*: A two-sided finite difference with a proportional bump size of 1% as described in section 5.1.

Parameters used for the Test

Number of LIBORs $n = 3$
First LIBOR $L_0[-1] = 0.1$
Initial LIBORs $L_0 = [0.08, 0.018, 0.04]$
Discounting present: no
Strike $K = 0.05445$
Payoff : $\mathbb{I}_{\left(\sum_{i=1}^{n} \frac{1}{n} L_T[i] - K > 0\right)}$ where we applied a ramp with a ramp width $h = 0.0005$.



$$\text{Volatility matrix} \begin{pmatrix} 0.25 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.44 \end{pmatrix} \text{Correlation matrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Number of Monte Carlo simulations: 5 000 000
Seed of the Mersenne Twister Random Numbers Generator: 47

Table 7: Computation times

Method	Time (ms)
Fwd payoff	940454
Fwd simu	944281
Adj payoff	1446031
Adj simu	1452765
Bumping	429782

Interpretation

The results displayed in tables 7, 8, 9 and 10 are very encouraging. They show that the different methods give some very close results when applied in the same context.

Table 8: Prices

Method	Price	Std error
Fwd payoff	0.423945	0.00022
Fwd simu	0.423945	0.00022
Adj payoff	0.423945	0.00022
Adj simu	0.423945	0.00022
Bumping	0.423945	0.00022

Table 9: Deltas

Method	Delta 0	Std error	Delta 1	Std error	Delta 2	Std error
Fwd payoff	11.97986	0.041306	11.38156	0.038641	9.783272	0.03627
Fwd simu	11.97986	0.041306	11.38156	0.038641	9.783272	0.03627
Adj payoff	12.06618	0.006224	11.45959	0.005911	9.838732	0.005076
Adj simu	12.06619	0.006224	11.45959	0.005911	9.838734	0.005076
Bumping	11.98548	0.032037	11.39005	0.037045	9.792603	0.032277

Table 10: Vegas

Method	Vega 0	Std error	Vega 1	Std error	Vega 2	Std error
Fwd payoff	0.140032	0.002353	0.002001	0.000993	-0.30462	0.001309
Fwd simu	0.140032	0.002353	0.002001	0.000993	-0.30462	0.001309
Adj payoff	0.142472	7.34E-05	0.00283	1.38E-06	-0.30745	0.000158
Adj simu	0.142472	7.34E-05	0.00283	1.38E-06	-0.30745	0.000158
Bumping	0.140658	0.002236	0.002024	0.000991	-0.30518	0.001258

The bump-and-revalue method (referred to as "Bumping" in the tables) allows us to validate the accuracy of the results which use the Automatic Differentiation methodology.
We stress again that the bump-and-revalue method is implemented in an optimised way, which explains why this method seems to be so efficient compared to those that use Automatic Differentiation.

Also, we observe that, although the values of the Greeks obtained by the two adjoint methods should agree perfectly with those obtained by the two forward methods, there are, in fact, discrepancies. We have omitted the results for the case of the Proxy scheme since they were of disappointing accuracy.

6.4 Computational efficiency

The computational efficiency of the Adjoint Automatic Differentiation methodology seems to be disappointing for the moment but we need to consider a case with a larger number of LIBORs and also to consider the possibility of computing all Greeks in the same time. This idea is illustrated in the following test.

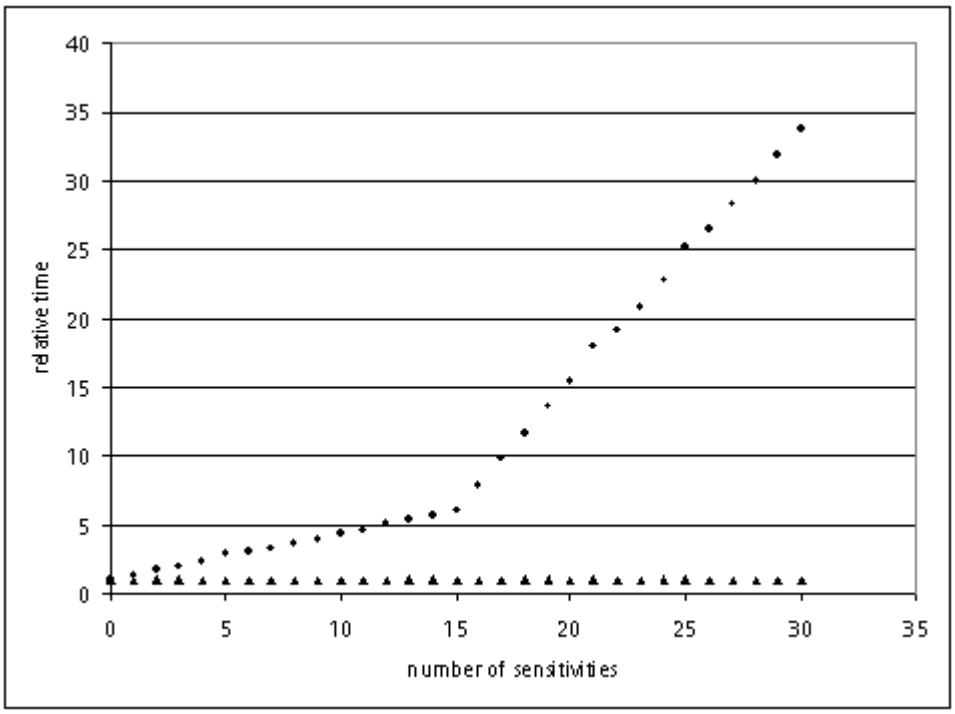
We consider 15 LIBORs (each over one year) and we compute deltas and vegas using two different methods: on one hand, bumping using a two-sided finite difference and re-evaluating and on the other hand, the adjoint mode operated on each payoff.

We plot the graph of the relative computational time in function of the number of sensitivities we want to compute (between 0 and 30, since we have 15 potential deltas and 15 potential vegas). The relative time is the time spent to compute the sensitivities divided by the time spent to compute the price alone.

The option considered is a caplet on the last LIBOR, ie the LIBOR that sets 15 years from now and pays 16 years from now as observed at its fixing time and which is denoted by $L_{T_{15}}[14]$. The payoff of the option is therefore $(L_{T_{15}}[14] - K)^+$, where K is the strike chosen to be $0.99 \cdot L_0[14]$.
Initially, we have set all the parameters at standard values: 10^5 simulations, one timestep per LIBOR interval, initial LIBORs all at 5%, initial volatilities all at 20%, no correlation and no discount factor applied on the payout function.

The results displayed here show the advantages of an adjoint mode when we need to compute a large number of sensitivities.

Figure 2: Computational Efficiency



The triangles represent the adjoint mode whereas the dots are used for the bump-and-revalue.
We notice that the computational time of the adjoint mode does not depend on the number

of sensitivities we want to compute. It is essentially always equal to one. It does not require any additional computational time to obtain 30 sensitivities compared to one sensitivity.

By contrast, bump-and-revalue become very inefficient when one wants to compute a large number of sensitivities. The cost in computational time is proportional to the number of sensitivities.

In this particular example, we have started by computing each delta followed by each vega so the discontinuity that we can notice at the 15^{th} sensitivity corresponds to the transition between a new delta computation and a new vega computation.
This means that the bump-and-revalue methodology, computing one vega is slower than computing one delta but it does not affect our conclusion concerning the efficiency of the adjoint mode.

If we now consider the absolute times of computation needed for these results, we observe that the AD methodology becomes quicker than the bump-and-revalue method as soon as we reach the number of 15 sensitivities. Before this time, the bump-and-revalue method is faster but beyond that time, the AD software is more efficient.

6.5 Forward Starting Digital Caplet

In this section, we consider a different type of payoff: a forward starting digital caplet.

This option is essentially a Binary Cash or Nothing caplet where the underlying is some LIBOR rate observed at its fixing time and whose payoff is made at the end of the accrual period of that given LIBOR, but where the strike of the caplet is the LIBOR rate for an earlier period observed at its fixing time. In other words, the strike of the caplet is not known at time zero.

If we consider n LIBORs and two indices i_{final} and i_{previous} for two different forward LIBORs, a final one (possibly before the n^{th} i.e. the last forward LIBOR) and an earlier one, then the payout of a forward starting digital caplet with multiplicative constant k would be:

$$\mathbb{I}_{\left(L_{T_{i_{\text{final}}}}[i_{\text{final}}-1] \geq k \cdot L_{T_{i_{\text{previous}}}}[i_{\text{previous}}-1]\right)}$$

which is paid at time $T_{i_{\text{final}}+1}$, the time which corresponds to the end of the accrual period for the LIBOR which sets at $T_{i_{\text{final}}}$, and where k is a constant which has the form of a multiplicative strike term.

For our example, we consider an option in which $i_{\text{final}} = n$ and $i_{\text{previous}} = n-1$. The payoff of the option is at time T_{n+1}.
We also chose the strike k to be proportional to the ratio of the two forward LIBORs considered, evaluated at time zero, i.e.

$$k = 1.0005 \, \frac{L_0[n-1]}{L_0[n-2]} \, .$$

Parameters used for the test

Number of Libors $n = 8$
First Libor $L_0[-1] = 0.02$
Initial Libors $L_0 = [0.021, 0.022, 0.023, 0.024, 0.025, 0.026, 0.025, 0.024]$

We note that the initial LIBORs slightly increase and then slightly decrease, which gives the yield curve a hump-shape which is not uncommon, in practice.

Discounting Present: yes
Payoff : $\mathbb{I}_{\left(L_{T_8}[7] \geq \frac{1.0005 L_0[7]}{L_0[6]} \cdot L_{T_7}[6]\right)}$ smoothed by a ramp of width h (h will be a parameter of interest in the following series of tests).

$$\text{Volatility matrix} \begin{pmatrix} 0.25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.28 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.31 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.29 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.28 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.27 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.26 \end{pmatrix}$$

Likewise, the initial volatilities slightly increase and then decrease with increasing LIBOR tenor. In practice, this humped volatility structure is very common. The correlation matrix does not change and remains the identity matrix.

Number of Monte Carlo simulations: 5000
Seed of the Mersenne Twister Random Numbers Generator: 2

Results

As well as computing the price, the deltas and the vegas, we will also consider the impact of using different values of the ramp width h.
The results are displayed in tables 11, 12, 13 and 14 in the following pages. We use ramp widths of $h = 0.00005$ (table 11), $h = 0.0005$ (table 12), $h = 0.005$ (table 13) and $h = 0.05$ (table 14).

We observe that as the ramp width h increases, the price of the forward starting digital caplet increases whereas the standard error of the price decreases. This is highly intuitive - as the ramp width h increases - there will be more paths which fall into the ramp region where the payoff (before discounting) is strictly between zero and one.

Note that the computation times are essentially independent of the ramp width (and therefore we omit them for brevity).

Note that Delta 0 through to Delta 6 correspond to LIBORs which do not appear in the option payoff. They impact the price of the option only through the discounting term. These deltas increase in absolute value as the ramp width increases. This is intuitive since

Table 11: Case $h = 0.00005$

Method	Adjoint	Forward	Bumping
Price	0.396742	0.396742	0.396742
Std error	0.005741	0.005741	0.005741
Time (ms)	79888	37655	37908
Delta 0	-0.389625	-0.388582	-0.388582
Std error	0.006360	0.005623	0.005623
Delta 1	-0.389244	-0.388202	-0.388202
Std error	0.006354	0.005618	0.005618
Delta 2	-0.388864	-0.387822	-0.387822
Std error	0.006347	0.005612	0.005612
Delta 3	-0.388484	-0.387443	-0.387443
Std error	0.006341	0.005607	0.005607
Delta 4	-0.388105	-0.387065	-0.387065
Std error	0.006335	0.005601	0.005601
Delta 5	-0.387727	-0.386688	-0.386688
Std error	0.006329	0.005596	0.005596
Delta 6	-20.63796	-17.81208	-14.10596
Std error	0.334569	7.671166	2.047353
Delta 7	20.70210	17.76243	13.86244
Std error	0.335789	7.989556	2.132683
Vega 0	0	0	0
Std error	0	0	0
Vega 1	0	0	0
Std error	0	0	0
Vega 2	0	0	0
Std error	0	0	0
Vega 3	0	0	0
Std error	0	0	0
Vega 4	0	0	0
Std error	0	0	0
Vega 5	0	0	0
Std error	0	0	0
Vega 6	0.736709	0.674112	0.601339
Std error	0.009948	0.278434	0.153592
Vega 7	-0.690191	-0.517528	-0.646032
Std error	0.010793	0.291770	0.159834

Table 12: Case $h = 0.0005$

Method	Adjoint	Forward	Bumping
Price	0.397159	0.397159	0.397159
Std error	0.005719	0.005719	0.005719
Time (ms)	79469	36750	37765
Delta 0	-0.38986	-0.38899	-0.38899
Std error	0.006364	0.005602	0.005602
Delta 1	-0.38948	-0.38861	-0.38861
Std error	0.006358	0.005596	0.005596
Delta 2	-0.3891	-0.38823	-0.38823
Std error	0.006352	0.005591	0.005591
Delta 3	-0.38872	-0.38785	-0.387851
Std error	0.006346	0.005585	0.005585
Delta 4	-0.38834	-0.38747	-0.387472
Std error	0.006339	0.00558	0.00558
Delta 5	-0.38796	-0.38709	-0.387095
Std error	0.006333	0.005574	0.005574
Delta 6	-13.6589	-14.1077	-13.8602
Std error	0.228331	1.956327	1.63694
Delta 7	13.43312	13.89319	15.24871
Std error	0.224873	2.036517	1.706187
Vega 0	0	0	0
Std error	0	0	0
Vega 1	0	0	0
Std error	0	0	0
Vega 2	0	0	0
Std error	0	0	0
Vega 3	0	0	0
Std error	0	0	0
Vega 4	0	0	0
Std error	0	0	0
Vega 5	0	0	0
Std error	0	0	0
Vega 6	0.647035	0.609805	0.555158
Std error	0.010639	0.0939	0.093906
Vega 7	-0.69084	-0.65278	-0.59629
Std error	0.011361	0.098822	0.098082

Table 13: Case $h = 0.005$

Method	Adjoint	Forward	Bumping
Price	0.39828	0.39828	0.39828
Std error	0.00551	0.00551	0.00551
Time (ms)	79469	36563	38531
Delta 0	-0.39081	-0.39009	-0.390088
Std error	0.006379	0.005396	0.005396
Delta 1	-0.39043	-0.38971	-0.389706
Std error	0.006373	0.005391	0.005391
Delta 2	-0.39005	-0.38933	-0.389325
Std error	0.006367	0.005386	0.005386
Delta 3	-0.38967	-0.38895	-0.388945
Std error	0.00636	0.005381	0.005381
Delta 4	-0.38929	-0.38857	-0.388566
Std error	0.006354	0.005375	0.005375
Delta 5	-0.38891	-0.38819	-0.388187
Std error	0.006348	0.00537	0.00537
Delta 6	-13.2552	-12.6874	-11.9083
Std error	0.214462	0.566646	0.551984
Delta 7	12.90796	12.36678	13.15594
Std error	0.209098	0.588098	0.573804
Vega 0	0	0	0
Std error	0	0	0
Vega 1	0	0	0
Std error	0	0	0
Vega 2	0	0	0
Std error	0	0	0
Vega 3	0	0	0
Std error	0	0	0
Vega 4	0	0	0
Std error	0	0	0
Vega 5	0	0	0
Std error	0	0	0
Vega 6	0.581029	0.569121	0.576536
Std error	0.009497	0.032016	0.030857
Vega 7	-0.62399	-0.60897	-0.61573
Std error	0.010184	0.033156	0.032196

Table 14: Case $h = 0.05$

Method	Adjoint	Forward	Bumping
Price	0.402491	0.402491	0.402491
Std error	0.003779	0.003779	0.003779
Time (ms)	81750	36766	37562
Delta 0	-0.39419	-0.39421	-0.394212
Std error	0.006434	0.003702	0.003702
Delta 1	-0.39381	-0.39383	-0.393826
Std error	0.006428	0.003698	0.003698
Delta 2	-0.39342	-0.39344	-0.393441
Std error	0.006421	0.003694	0.003694
Delta 3	-0.39304	-0.39306	-0.393057
Std error	0.006415	0.003691	0.003691
Delta 4	-0.39265	-0.39267	-0.392674
Std error	0.006409	0.003687	0.003687
Delta 5	-0.39227	-0.39229	-0.392291
Std error	0.006403	0.003684	0.003684
Delta 6	-9.8786	-9.84238	-9.05675
Std error	0.161635	0.118115	0.119338
Delta 7	9.398603	9.326458	10.10686
Std error	0.153365	0.122336	0.123562
Vega 0	0	0	0
Std error	0	0	0
Vega 1	0	0	0
Std error	0	0	0
Vega 2	0	0	0
Std error	0	0	0
Vega 3	0	0	0
Std error	0	0	0
Vega 4	0	0	0
Std error	0	0	0
Vega 5	0	0	0
Std error	0	0	0
Vega 6	0.2662	0.26663	0.266373
Std error	0.004347	0.006929	0.006909
Vega 7	-0.29576	-0.29498	-0.29454
Std error	0.004846	0.007064	0.007076

it is clear that they should be proportional to the price (since they impact only through the discounting term).

Clearly, Vega 0 through to Vega 5 should be (and, observing the tables, are) zero.

We can see that the choice of the ramp width clearly has an impact on the price, the deltas and the vegas.

This leaves open the issue of how to optimally choose the ramp width: Choosing it too large produces too much bias in the price (and hence also the deltas and vegas); choosing it too small produces higher standard errors. In addition, any optimal choice should additionally depend upon the number of Monte Carlo simulations.
Increasing the number of Monte Carlo simulations will, all things being equal, tend to increase the number of paths which fall into the ramp region where the payoff (before discounting) is strictly between zero and one. It seems intuitive, therefore, than an optimal choice of the ramp width would decrease with an increasing number of Monte Carlo simulations.

6.6 Caplet in a displaced diffusion model

In this section, we try to give a more realistic example, using the Automatic Differentiation methodologies for pricing and computing the Greeks of a Vanilla Caplet in a displaced diffusion model with discounting present.

The displaced diffusion model essentially presents the same stochastic differential equation satisfied by the forward LIBORs as before, with the exception that a coefficient, say α_i, called the displaced diffusion coefficient, is added to the dynamics of each forward LIBOR in the following fashion:

$$\frac{d(L_i(t)+\alpha_i)}{L_i(t)+\alpha_i} = \mu_i(t,\mathbf{L}(t))dt + \sigma_i(t)dW_i(t), \quad i = 0,\ldots,n-1. \tag{6.1}$$

This essentially adds a local volatility function to the dynamics of forward LIBORs. This is an extra complication that needs to be dealt with by the Automatic Differentiation (FADBAD) software. It also adds realism to the LIBOR market model in that it allows the model to fit a skew in implied caplet volatilities.

We consider 15 LIBORs and more realistic values for both the initial forward LIBORs and their volatilities, choosing hump-shapes, as displayed in figures (3) and (4).
We choose the vector of displaced diffusion coefficients to have the same value of 4% for all LIBORs, i.e. $\alpha = [0.04,\ldots,0.04]$. The other parameters used for the test are displayed below:

Parameters used for the test

Number of Libors $n = 15$
First Libor $L_0[-1] = 0.06$
$L_0 = [0.061, 0.062, 0.063, 0.064, 0.065, 0.066, 0.067, 0.068, 0.069, \ldots$

Figure 3: Initial forward LIBORs

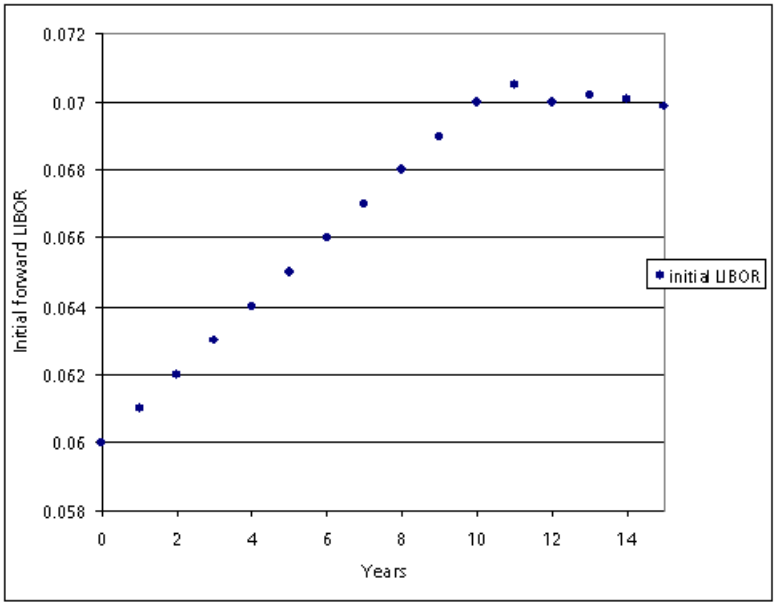
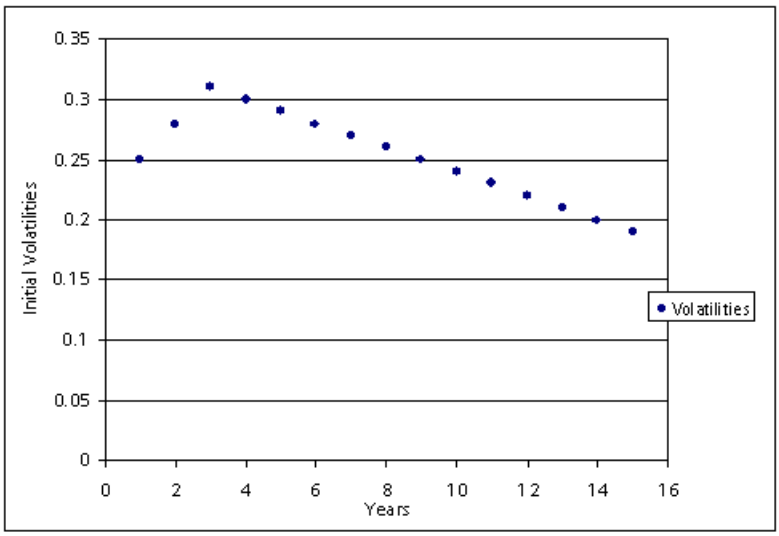


Figure 4: Volatilities



$\ldots 0.07, 0.0705, 0.07, 0.0702, 0.0701, 0.0699]$
Volatilities: $[0.25, 0.28, 0.31, 0.30, 0.29, 0.28, 0.27, 0.26, 0.25, 0.24, 0.23, 0.22, 0.21, 0.20, 0.19]$
Correlation matrix: Identity
Discounting Present: Yes
Strike of the caplet: $K = 0.99\, L_0[n-1] = 0.0702495$
Number of Monte Carlo simulations: 100 000
Seed of the Mersenne Twister Random Numbers Generator: 21

Results

The results of this Monte Carlo simulation are displayed in tables 15, 16, 17 and 18.

Table 15: Price of the caplet

Method	Black	Adj simu	Adj payoff	Fwd simu	Fwd payoff	Bumping
Price	0.011201	0.011268	0.011268	0.011268	0.011268	0.011268
Std error	0	8.59E-05	8.59E-05	8.59E-05	8.59E-05	8.59E-05

Table 16: Computational times

Method	Black	Adj simu	Adj payoff	Fwd simu	Fwd payoff	Bumping
Time (ms)	32	3250594	3347031	1571093	1549297	3263703

Comments

The prices agree with the analytical formula (i.e the Black (1976) formula adjusted for the displaced diffusion case) with a precision of 4 decimals and are consistent with the standard errors reported.

Concerning the computational times, we still see that the adjoint mode is slower that the forward mode since the number of LIBORs that we are considering is not large enough to make the adjoint mode more efficient (as previously said, there is a fixed cost to run the adjoint mode, lying in the storage of values during the forward sweep in order to build the directed acylic graph needed for the backward sweep). The "bump-and-revalue" method seems to stay competitive. However, we stress again that it is an optimised version which would be difficult to generalize. For this case, where we have 15 LIBORs, the results for the forward mode are the best in calculation time.

Table 17 has very satisfying results. Indeed all the results displayed for the deltas are consistent with their respective standard errors. Here again, the values obtained with the forward mode seem better than the one obtained with the adjoint mode. This is difficult to interpret (since the values obtained by the forward mode and the adjoint mode should be the same) but we suspect is due to a bug in the Automatic Differentiation software.

The vegas displayed in table 18 are accurate. The zero values for Vega 0 to Vega 13 show that there is no spurious noise disrupting the accuracy. Vega 14 (which is the partial

Table 17: Deltas and standard errors

Method	Black	Adj simu	Adj payoff	Fwd simu	Fwd payoff	Bumping
Delta 0	-0.01056	-0.01066	-0.01066	-0.01062	-0.01062	-0.01062
Std error	0	3.89E-05	3.89E-05	8.10E-05	8.10E-05	8.10E-05
Delta 1	-0.01055	-0.01065	-0.01065	-0.01061	-0.01061	-0.01061
Std error	0	3.89E-05	3.89E-05	8.09E-05	8.09E-05	8.09E-05
Delta 2	-0.01054	-0.01064	-0.01064	-0.0106	-0.0106	-0.0106
Std error	0	3.89E-05	3.89E-05	8.08E-05	8.08E-05	8.08E-05
Delta 3	-0.01053	-0.01063	-0.01063	-0.01059	-0.01059	-0.01059
Std error	0	3.88E-05	3.88E-05	8.07E-05	8.07E-05	8.07E-05
Delta 4	-0.01052	-0.01062	-0.01062	-0.01058	-0.01058	-0.01058
Std error	0	3.88E-05	3.88E-05	8.07E-05	8.07E-05	8.07E-05
Delta 5	-0.01051	-0.01061	-0.01061	-0.01057	-0.01057	-0.01057
Std error	0	3.88E-05	3.88E-05	8.06E-05	8.06E-05	8.06E-05
Delta 6	-0.0105	-0.0106	-0.0106	-0.01056	-0.01056	-0.01056
Std error	0	3.87E-05	3.87E-05	8.05E-05	8.05E-05	8.05E-05
Delta 7	-0.01049	-0.01059	-0.01059	-0.01055	-0.01055	-0.01055
Std error	0	3.87E-05	3.87E-05	8.04E-05	8.04E-05	8.04E-05
Delta 8	-0.01048	-0.01059	-0.01059	-0.01054	-0.01054	-0.01054
Std error	0	3.86E-05	3.86E-05	8.04E-05	8.04E-05	8.04E-05
Delta 9	-0.01047	-0.01058	-0.01058	-0.01053	-0.01053	-0.01053
Std error	0	3.86E-05	3.86E-05	8.03E-05	8.03E-05	8.03E-05
Delta 10	-0.01046	-0.01057	-0.01057	-0.01053	-0.01053	-0.01053
Std error	0	3.86E-05	3.86E-05	8.03E-05	8.03E-05	8.03E-05
Delta 11	-0.01047	-0.01058	-0.01058	-0.01053	-0.01053	-0.01053
Std error	0	3.86E-05	3.86E-05	8.03E-05	8.03E-05	8.03E-05
Delta 12	-0.01047	-0.01057	-0.01057	-0.01053	-0.01053	-0.01053
Std error	0	3.86E-05	3.86E-05	8.03E-05	8.03E-05	8.03E-05
Delta 13	-0.01047	-0.01057	-0.01057	-0.01053	-0.01053	-0.01053
Std error	0	3.86E-05	3.86E-05	8.03E-05	8.03E-05	8.03E-05
Delta 14	0.218332	0.219032	0.219032	0.218867	0.218867	0.218912
Std error	0	0.0008	0.0008	0.0011	0.0011	0.001099

Table 18: Vegas and standard errors

Method	Black	Adj simu	Adj payoff	Fwd simu	Fwd payoff	Bumping
Vega 0	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 1	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 2	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 3	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 4	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 5	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 6	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 7	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 8	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 9	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 10	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 11	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 12	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 13	0	0	0	0	0	0
Std error	0	0	0	0	0	0
Vega 14	0.056652	0.057836	0.057836	0.057349	0.057349	0.057345
Std error	0	0.000211	0.000211	0.000736	0.000736	0.000736

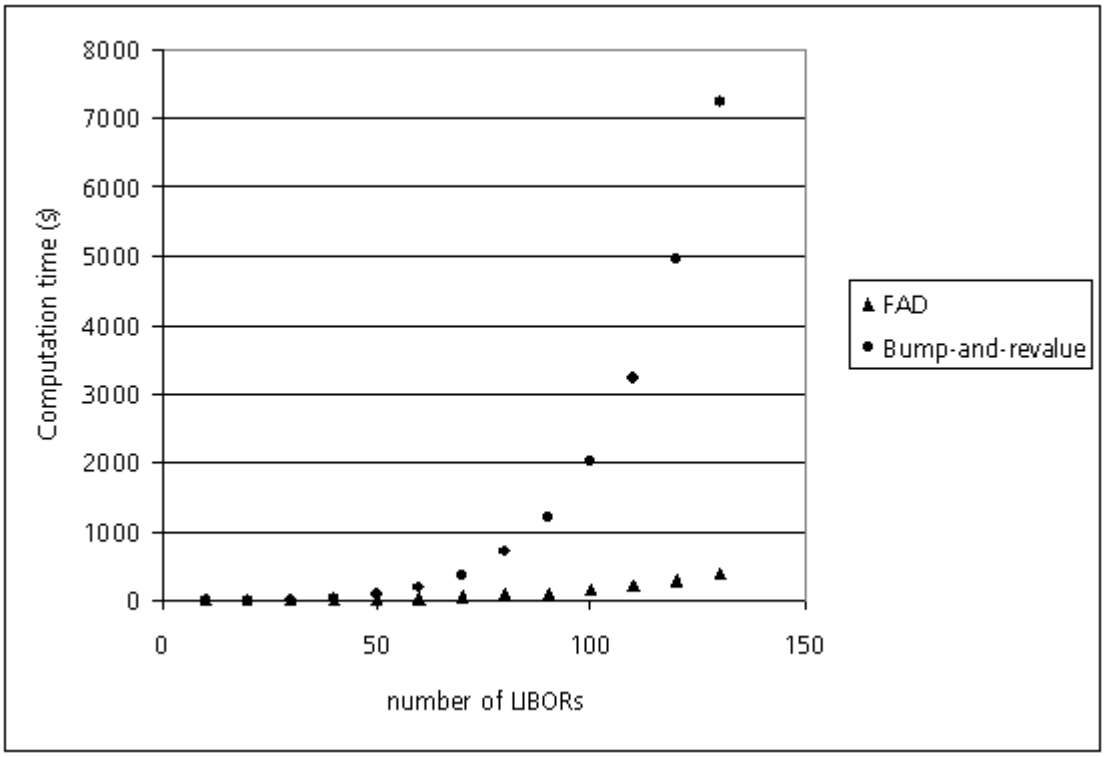
derivative of the price with respect to the last LIBOR), shows only a thin consistency with the standard errors reported for the case of the adjoint modes. Moreover, both the forward modes and the "bump-and-revalue" methods only possess two decimals of precision, compared to the analytical value of the Black formula.

6.7 Final comments on the AD software efficiency

We conclude this chapter of numerical examples with a test on the efficiency of the AD software.

We consider a LIBOR market model with a variable number of LIBORs (each over one year) but with a constant number of Monte Carlo simulations of $n_{MC} = 10$. We allow the number n of LIBORs to vary from 10 to 130. The option is a Vanilla caplet on the last LIBOR with initial forward rates at 5%, initial volatilities at 25% and for each n, we compute all the deltas and vegas available via both the bump-and-revalue and the FAD methodologies.

Figure 5: Computational times in function of the number of LIBORs



The results are displayed graphically in figure (5). We can see that the efficiency of the FAD is real. The FAD is represented by triangles and is far quicker than the bump-and-revalue method represented by dots as the number of LIBORs n becomes large. Since we consider n LIBORs, we compute $2n$ sensitivities for each different n and, as soon as we need more than 100 Greeks (i.e. $n = 50$), the FAD methodology is more efficient than the

bump-and-revalue method (which, to repeat, is written in an optimised way that would be difficult to implement in a real-world application).

7 Conclusion

We have compared three different methodologies, bump-and-revalue, proxy scheme and pathwise differentiation (in forward and in adjoint modes) for obtaining Greeks (partial derivatives of option prices with respect to input parameters) via Monte Carlo simulation. It is clear, conceptually, that using pathwise differentiation in adjoint mode is the best methodology. It is potentially much more efficient than pathwise differentiation in forward mode. In turn, pathwise differentiation in forward mode is, conceptually, much more efficient than bump-and-revalue. Our conclusions are in accordance with the conclusions of Glasserman (2004) [11] and Giles and Glasserman (2006) [10]. We have used Automatic Differentiation software to implement the forward and adjoint modes. This is an extremely attractive method of implementation since it completely automates the task of computing the necessary derivatives and gives the user very significant flexibility when using a large number of different models and payoffs. On the other hand, it is clear that there is a considerable computational overhead from using the FADBAD Automatic Differentiation code. It runs something of the order of 20 times slower than code hand-written in C++. This is a major disadvantage. However, it does not take away the fact that Automatic Differentiation is a very elegant solution because it is so generic. For future research, we suggest that the use of alternative Automatic Differentiation software should be investigated. There are many alternatives available that can be downloaded from the internet, such as ADOL-C, Sacado or ADMB which are all implemented in C++.
These should be investigated. Doubtless each different type of Automatic Differentiation software will have its advantages and disadvantages. But, given the computational overhead of FADBAD, finding the most appropriate Automatic Differentiation software for financial applications is crucial.

References

[1] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for Automatic Differentiation. *Department of Mathematical Modelling, Technical University of Denmark*, 1996.

[2] A. Brace, D. Gatarek, and M. Musiela. The Market Model of Interest Rate Dynamics. *Mathematical Finance*, 7(2):127–155, 1997.

[3] M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Management Science*, 42(2):269–285, 1996.

[4] L. Capriotti. Fast Greeks by Algorithmic Differentiation. 2010.

[5] L. Capriotti and M. Giles. Fast Correlation Greeks by Adjoint Algorithmic Differentiation. 2010.

[6] C.P. Fries. Localized Proxy Simulation Schemes for generic and robust Monte Carlo Greeks. 2007.

[7] C.P. Fries and M.S. Joshi. Partial Proxy Simulation Schemes for generic and robust Monte Carlo Greeks. 2006.

[8] C.P. Fries and J. Kampen. Proxy Simulation Schemes for generic robust Monte Carlo sensitivities, process oriented importance sampling and high accuracy drift approximation (with applications to the LIBOR market model). 2006.

[9] M. Giles. Monte Carlo evaluation of sensitivities in computational finance. In *HERCMA 2007 Conference Proceedings*, 2007.

[10] M. Giles and P. Glasserman. Smoking Adjoints: fast evaluation of Greeks in Monte Carlo calculations. *Risk*, pages 88–92, January 2006.

[11] P. Glasserman. *Monte Carlo methods in financial engineering*. Springer, 2004.

[12] P. Glasserman and X. Zhao. Fast Greeks by simulation in forward LIBOR models. *Journal of Computational Finance*, 3(1):5–39, 1999.

[13] C.J. Hunter, P. Jackel, and M.S. Joshi. Drift approximations in a forward-rate-based LIBOR market model. *Getting the Drift*, pages 81–84, 2001.

[14] M. Leclerc, Q. Liang, and I. Schneider. Fast Monte Carlo Bermudan Greeks. *Risk*, pages 84–88, July 2009.

[15] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. 1998.

[16] D. Nualart. *The Malliavin calculus and related topics*. Springer, 1995.

[17] R. Rebonato. *Modern pricing of interest-rate derivatives: the LIBOR market model and Beyond*. Princeton Univ Pr, 2002.

[18] O. Stauning. Flexible Automatic differentiation using templates and operator overloading in C++. `http://www.fadbad.com`.