Application of Adjoint Differentiation (AD) for Calculating Libor Market Model Sensitivities

Niall Morley

A dissertation submitted to the Faculty of Commerce, University of Cape Town, in partial fulfilment of the requirements for the degree of Master of Philosophy.

September 28, 2018

MPhil in Mathematical Finance,
University of Cape Town.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Philosophy to the University of Cape Town. It has not been submitted before for any degree or examination to any other university.

Niall Morley

September 28, 2018

Abstract

This dissertation explores a key challenge of the financial industry — the efficient computation of sensitivities of financial instruments. The adjoint approach to solving affine recursion problems (ARPs) is presented as a solution to this challenge. A Monte Carlo setting is adopted and it is illustrated how computational efficiency in sensitivity calculation may be significantly improved via the pathwise derivatives method through adapting an adjoint approach. This is achieved through the reversal of the order of differentiation in the pathwise derivatives algorithm in comparison to the standard, intuitive 'forward' approach. The Libor market model (LMM) framework is selected for examples to demonstrate these computational savings, with varying degrees of complexity of the LMM explored, from a one-factor model with constant volatility to a full factor model with time homogeneous volatilities.

Acknowledgements

Firstly, I would like to sincerely thank my internal supervisor, Professor Tom McWalter, for his guidance and support throughout the composition of this dissertation. Further, my gratitude to Dr Jörg Kienitz for providing the topic of this body of work and his advice through the process.

To my parents, John and Paula Morley, I thank you for providing me with every opportunity and always being there to encourage me. To my siblings, Regan and Ciara Morley, thank you for your motivation and always challenging me to strive for better.

To the academic and administrative staff of AIFMRM, my sincere gratitude for opening my mind to a whole new worldview as well as creating a friendly and professional environment that I have been exposed to over the year.

Finally, to the Wardens, Professor Kelly Chibale and Dr Chao Mulenga, and my friends of both Smuts and Fuller Hall, my profound appreciation for all the motivation and support, which has constantly pushed me throughout this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Amongst the greatest practical challenges experienced in the derivatives industry by users of Monte Carlo methods is the accurate and efficient estimation of the Greeks — the sensitivities of a contingent claim to underlying model parameters. The sensitivities form a crucial part of the hedging and risk management operations of market participants. However, their calculations in comparison to derivative pricing often require substantially more computational time. Practitioners employ a large variety of pricing models, however techniques such as deterministic analytical or numerical methods generally can't be implemented for evaluation purposes as the models are too complex. This has resulted in the popularity of Monte Carlo simulation methods for pricing and hedging of complex derivative options, as they have proven the most computationally feasible pricing technique (Capriotti and Giles, 2012).

A finite difference approximation is considered the simplest method for estimating sensitivities. The method involves re-running a Monte Carlo pricing routine multiple times with a perturbation in input parameters to estimate the Greeks. The method is popular due to the simplicity of implementation as well as the relative ease of understanding the method affords. However, disadvantages of the method include an increasing linear computational cost as the number of input parameters increase, leading to severe computational expense. Furthermore, bias and variance properties can be relatively poor for sensitivity estimates calculated via finite difference methods (Glasserman, 2013).

There are more accurate techniques available to practitioners that use information about the underlying model dynamics in a Monte Carlo simulation to derive better estimates of price sensitivities compared to finite difference methods. Two alternative techniques proposed are the pathwise derivatives method (Broadie and Glasserman, 1996) and the likelihood ratio method (Boyle *et al.*, 1997). These techniques, using a single set of simulated paths, are capable of producing unbiased estimates of the Greeks. The pathwise method achieves this through computing

the derivatives of the evolution of the state variables or underlying assets along each path. Comparatively, the likelihood ratio method computes the derivatives of the transition density of the state variables or underlying assets. The disadvantages of applying such techniques are the additional model analysis and implementation required compared to finite difference techniques (Glasserman, 2013).

Giles and Glasserman (2006) presented a further alternative technique to accelerate the computation of sensitivities via Monte Carlo simulation. Their method, known as the adjoint method, applies concepts used in computational fluid dynamics, increasing computational efficiency to calculate pathwise estimates of the derivative sensitivities in the context of the Libor market model (LMM). The advantage of their proposed method is not statistical as it produces identical results to the standard pathwise method. However, the benefit is experienced through the potential computational efficiency of the method.

Kienitz and Nowaczyk (2011) develop the above concepts by demonstrating that calculating sensitivities can be viewed in a broader setting as an affine recursion problem (ARP). Their paper proposes two solutions to the ARPs, namely the forward method and the adjoint method. The forward method applies a forward recursion sequence of calculations to derive the required results. However, the forward method results in a substantial computational cost due to the repeated matrix multiplications that are undertaken. Whereas, the adjoint method achieves tremendous computational savings through the transformation of the matrix recursion problem in the forward method to a vector recursion problem, significantly reducing the required computational complexity. Kienitz and Nowaczyk (2011), through the use of Euler schemes, demonstrate how sensitivities can be calculated, with the approaches being analogous to those presented by Giles and Glasserman (2006). This general approach allows the method to be applied to different models, however for illustrative purposes the LMM is also investigated.

The adjoint method provides users with great versatility. However, as much as this is a strength of the method it is also considered a weakness. The versatility results in users having to derive the precise setup for each Greek calculation separately. The method requires the differentiation of both the evolution equation — the equation which evaluates, at each time step, the approximated diffusion processes, as well as the payoff function of the contingent claim. The evaluation of these derivatives in analytical form may prove difficult, or the ability to evaluate numerically may be computationally inefficient. Further, smoothed approximations may be required as financial instruments often have payoff functions that fail to be twice differentiable. Therefore, an automation of this process would prove beneficial. Thus, the principles of automatic differentiation could be considered.

A set of programming techniques known as algorithmic or automatic differentiation (AD) (Griewank and Walther, 2008) can be applied to computer programs which seek to accurately and efficiently calculate the derivatives of functions. The main concept which underpins these techniques is that any function can be broken down into a composition of elementary arithmetic and intrinsic operations that are easily differentiable, irrespective of the complexity of the composite function. Then, through the appropriate application of the chain rule, users are able to calculate the complex derivatives required to estimate sensitivities. This computational efficiency makes AD desirable compared to standard methods of calculating Greeks. The dependencies between its various paths and the information of the structure of the computer function are exploited to optimise Greek calculation (Capriotti and Giles, 2010). Application of the pathwise derivative method utilising the adjoint mode of AD — adjoint algorithmic differentiation (AAD) — is capable of improving, by several orders of magnitude, the computational efficiency of derivative calculations in comparison to other methods discussed. Capriotti and Giles (2010) demonstrate this point when considering the situation of aggregate risk contained in a portfolio, or when the number of options simultaneously evaluated in a payoff function is less than the number of underlying assets observed.

Therefore, it can be seen that there are two similar approaches available in calculating the sensitivities of financial derivatives — namely the adjoint approach to solving ARPs and AAD. Distinguishing between these approaches can prove challenging due to the overlap and similarities observed. The adjoint approach to solving ARPs seeks to optimise efficiency in the calculation of financial derivative sensitivities through the transformation of matrix-matrix products to matrix-vector products. Whereas, efficiently calculating derivatives by AAD exploits computing code structures — it is dependent on embedded computer functionality and specific software. Therefore, a mathematical manipulation best describes the adjoint approach to evaluating ARPs and falls within the mathematical finance discipline when applied to financial derivatives. In comparison, AAD belongs to the sphere of computer science and is viewed as a 'technological' manipulation. The focus of this dissertation will fall upon the ARP method of calculating sensitivities.

This dissertation continues by providing a discussion on the pertinent literature. The theoretical framework upon which the applications are based will then be laid out. The diffusion processes, approximations of these processes by Euler schemes, the Greeks and the pathwise derivative method are all presented. The ARP in a general context as well as its two solution forms, the forward and adjoint approaches, are then introduced and derived. Subsequently, the LMM is presented with illustration as to how the forward and adjoint methods can be applied to cal-

culate sensitivities. Finally, the relative computational efficiencies of the proposed methods are empirically tested through two example financial options — a caplet and a European swaption.

Chapter 2

Literature Review

The principles of algorithmic differentiation (AD) have in various fields, been applied for decades, including areas such as atmospheric sciences, computational fluid dynamics, engineering design optimization and meteorology. The paper of Homescu (2011) supplies an extensive list of references demonstrating these earlier applications in a wide sphere of areas. The paper also explores the development of AD within the computational finance literature.

The seminal work in AD for computational finance is the paper of Giles and Glasserman (2006), which first introduced and explored adjoint and AD in such a context. A principle reference, outside the field of computational finance, for the application and theory of AD is the work of Griewank and Walther (2008).

Giles and Glasserman (2006) outline various methods of calculating sensitivities before demonstrating how implementing the pathwise derivative method allows these calculations using the adjoint method. The interest rate derivatives considered, in a Libor market model (LMM) context for illustration purposes, are caplets and a portfolio of swaptions. The central finite difference approach is used as a comparative technique to demonstrate the significant computational savings of the adjoint approach in sensitivity calculations. The paper shows the forward method of the pathwise derivatives method to be $10 - 20$ times more efficient than the central differences approach. Furthermore, the adjoint method was shown to be significantly more efficient than the forward method when calculating Deltas and Vegas, with a reduction in several orders of magnitude. The forward method experiences a linear increasing trend in computational cost compared to an approximately constant trend for the adjoint method as maturity values increase.

Whilst, Giles and Glasserman (2006) focus on European derivatives the principle has been extended to simple Bermudan-style financial options by Leclerc *et al.* (2009). The computational efficiency resulting from applying the adjoint approaches is further illustrated in this paper.

In a paper considering correlation Greeks, Capriotti and Giles (2010) further

explore adjoint methodology and apply the method of AD. A Gaussian copula-based Monte Carlo framework is utilised to numerically test the methods on basket default options. To ensure efficiently sampled joint normal random variables a Cholesky factorisation of the correlation matrix is implemented. The adjoint of the Cholesky factorization is used to determine the correlation risk for the pathwise difference approach to be implemented. The paper further demonstrates substantial computational savings in comparison to the finite difference approach.

A companion paper by Capriotti (2011) further demonstrates how speed-ups of several orders of magnitude can be achieved through the application of AD and the pathwise derivative method over standard finite difference methods. Two further types of options are explored to show these savings — the options being a 'best-of' Asian option and a basket option.

The optimisation of calculating counterparty credit risk is the focus of Capriotti *et al.* (2011) through the application of AAD. Credit value adjustment (CVA) calculations form the core of the paper, with results demonstrating real time risk management in a Monte Carlo setting being plausible. Substantial improvements in calculation costs over finite difference approaches are again clearly demonstrated.

As previously mentioned Homescu (2011) explores a detailed history of adjoint methods applied in various computational finance areas to reduce computational costs of sensitivity calculations. The paper expands on all the examples listed below. Joshi and Yang (2011b) explored the co-terminal swap-rate market model, a paper by Joshi and Pitt (2010) studied the displaced-diffusion Libor market model (LMM). Whilst, Beveridge *et al.* (2010) considered the cross-currency displaced diffusion LMM; Adjoint approaches to higher orders were studied by Joshi and Yang (2011a); further Denson and Joshi (2010) detailed the use of adjoint partial differential equation (PDE) methods in the context of Markov-functional models; the Heston model is discussed in Chan *et al.* (2015).

In a further paper considering risk management in real time, Capriotti *et al.* (2015) consider price dynamics given by PDEs and demonstrate, through using AAD, how price sensitivity calculations are orders of magnitude faster than finite difference approaches. Capriotti *et al.* (2017) are able to demonstrate how similar results are achieved through AAD when considering Bermudan-style options for calculating XVA Greeks. Again, significant improvement over finite difference methods are illustrated.

Xu *et al.* (2016) demonstrate how complex functions in computational finance exhibit a natural substitution structure that can be exploited to significantly improve computational time. Through calculations of Greeks in a Monte Carlo setting the paper demonstrates this computational saving in comparison to the standard

reverse-mode AD.

The literature around adjoint methods and AD is a diverse and intriguing space for researchers particularly in the field of computational finance. The value added by implementation of these procedures in driving significant computational savings in a variety of areas has resulted in an ever-expanding body of work. Traditional methods are consequently being replaced by these more efficient procedures as greater understanding and knowledge is gained on the potential of these approaches to solve complex problems.

Chapter 3

Theoretical Framework

This chapter seeks to lay the theoretical foundations upon which the methods and techniques of sensitivity calculation are constructed. The notational framework implemented throughout this paper will further be developed.

3.1 SDE of Diffusion Processes

The following section makes use of notation and terminology that is consistent with the work of Kienitz and Nowaczyk (2011). A probability space $(\Omega, \mathcal{F}, \mathbb{P})$ is considered and let

$$\tilde{X} : \Omega \times \mathbb{R}_{\geq 0} \to \mathbb{R}^m$$

be a stochastic process that is adapted to a filtration $\mathcal{F}_t$. Consider that $\tilde{X}$ is a diffusion, it therefore, satisfies the stochastic differential equation (SDE)

$$d\tilde{X}_t = a(\tilde{X}_t, t)dt + b(\tilde{X}_t, t)dW(t). \tag{3.1}$$

The function $a \in \mathcal{C}^2(\mathbb{R}^m \times \mathbb{R}_{\geq 0}, \mathbb{R}^m)$ is the drift vector, $b \in \mathcal{C}^2(\mathbb{R}^m \times \mathbb{R}_{\geq 0}, \mathbb{R}^{m\times d})$ is the diffusion matrix and W is a d-dimensional Brownian motion. The functions $a = a(y, \sigma)$ and $b = b(y, \sigma)$ may also depend on a set of parameters $\sigma \in \mathbb{R}^q$. It is further assumed that $T > 0$ and that there is a function $g \in \mathcal{C}^2(\mathbb{R}^m, \mathbb{R})$ such that

$$\begin{aligned} g : \mathbb{R}^m &\to \mathbb{R} \\ y &\mapsto g(y) \end{aligned}$$

3.2 Approximation of Diffusions via Euler Schemes

Select any discrete time grid

$$0 = T_1 < ... < T_N = T$$

and define $\delta_n := T_{n+1} - T_n$, $n = 1, ..., N-1$. Select any sequence

$$X(n+1) := F_n(X(n), \sigma), \quad X(1) := \tilde{X}_{T_1} = \tilde{X}_0 \quad \forall 1 \leq n \leq N-1, \tag{3.2}$$

where

$$\begin{aligned} F_n : \mathbb{R}^m \times \mathbb{R}^q &\to \mathbb{R}^m \\ (y, \sigma) &\mapsto F_n(y, \sigma) \end{aligned}$$

is a $\mathcal{C}^2$-map. Equation (3.2) is referred to as the evolution equation. F_n is selected such that $X(n) \approx \tilde{X}(T_N)$. An Euler scheme can be implemented to achieve this:

$$F_n(y, \sigma) := y + a(y, \sigma)\delta_n + b(y, \sigma)Z(n+1)\sqrt{\delta_n}. \tag{3.3}$$

Where $Z(n+1) \in \mathbb{R}^d$ is a sequence of random vectors drawn from the standard normal distribution.

3.3 The Greeks

The above function g is considered and its derivatives are abbreviated as follows:

$$\forall 1 \leq j \leq m : \partial_j(g) := \frac{\partial g}{\partial y_j}.$$

Consider the quantity

$$\tilde{p} := E[g(\tilde{X}_T)],$$

where the function g could be the discounted payoff of a contingent claim at time T, whose underlying is $\tilde{X}$. The quantity $\tilde{p}$ can then be regarded as the fair price of the claim.

The quantity $\tilde{p}$ can be regarded as a function of many variables, namely $\tilde{X}_i(1)$, $i = 1, ..., m$ and σ. The sensitivities below are considered

$$\begin{aligned} \tilde{\Delta} &:= \nabla_{\tilde{X}(1)}(\tilde{p}) \in \mathbb{R}^m, \\ \tilde{\Gamma} &:= \text{Hess}_{\tilde{X}(1)}(\tilde{p}) \in \mathbb{R}^{m \times m}, \\ \tilde{\mathcal{V}} &:= \nabla_{\sigma}(\tilde{p}) \in \mathbb{R}^q. \end{aligned}$$

Respectively they are known as Delta, Gamma and Vega, whilst as a collective they form a subset of sensitivities referred to as the Greeks. Consideration is only given to the above mentioned sensitivities, however higher order or Greeks with respect to other variables can also be selected.

3.4 The Pathwise Derivatives Method

The sensitivities of derivatives can be evaluated through the pathwise derivatives method in a Monte Carlo simulation setting. Drawing on the papers by Giles and Glasserman (2006) and Capriotti (2011) a review of the method is presented.

A multidimensional diffusion process, satisfying a stochastic differential equation, with risk neutral dynamics given by (3.1) is considered. A derivative option with a maturity time T_N and a discounted payoff $g(\bar{X}(T_N))$ has a price $V = \mathbb{E}_{\mathbb{Q}}[g(\bar{X}(T_N))]$ — the expected value, under the risk neutral measure $\mathbb{Q}$, of the discounted payoff. It is also possible that earlier values of the evolution of $\bar{X}$ impact the discounted payoff g. This implies, with $T_1, ..., T_N$ being the appropriate reference dates, that the price of the option will be given by $V = \mathbb{E}_{\mathbb{Q}}[g(\bar{X}(T_1), ..., \bar{X}(T_N))]$.

Monte Carlo methods can be implemented to estimate these expectation values by simulating a number N_{MC} of random samples of the underlying state vector $X = (X(T_1), ..., X(T_M))^t$, resulting in $X[1], ..., X[N_{\mathrm{MC}}]$ — $X[i]$ is the i^{th} sample. Then the associated payoff function $g(X)$ for each simulation is evaluated. To demonstrate approximations of $\bar{X}$ are given by X, a notational change from $\bar{X}$ to X is undertaken. Through the application of the central limit theorem (Kallenberg, 2006) the value of the derivative option can be estimated as:

$$V \approx \frac{1}{N_{\mathrm{MC}}} \sum_{i_{\mathrm{MC}}=1}^{N_{\mathrm{MC}}} g(X[i_{\mathrm{MC}}]). \tag{3.4}$$

Through the use of a single simulation, the pathwise method allows the estimation of the Greeks of an option price V with respect to a set of N_θ parameters $\theta = (\theta_1, ..., \theta_{N_\theta})$. The sensitivity with respect to the kth parameter θ_k is estimated using $\frac{\partial}{\partial \theta_k} g(X[i_{\mathrm{MC}}])$, which is the sensitivity along the ith path of the Monte Carlo simulation of the discounted payoff.

This method produces an unbiased estimate if

$$\frac{\partial}{\partial \theta_k} \mathbb{E}_{\mathbb{Q}}[g(X)] = \mathbb{E}_{\mathbb{Q}}\left[\frac{\partial}{\partial \theta_k} g(X)\right],$$

implying the derivative and the expectation can be interchanged. Glasserman (2013) provides a discussion under which conditions this interchange is permitted. The condition that the discounted payoff g must be Lipschitz continuous is crucial. To demonstrate dependency of g on the set of parameters it is ascribed the subscript θ. Therefore the pathwise derivatives estimate, after ensuring the relevant conditions are satisfied, is given by

$$\bar{\theta}_k = \frac{\partial g_\theta(X)}{\partial \theta_k} = \sum_{j=1}^{m} \frac{\partial g_\theta(X)}{\partial X_j} \frac{\partial X_j}{\partial \theta_k} + \frac{\partial g_\theta(X)}{\partial \theta_k}. \tag{3.5}$$

The sensitivity of the contingent claim prices to θ_k is estimated by calculating and averaging the value (3.5) over each Monte Carlo path.

It is further noted in Capriotti (2011) that g_θ may not only implicitly depend on θ through the state vector $X(\theta)$, but also explicitly. Therefore, when implementing the pathwise derivatives method the second term in (3.5) is valuable and needs to be considered. This term is generally overlooked in the academic literature.

When an m-dimensional diffusion process is considered, where the state vector $X = (X(T_1), ...X(T_N))$ is a respective path, the pathwise derivatives estimate (3.5) may instead be given by

$$\bar{\theta}_k = \sum_{l=1}^{N} \sum_{j=1}^{m} \frac{\partial g_\theta(X(T_1), ...X(T_N))}{\partial X_j(T_l)} \frac{\partial X_j(T_l)}{\partial \theta_k} + \frac{\partial g_\theta(X)}{\partial \theta_k}. \tag{3.6}$$

3.5 The Affine Recursion Problem

The calculation of derivative option sensitivities can be viewed as an affine recursion problem (ARP) according to Kienitz and Nowaczyk (2011). They provide two approaches, known as the forward method and the adjoint method. The merits of the methods are that they are able to provide a general framework which can be applied to estimate sensitivities for a wide range of financial instruments. The forward method relies on repeated matrix multiplication in the recursion scheme therefore a substantial computational cost is observed. However, the adjoint method counteracts this, as the method relies on vector multiplication throughout its recursion scheme. This significantly reduces the computation time experienced. An outline of the two methods presented by Kienitz and Nowaczyk (2011) in a general context is described in the following section.

3.5.1 The Forward Method

It is assumed the following data is provided with $N, m, q \in \mathbb{N}$ and for any $n = 1, ..., N-1$

$$A_1 \in \mathbb{R}^{m \times q}, \quad D(n) \in \mathbb{R}^{m \times m}, \quad C(n) \in \mathbb{R}^{m \times q}, \quad v \in \mathbb{R}^{1 \times m}. \tag{3.7}$$

The forward recursion is then assumed to be satisfied by a sequence of matrices $A(n) \in \mathbb{R}^{m \times q}$ where $\forall 1 \leq n \leq N-1$

$$A(n+1) = D(n)A(n) + C(n), \quad A(1) = A_1. \tag{3.8}$$

With the ARP given by

$$w := vA(N) \in \mathbb{R}^{1 \times q}. \tag{3.9}$$

The matrix A is the recursing matrix, A_1 is the initial matrix, D are the factor matrices, C are the translation matrices, v is the start vector and w is the result vector of the ARP.

Kienitz and Nowaczyk (2011) demonstrate that any ARP which is defined as above in (3.8) is always uniquely solvable. It can be shown

$$\forall 1 \leq n \leq N : A(n) = \left(\sum_{j=1}^{n-1} \left(\prod_{k=1}^{j-1} D(n-k) \right) C(n-j) \right) + \left(\prod_{j=1}^{n-1} D(n-j) \right) A_1. \quad (3.10)$$

Consequently,

$$w = vA(N) = \left(\sum_{j=1}^{N-1} v \left(\prod_{k=1}^{j-1} D(N-k) \right) C(N-j) \right) + v \left(\prod_{j=1}^{N-1} D(N-j) \right) A_1. \quad (3.11)$$

To derive the proof of this result induction is undertaken over n. The initial case considered is $n = 1$:

$$A(1) = \left(\sum_{j=1}^{0} v \left(\prod_{k=1}^{j-1} D(1-k) \right) C(1-j) \right) + \left(\prod_{j=1}^{0} D(1-j) \right) A_1 = A_1 \quad (3.12)$$

Since the initial term sums to 0 and the product term is equal to 1. To demonstrate for general n the standard method of mathematical induction is followed. It is assumed that (3.10) holds true for n, and it is shown below that it holds true for $(n+1)$:

$$\begin{aligned}
&A(n+1) = C(n) + D(n)A(n) \\
=&C(n) + D(n) \left(\left(\sum_{j=1}^{n-1} \left(\prod_{k=1}^{j-1} D(n-k) \right) C(n-j) \right) + \left(\prod_{j=1}^{n-1} D(n-j) \right) A_1 \right) \\
=&C(n) + \left(D(n) \sum_{j=1}^{n-1} \left(\prod_{k=1}^{j-1} D(n-k) \right) C(n-j) \right) + D(n) \left(\prod_{j=1}^{n-1} D(n-j) \right) A_1 \\
=&C(n) + \left(\sum_{j=1}^{n-1} \left(D(n) \prod_{k=2}^{j} D(n-(k-1)) \right) C(n-j) \right) \\
&+ D(n) \left(\prod_{j=2}^{n} D(n-(j-1)) \right) A_1 \\
=&C(n) + \left(\sum_{j=2}^{n} \left(\prod_{k=1}^{j-1} D(n-(k-1)) \right) C(n-(j-1)) \right) + \left(\prod_{j=1}^{n} D(n-(j-1)) \right) A_1 \\
=&\left(\sum_{j=1}^{(n+1)-1} \left(\prod_{k=1}^{j-1} D((n+1)-k) \right) C((n+1)-j) \right) + \left(\prod_{j=1}^{(n+1)-1} D((n+1)-j) \right) A_1.
\end{aligned}$$

The above solution results from the use of algebraic manipulation. The solution utilises the fact that a summation that runs from $j = 1$ to n over j is equivalent to a summation that runs from $j = 2$ to n over $(j - 1)$. Therefore, through the principle of mathematical induction, the above derivation proves (3.10) holds true $\forall 1 \leq n \leq N$.

Through the implementation of the forward recursion (3.8), the result vector w can be calculated using a straightforward algorithm. The computational expense to compute $A(n + 1)$ from $A(n)$ consists of two components, the matrix multiplication of $D(n)A(n)$ and the matrix addition $D(n)A(n) + C(n)$. Through a naive implementation of matrix multiplication, a complexity of $\mathcal{O}(m^3)$ occurs. Whilst, the matrix addition has a complexity of $\mathcal{O}(m^2)$ in comparison. The total computational cost to calculate w results from the $N - 1$ recursions to calculate $A(N)$ and the final matrix-vector multiplication $A(N)v$. The final matrix multiplication has a complexity of $\mathcal{O}(m^2)$. Therefore, the total cost of calculating w is in $\mathcal{O}(Nm^3)$. Kienitz and Nowaczyk (2011) have called this method the forward method.

3.5.2 The Adjoint Method

To reduce the high computational cost of $\mathcal{O}(Nm^3)$ of the forward method Kienitz and Nowaczyk developed an alternative approach.

Consider the vectors $V(n) \in \mathbb{R}^{m\times 1}$, which are known as the sequence adjoint to the ARP in (3.8) where $\forall 1 \leq n \leq N - 1$:

$$V(n) := D(n)^t V(n + 1), \qquad V(N) := v^t. \tag{3.13}$$

The vectors $\bar{V}(n) \in \mathbb{R}^{q\times 1}$ are known as the total adjoint sequence of the ARP in (3.8), where $\forall 1 \leq n \leq N - 1$:

$$\bar{V}(n) := C(n)^t V(n + 1) + \bar{V}(n + 1), \qquad \bar{V}(N) := 0. \tag{3.14}$$

The formulation of these sequences then allows the result vector of the ARP in (3.8) to be derived through the alternative adjoint method:

$$w = vA(n) = \sum_{n=1}^{N-1} V(N + 1)^t C(n) + V(1)^t A_1 = \bar{V}(1)^t + V(1)^t A_1. \tag{3.15}$$

The explicit sequences for $V(n)$ and $\bar{V}(n)$ are shown to be $\forall 1 \leq n \leq N - 1$:

$$V(n) = \left(\prod_{k=n}^{N-1} D(k)^t \right) v^t, \tag{3.16}$$

$$\bar{V}(n) = \sum_{j=n}^{N-1} C(j)^t V(j + 1). \tag{3.17}$$

Kienitz and Nowaczyk provide justification for the above results. Through the application of the definition in (3.13), the claim in (3.16) follows:

$$\begin{aligned} V(N-1) &= D(N-1)^t v^t \\ V(N-2) &= D(N-2)^t (D(N-1)^t v^t) \\ &= (D(N-2)^t D(N-1)^t) v^t \\ &\dots \\ V(n) &= \left(\prod_{k=n}^{N-1} D(k)^t \right) v^t. \end{aligned}$$

Similarly it can be seen, claim (3.17) follows from definition (3.14):

$$\begin{aligned} \overline{V}(N-1) &= C(N-1)^t V(N) + 0 \\ \overline{V}(N-2) &= C(N-2)^t V(N-1) + C(N-1)^t V(N) \\ &\dots \\ \overline{V}(n) &= \sum_{j=n}^{N-1} C(j)^t V(j+1). \end{aligned}$$

To prove the claim made in (3.15) the result vector w^t is calculated instead of w. The proof utilises algebraic manipulation and re-indexation to justify the claim:

$$w = vA(N) = \left(\sum_{j=1}^{N-1} v \left(\prod_{k=1}^{j-1} D(N-k) \right) C(N-j) \right) + v \left(\prod_{j=1}^{N-1} D(N-j) \right) A_1$$

which implies

$$\begin{aligned} w^t &= \left(\sum_{j=1}^{N-1} C(N-j)^t \left(\prod_{k=1}^{j-1} D(N-k) \right)^t v^t \right) + A_1^t \left(\prod_{j=1}^{N-1} D(N-j) \right)^t v^t \\ &= \left(\sum_{j=1}^{N-1} C(N-j)^t \left(\prod_{k=N-j+1}^{N-1} D(k)^t \right) v^t \right) + A_1^t \left(\prod_{j=1}^{N-1} D(j)^t \right) v^t \\ &= \left(\sum_{n=1}^{N-1} C(n)^t \left(\prod_{k=n+1}^{N-1} D(k)^t \right) v^t \right) + A_1^t \left(\prod_{j=1}^{N-1} D(j)^t \right) v^t \\ &= \left(\sum_{n=1}^{N-1} C(n)^t V(n+1) \right) + A_1^t V(1) \\ &= \bar{V}(1) + A_1^t V(1). \end{aligned}$$

Therefore,

$$w = (w^t)^t = \bar{V}(1)^t + V(1)^t A_1.$$

The significant numerical advantage of the adjoint approach results from the use of vector recursions to calculate the sensitivities. In the forward method (3.8) matrix recursion is undertaken, whereas (3.16) and (3.17) utilise vector recursion only. This results in an overall computational expense of complexity of $\mathcal{O}(Nm^2)$ to calculate the sequences of matrices $V(n)$ and $\bar{V}(n)$. This is the complexity for the entire algorithm of the adjoint method, providing significant computational savings compared to the forward method with its complexity of $\mathcal{O}(Nm^3)$ (Kienitz and Nowaczyk, 2011).

Chapter 4

The LMM and ARP Applications

4.1 The Libor Market Model

The Libor market model (LMM) is one of the most popular families of interest rate models with the pioneering work on the model credited to Brace *et al.* (1997) and Jamshidian (1997). The model earns its popularity through its ability to model the actual traded instruments present in the market and not simply mathematical idealisations such as the short rate or the continuously compounded forward rates. Giles and Glasserman (2006) demonstrated in the context of the LMM how the adjoint method could be applied. This chapter will derive the general framework to be applied on market instruments in the following chapters.

The dynamics of the LMM of Brace *et al.* (1997) are specified below based on the derivation and notation from Glasserman (2013) and Brigo and Mercurio (2007).

Let time t represent the current time. Consider a set of $m+1$ market instrument tenors T_i, for $i = 1, ..., m+1$ with corresponding accrual periods denoted by $\delta_i = T_{i+1} - T_i$, for $i = 1, ..., m$. The simple compounded forward Libor rates at fixed time t for the interval $[T_i, T_{i+1})$ are denoted by $\tilde{L}_i(t) = \tilde{L}(t, T_i, T_{i+1})$ for $i = 1, ..., m$. Let ρ_{ij} represent the correlation between forward rates L_i and L_j. Let $\eta(t)$ represent the index of the next tenor date at time t, $T_{\eta(t)-1} \leq t < T_{\eta(t)}$. The arbitrage-free dynamics of the forward rates are given by

$$\frac{d\tilde{L}_i(t)}{\tilde{L}_i(t)} = \mu_i(\tilde{L}_i(t))dt + \sigma_i^t dW(t), \qquad 0 \leq t \leq T_i, \qquad i = 1, ..., m, \tag{4.1}$$

where W is a d-dimensional standard Brownian motion under the associated risk-adjusted measure, and

$$\mu_i(\tilde{L}(t))) = \sum_{j=\eta(t)}^{i} \frac{\sigma_i^t \sigma_j \rho_{ij} \delta_j \tilde{L}_j(t)}{1 + \delta_j \tilde{L}_j(t)}. \tag{4.2}$$

In order to simulate Libor forward rates under the spot measure, Glasserman (2013) follows the procedure below. A discretized time grid of $0 = t_0 < t_1 < ... < t_{m+1}$ is

selected, which contains all specified tenors. It is often assumed that $t_i = T_i$. Let ρ represent the correlation matrix of the forward rates and apply a Cholesky decomposition upon the matrix to form a lower triangular matrix P, where P_i indicates row i of this matrix.

The Euler-Maruyama scheme can be implemented such that:

$$L_i(n+1) = L_i(n) + \mu_i(L(n))L_i(n)\delta_{i-1} + \sigma_i L_i(n)\sqrt{\delta_{i-1}}P_i Z(n+1),$$

where $Z(n+1) \sim N(0,1)$ is a column vector of m random variables. Under the Heath-Jarrow-Morton framework, to preserve the martingale property of bonds, discretized μ_i were constructed (Glasserman, 2013). However, it is not tractably possible under the spot measure numeraire to preserve this property. Therefore, to implement a more accurate discretization, the log of the process is considered:

$$L_i(n+1) = L_i(n)\exp((\mu_i(L(n)) - \frac{1}{2}\|\sigma_i\|^2)\delta_{i-1} + \sigma_i\sqrt{\delta_{i-1}}P_i Z(n+1)), \tag{4.3}$$

where

$$\mu_i(L(n)) = \sum_{j=\eta(t)}^{i} \frac{\sigma_i\sigma_j\rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)}, \tag{4.4}$$

for $i = 1, ..., m$ where $Z(n+1) \in \mathbb{R}^{m\times 1} \sim N(0,1)$.

In this dissertation three versions of the LMM are considered. The first version, which is the simplest model, is a 1-factor model with constant volatilities. The second version is also a 1-factor model, however volatility is no longer considered constant. To implement this second version work presented by Joshi (2003) on piecewise constant instantaneous forward volatilities is outlined in the following section. The final version of the LMM considered is a full-factor model with correlated Brownian motions. To demonstrate how these correlations are calculated again, implementations proposed by Joshi (2003) are discussed. All the derivations that follow will be under the full factor version with minimal mathematical manipulation required to adapt the functions to the other two model versions.

4.1.1 Piecewise Constant Forward Volatilities

The simulation of forward rates through Monte Carlo methods requires as an input parameter a matrix containing the piecewise constant instantaneous forward volatilities, the most general specification is shown in Table 1:

	$t \in (t_0, t_1]$	$t \in (t_1, t_2]$	$t \in (t_2, t_3]$	...	$t \in (t_{M-1}, t_M]$
$F_1(t)$	$\sigma_{1,1}$	-	-	-	-
$F_2(t)$	$\sigma_{2,1}$	$\sigma_{2,2}$	-	-	-
$\vdots$	...	...	...	...	-
$F_M(t)$	$\sigma_{M,1}$	$\sigma_{M,2}$	$\sigma_{M,3}$	...	$\sigma_{M,M}$

Tab. 4.1: Table of piecewise constant volatilities.

This specification considered by Giles and Glasserman (2006) implies the volatilities of the SDEs and drifts above are determined as $\sigma_i(t_{k-1}) = \sigma_{i,k}$.

Brigo and Mercurio (2007) provide detailed examples of numerous volatility schemes that can be fitted. A single parametric time homogeneous piecewise constant scheme is selected to allow for easier implementation. There, is however, no restriction on the choice of scheme which is to be implemented, provided it is able to correctly populate the volatility matrix.

The instantaneous volatilities of the forward rates $L_i(t)$ can be linked to the volatility of the $T_i \times T_{i+1}$ caplet through the function

$$\sigma_{i,\text{caplet}} = \sqrt{\frac{1}{T_i}\int_0^{t_i} \sigma_i(t)^2 dt} = \sqrt{\frac{1}{T_i}\sum_{k=1}^{i} \sigma_{i,k}^2 \delta_{k-1}}.$$

Therefore, market values are utilised to determine the correct volatility structure. However, this market information is not always available and therefore, to allow for interpolation and extrapolation procedures to calculate the unobserved market volatilities, a parametric form of the caplet volatilities is considered. Joshi (2003) proposes that a forward rate instantaneous volatility is a function of the amount of time until reset. For example, this implies the 6-month forward rate today will have volatility 3 months from now equal to the volatility of the 3-month forward rate today.

Therefore, the volatility, $\sigma_i(t)$, of forward rate $L_i(t)$ ranging from t_j to t_{j+1} has the form

$$\sigma_i(t) = (a + b(t_i - t))\exp(-c(t_i - t)) + d, \tag{4.5}$$

where a, b, c, d are constants. The constant parameters are calibrated from observed market caplet volatilities, which ensures the function $p(t_i - t)$ fits them all optimally.

This simple time homogeneous scheme corresponds to setting $\sigma_{i,k} = \eta_{i-k+1}$, transforming the instantaneous volatility matrix into Table 2 below:

	$t \in (t_0, t_1]$	$t \in (t_1, t_2]$	$t \in (t_2, t_3]$	...	$t \in (t_{M-1}, t_M]$
$F_1(t)$	η_1	-	-	-	-
$F_2(t)$	η_2	η_1	-	-	-
$\vdots$	...	...	...	...	-
$F_M(t)$	η_M	η_{M-1}	η_{M-2}	...	η_1

Tab. 4.2: Time-homogeneous volatilities.

4.1.2 Instantaneous Correlated Forward Rates

For the pricing of certain options considering the amount of decorrelation between neighbouring forward rates is vital. However, there is no obvious market instrument whose price directly reflects the correlation between forward rates. Joshi (2003), however, highlights that one can expect greater correlation in movements of neighbouring forward rates than those far apart. Further, since the predominant movements of the yield curve are upwards and downwards, instantaneous correlations can be expected to be high.

A simplification is then to assume forward rate correlations are a function of time that separates the rates. This provides a simple correlation structure given by

$$\rho_{ij} = \exp(-\beta(\mid t_i - t_j \mid)). \tag{4.6}$$

Where ρ_{ij} represents the correlation between the forward rates L_i and L_j with $\beta \geq 0$ being a constant parameter. The value of β can be determined using additional information contained in swaption volatilities. Joshi (2003) however, finds a value of around $\beta = 0.1$ fits the market well.

4.2 Delta Approximation via an ARP

Recall the Greeks discussed in Section 3.3. This section focuses on Delta, which is the first derivative and therefore describes the first order effects to the option. Consider X to be an approximation of $\tilde{X}$, which is computed through applying an Euler scheme. Therefore the function $p := \mathbb{E}[g(X(N))]$ approximates $\tilde{p}$. The approximated Delta is therefore given by

$$\triangle := \nabla_X(1)(p) \in \mathbb{R}^{1\times m},$$

This approximation is calculated as the result vector

$$\triangle = v\triangle(N) \tag{4.7}$$

of an affine recursion problem (ARP) with matrix recursion $\forall 1 \leq n \leq N-1$ given by

$$\triangle(n+1) = D(n)\triangle(n) \quad \triangle(1) = I. \tag{4.8}$$

The recursing matrix $\triangle(n)$ is defined as

$$\triangle_{ij}(n) := \frac{\partial X_i(n)}{\partial X_j(1)} \quad \forall 1 \leq i,j \leq m, \quad \forall 1 \leq n \leq N. \tag{4.9}$$

The factor matrix $D(n)$ is given by

$$D_{ik}(n) := \frac{\partial F_n^i}{\partial y_k}(X(n),\sigma) \quad \forall 1 \leq i,k \leq m, \quad \forall 1 \leq n \leq N-1, \tag{4.10}$$

and the start vector is defined as

$$v := \nabla g(X(N)) \in \mathbb{R}^{1\times m}. \tag{4.11}$$

To solve the ARP through the forward method recall the general form of the ARP defined in Section 3.5, the components are therefore given by:

$$A(n) = \triangle(n), \quad A(1) = \triangle(1), \quad D(n) = D(n), \quad C(n) = 0, \quad v = \frac{\partial g}{\partial X(N)}.$$

The Delta could also be calculated through application of the adjoint method. If $V(n)$ is the vector sequence adjoint to the recursion in (4.8),

$$V(n) := D(n)^t V(n+1), \quad \forall 1 \leq n \leq N-1, \quad V(N) := \nabla g(X(N))^t. \tag{4.12}$$

This allows for an alternative calculation of $\triangle$ by

$$\triangle = V(1)^t. \tag{4.13}$$

For the above claims to be justified an application of the chain rule of differentiation is required. Recall the definition of the evolution equation in (3.2):

$$X(n+1) := F_n(X(n),\sigma),$$

through application of the chain rule:

$$\begin{aligned}
\triangle_{ij}(n+1) = \frac{\partial X_i(n+1)}{\partial X_j(1)} &= \frac{\partial}{\partial X_j(1)}(F_n(X(n),\sigma)) \\
&= \sum_{k=1}^{m} \frac{\partial F_n^i}{\partial y_k}(X(n),\sigma)\frac{\partial X_k(n)}{\partial X_j(1)} \\
&= \sum_{k=1}^{m} D_{ik}(n)\triangle_{kj}(n) = (D(n)\triangle(n))_{ij} \\
\Rightarrow \triangle(n+1) &= D(n)\triangle(n).
\end{aligned}$$

This proves the claim of (4.8). Under the adjoint method to prove the claim (4.13), recall the formulation described in (3.15) and note that the translation matrices $C(n)$ for this particular ARP are zero:

$$\begin{aligned}\triangle = v\triangle(N) &= \sum_{n=1}^{N-1} V(n+1)^t C(n) + V(1)^t \triangle(1) \\ &= 0 + V(1)^t \times I = V(1)^t.\end{aligned}$$

Through the use of an Euler scheme (4.3), the evolution equation is required to be differentiated to explicitly calculate the entries $D_{ik}(n)$ of the factor matrices $D(n)$. This differentiation can be undertaken on the general form of the Euler scheme, however it is more appropriate to differentiate the evolution equation of a specific model. The LMM is therefore considered and the concepts described above are applied to the model.

4.3 Pathwise Delta in LMM

The computation of Deltas is now specifically addressed within the context of the LMM. The first step is to derive the form of the factor matrices $D(n)$. These matrices represent the derivative of the evolution equation (4.3), which describe the transformation $L_i(n)$ to $L_i(n+1)$, and are shown to be:

$$D_{ii}(n) = \begin{cases} 1 & i < \eta(nh); \\ \frac{L_i(n+1)}{L_i(n)} + \frac{L_i(n+1)\|\sigma_i\|^2 \rho_{ii}\delta_i h}{(1+\delta_i L_i(n))^2} & i \geq \eta(nh); \end{cases}$$

and, for $j \neq i$,

$$D_{ij}(n) = \begin{cases} \frac{L_i(n+1)\sigma_i^t \sigma_j \rho_{ij}\delta_j h}{(1+\delta_j L_j(n))^2} & i > j \geq \eta(nh); \\ 0 & \text{otherwise.} \end{cases}$$

It is now shown how these expressions follow directly from the evolution equation found in Section 3.2. Consider calculating the diagonal elements of $D(n)$, when $i = j$, for the restriction $i < \eta(nh)$. The rate has settled at its maturity and is fixed, $L_i(n+1) = L_i(n)$ for $i < \eta(nh)$. Therefore $D_{ii}(n) = 1$ in this range. Next, consider $i \geq \eta(nh)$:

$$
\begin{aligned}
D_{ii}(n) =& \frac{\partial}{\partial L_i(n)}\left[L_i(n)\exp\left([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right)\right] \\
=& \overbrace{\exp([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h})}^{=\frac{L_i(n+1)}{L_i(n)}} \\
&+ \overbrace{L_i(n)\exp([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h})}^{=L_i(n+1)} \\
&\times \frac{\partial}{\partial L_i(n)}\left[\left(\sigma_i^t \sum_{j=\eta(t)}^{i} \frac{\sigma_j \rho_{ij}\delta_j L_j(n)}{1+\delta_i L_j(n)} - \frac{\|\sigma_i\|^2}{2}\right)h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right] \\
=& \frac{L_i(n+1)}{L_i(n)} + \frac{L_i(n+1)\|\sigma_i\|^2 \rho_{ii}\delta_i h}{(1+\delta_i L_i(n))^2}
\end{aligned}
$$

Next the case where $i \neq j$ and $j < i < \eta(nh)$ is considered. For $i < \eta(nh)$ it is known $L_i(n+1) = L_i(n)$, therefore $D_{ij}(n) = 0$. Next, to construct μ_i the summation occurs over the range $\eta(t)$ to i, therefore the evolution equation will not include any $L_j(n)$ for $j > i$, thus $D_{ij}(n) = 0$ for $\eta(nh) < i < j$.

The final case to consider is where $i > j \geq \eta(nh)$:

$$
\begin{aligned}
D_{ij}(n) &= \frac{\partial}{\partial L_j(n)}\left[L_i(n)\exp\left([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right)\right] \\
&= L_i(n+1) \times \frac{\partial}{\partial L_j(n)}\left[\sigma_i^t \sum_{k=\eta(t)}^{i} \frac{\sigma_k \delta_k \rho_{ik} L_k(n)}{1+\delta_k L_k(n)} - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right] \\
&= \frac{L_i(n+1)\sigma_i^t \sigma_j \rho_{ij}\delta_j h}{(1+\delta_j L_j(n))^2}
\end{aligned}
$$

This shows the results hold. An efficient implementation of this algorithm is proposed by Giles and Glasserman (2006):

$$
\triangle_{ij}(n+1) = \begin{cases} \triangle_{ij}(n) & i < \eta(nh); \\ \frac{L_i(n+1)}{L_i(n)}\triangle_{ij}(n) + L_i(n+1)\sigma_i^t \sum_{k=\eta(nh)}^{i} \frac{\sigma_k \rho_{ik}\delta_k h \triangle_{kj}(n)}{(1+\delta_k L_k(n))^2} & i \geq \eta(nh); \end{cases}
$$

The numerical evaluation of $\triangle_{ij}(n)$ can be computationally expensive. This is due to the number of forward rates m utilised in the LMM being potentially large. To counteract this cost, the adjoint method of solving ARPs can be implemented. The benefits of which are computational savings without the need to introduce any further approximations.

4.4 Adjoint Method under the LMM

Recall that under the adjoint method the desired vector of Deltas is $\frac{\partial g(X(N))}{\partial X(1)}$ given by

$$\begin{aligned}\frac{\partial g}{\partial X(1)} &= \frac{\partial g}{\partial X(N)}\triangle(N) \\ &= \frac{\partial g}{\partial X(N)}D(N-1)D(N-2)...D(1)\triangle(1) \\ &= V(1)^t\triangle(1)\end{aligned}$$

where $V(1)$ can be computed recursively by

$$V(n) = D(n)^t V(n+1), \qquad V(N) = \left(\frac{\partial g}{\partial X(N)}\right)^t.$$

The key insight from the above is that, as in the general case of ARPs mentioned previously, the forward mode requires matrix-matrix multiplications in its recursion, whilst the adjoint mode uses matrix-vector multiplications. Therefore, whereas the forward method requires m^2 variables to be updated at each time step, it is only required to update the m entries of the adjoint variables $V(n)$ in the adjoint method. This can represent significant savings (Giles and Glasserman, 2006).

4.5 Vega Approximation via an ARP

Recall the Greeks discussed in Section 3.3. This section focuses on Vega, which is the first derivative with respect to implied volatility and therefore provides information as to the dependence of the option on volatility. Consider X to be an approximation of $\tilde{X}$, which is computed through applying an Euler scheme. Therefore the function $p := \mathbb{E}[g(X(N))]$ approximates $\tilde{p}$. The approximated Vega is therefore given by

$$\mathcal{V} := \nabla_\sigma(p) \in \mathbb{R}^q.$$

This approximation is calculated as the result vector

$$\mathcal{V} = v\mathcal{V}(N) \tag{4.14}$$

of an ARP with matrix recursion given by

$$\mathcal{V}(n+1) = D(n)\mathcal{V}(n) + B(n) \qquad \mathcal{V}(1) = 0. \tag{4.15}$$

The recursing matrix $\mathcal{V}(n) \in \mathbb{R}^{m\times q}$ is defined as

$$\mathcal{V}_{ij}(n) := \frac{\partial X_i(n)}{\partial \sigma_j} \qquad \forall 1 \le i \le m, \qquad \forall 1 \le j \le q, \qquad \forall 1 \le n \le N. \tag{4.16}$$

The translation matrix $B(n)$ is defined by

$$B_{ij}(n) := \frac{\partial F_n^i}{\partial \sigma_j}(X(n), \sigma) \quad \forall 1 \leq i \leq m, \quad \forall 1 \leq j \leq q, \quad \forall 1 \leq n \leq N-1. \quad (4.17)$$

The start vector v and the factor matrices $D(n)$ are previously defined in (4.11) and (4.10) respectively. To solve the ARP through the forward method recall the general form of the ARP defined in Section 3.5, the components are, given by:

$$A(n) = \mathcal{V}(n), \quad A(1) = \mathcal{V}(1), \quad D(n) = D(n), \quad C(n) = B(n), \quad v = \frac{\partial g}{\partial X(N)}.$$

The Vega can also be calculated through application of the adjoint method. Consider $V(n)$ to be the vector sequence adjoint to the recursion and consider $\bar{V}(n)$ to be the total adjoint sequence. Vega is therefore also given by:

$$\mathcal{V} = v\mathcal{V}(N) = \sum_{n=1}^{N-1} V(n+1)^t B(n) = \bar{V}(1)^t. \quad (4.18)$$

Through the use of the chain rule on the evolution equation defined in (4.3), the claim (4.15) is proved.

$$\begin{aligned}
\mathcal{V}_{ij}(n+1) &= \frac{\partial}{\partial \sigma_j}(X_i(n+1)) = \frac{\partial}{\partial \sigma_j}(F_n^i(X(n), \sigma)) \\
&= \sum_{k=1}^{m} \partial_k(F_n^i)(X(n), \sigma)\frac{\partial X_k(n)}{\partial \sigma_j} + \frac{\partial F_n^i}{\partial \sigma_j}(X(n), \sigma) \\
&= \sum_{k=1}^{m} D_{ik}(n)\mathcal{V}_{kj}(n) + B_{ij}(n) \\
\Rightarrow \mathcal{V}(n+1) &= D(n)\mathcal{V}(n) + B(n).
\end{aligned}$$

It is also seen that $\mathcal{V}_{ij}(1) = \frac{\partial X_i(1)}{\partial \sigma_j} = 0$. Through another application of the chain rule, claim (4.14) is also proved.

$$\mathcal{V} = \nabla_\sigma(g(X(N))) = \nabla(g)(X(N))\nabla_\sigma(X(N)) = v\mathcal{V}(N).$$

Finally claim (4.18) is a result of (3.15):

$$\begin{aligned}
\mathcal{V} = v\mathcal{V}(N) &= \sum_{n=1}^{N-1} V(n+1)^t B(n) + V(1)^t\mathcal{V}(1) \\
&= \bar{V}(1)^t + V(1)^t \times 0 \\
&= \bar{V}(1)^t.
\end{aligned}$$

Through the use of an Euler scheme (4.3), and the specified evolution equation $F(n)$, the form of the translation matrices $B(n)$ can be derived. The derivation can be undertaken based on the general form of the Euler scheme, however it is more appropriate to implement a specific model. The LMM is therefore considered and the concepts described above are applied to the model.

4.6 Pathwise Vega in the LMM

This section contains many similarities with Section 4.3. The major distinction is that the volatility parameters affect the evolution equation as well as the initial conditions. Therefore the ARP for Vega requires a translation term, whereas the ARP for Delta (4.8) does not. The translation matrices $B(n)$, defined by (4.17), assume the form

$$B_{ij}(n) = \begin{cases} \frac{L_i(n+1)\sigma_i\rho_{ii}\delta_i L_i(n)h}{1+\delta_i L_i(n)} + \\ \left(\frac{\mu_i(L(n))}{\sigma_i^t}h - \sigma_i h + P_i Z(n+1)\sqrt{h}\right)L_i(n+1) & i=j\geq\eta(nh); \\ \frac{L_i(n+1)\sigma_i\rho_{ij}\delta_j L_j(n)h}{1+\delta_j L_j(n)} & i>j\geq\eta(nh); \\ 0, & \text{otherwise.} \end{cases} \tag{4.19}$$

Through computing the derivative of the i^{th} element of $F_n(X_n)$ with respect to σ_j equation (4.19) follows. The initial case of $i=j\geq\eta(nh)$ is considered:

$$\begin{aligned} \frac{\partial F_n^i}{\partial\sigma_i}(X(n),\sigma) &= \frac{\partial}{\partial\sigma_i}\left[L_i(n)\exp\left([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right)\right] \\ &= \underbrace{L_i(n)\exp\left([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right)}_{L_i(n+1)} \\ &\quad\times\frac{\partial}{\partial\sigma_i}\left[\left(\sigma_i^t\sum_{j=\eta(t)}^{i}\frac{\sigma_j\rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)} - \frac{\|\sigma_i\|^2}{2}\right)h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right] \\ &= L_i(n+1) \\ &\quad\times\frac{\partial}{\partial\sigma_i}\left[\left(\sigma_i^t\sum_{j=\eta(t)}^{i-1}\frac{\sigma_j\rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{\sigma_i^2\rho_{ii}\delta_i L_i(n)}{1+\delta_i L_i(n)} - \frac{\|\sigma_i\|^2}{2}\right)h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right] \\ &= L_i(n+1)\times\left[\left(\sum_{j=\eta(t)}^{i-1}\frac{\sigma_j\rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{2\sigma_i\rho_{ii}\delta_i L_i(n)}{1+\delta_i L_i(n)} - \sigma_i\right)h + P_i Z(n+1)\sqrt{h}\right] \\ &= L_i(n+1)\times\left[\left(\sum_{j=\eta(t)}^{i}\frac{\sigma_j\rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{\sigma_i\rho_{ii}\delta_i L_i(n)}{1+\delta_i L_i(n)} - \sigma_i\right)h + P_i Z(n+1)\sqrt{h}\right] \\ &= \frac{L_i(n+1)\sigma_i\rho_{ii}\delta_i L_i(n)h}{1+\delta_i L_i(n)} + \left(\frac{\mu_i(L(n))}{\sigma_i^t}h - \sigma_i h + P_i Z(n+1)\sqrt{h}\right)L_i(n+1). \end{aligned}$$

The second case examined is for $i > j \geq \eta(nh)$:

$$\frac{\partial F_n^i}{\partial \sigma_j}(X(n),\sigma) = \frac{\partial}{\partial \sigma_j}\left[L_i(n)\exp\left([\mu_i(L(n)) - \frac{\|\sigma_i\|^2}{2}]h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right)\right]$$
$$= L_i(n+1) \times \frac{\partial}{\partial \sigma_j}\left[\left(\sigma_i^t \sum_{j=\eta(t)}^{i} \frac{\sigma_j \rho_{ij}\delta_j L_j(n)}{1+\delta_j L_j(n)} - \frac{\|\sigma_i\|^2}{2}\right)h + \sigma_i^t P_i Z(n+1)\sqrt{h}\right]$$
$$= \frac{L_i(n+1)\sigma_i \rho_{ij}\delta_j L_j(n) h}{1+\delta_j L_j(n)}.$$

The final cases to consider are $j < \eta(nh)$ or $j > i$. However $B_{ij}(n)$, in both of these cases, is zero as σ_j would not appear in the evolution equation. Therefore (4.19) holds.

4.7 Gamma Approximation via an ARP

Recall the Greeks discussed in Section 3.3. This section focuses on Gamma, which provides information as to the convexity of an interest rate option. Consider X to be an approximation of $\tilde{X}$, which is computed through applying an Euler scheme. Therefore the function $p := \mathbb{E}[g(X(N))]$ approximates $\tilde{p}$. The approximated Gamma is therefore given by

$$\Gamma := \text{Hess}_{X(1)}(p) \in \mathbb{R}^{m\times m}.$$

The computation of Gamma through an ARP requires a number of additional structures and provides a greater challenge than calculating Delta or Vega. The following scalar quantities are defined for any $1 \leq i,j,k \leq m, 1 \leq n \leq N$,

$$\triangle_{ij}(n) := \frac{\partial X_i(n)}{\partial X_j(1)}, \qquad D_{ik}(n) := \frac{\partial}{\partial X_k(n)} F_n^i(x(n),\sigma),$$
$$G_{ik}^{(j)}(n) := \frac{\partial^2 X_i(n)}{\partial X_j(1)\partial X_k(1)}, \qquad E_{jk}^{(i)}(n) := \frac{\partial}{\partial X_j(n)\partial X_k(n)} F_n^i(X(n),\sigma),$$
$$C_{ik}^{(j)}(n) := (\triangle(n)^t E^{(i)}(n)^t \triangle(n))_{kj}, \qquad H_{ij} := \frac{\partial}{\partial X_i(N)\partial X_j(N)} g(X(N)).$$

These populate the entries of the matrices

$$\triangle(n), \quad D(n), \quad G^{(j)}(n), \quad E^{(i)}(n), \quad C^{(j)}(n) \quad \text{and} \quad H,$$

each of which are elements of $\mathbb{R}^{m\times m}$.

Firstly, the matrices $G^{(j)}(n)$ for any $1 \leq j \leq m$, must satisfy the recursion

$$G^{(j)}(n+1) = D(n)G^{(j)}(n) + C^{(j)}(n), \quad \forall 1 \leq n \leq N-1, \quad G^{(j)}(1) = 0. \tag{4.20}$$

To prove the above claim, it is demonstrated how the structure of $G^{(j)}(n+1)$ is inferred by studying the entry $G_{ik}^{(j)}(n+1)$,

$$
\begin{aligned}
G_{ik}^{(j)}(n+1) &= \frac{\partial^2 X_i(n+1)}{\partial X_j(1)\partial X_k(1)} \\
&= \frac{\partial}{\partial X_k(1)}\left(\frac{\partial}{\partial X_j(1)}X_i(n+1)\right) = \frac{\partial}{\partial X_k(1)}(\triangle_{ij}(n+1)) \\
&= \frac{\partial}{\partial X_k(1)}((D(n)\triangle(n))_{ij}) = \sum_{s=1}^{m}\frac{\partial}{\partial X_k(1)}\big(D_{is}(n)\triangle_{sj}(n)\big) \\
&= \sum_{s=1}^{m}\left(\frac{\partial}{\partial X_k(1)}\big(D_{is}(n)\big)\triangle_{sj}(n) + D_{is}(n)\frac{\partial}{\partial X_k(1)}\big(\triangle_{sj}(n)\big)\right) \\
&= \sum_{s=1}^{m}\frac{\partial}{\partial X_k(1)}\left(\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right)\triangle_{sj}(n) + \sum_{s=1}^{m}D_{is}(n)\frac{\partial^2 X_s(n)}{\partial X_j(1)\partial X_k(1)} \\
&= \sum_{s=1}^{m}\frac{\partial}{\partial X_k(1)}\left(\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right)\triangle_{sj}(n) + \sum_{s=1}^{m}D_{is}(n)G_{sk}^{(j)} \\
&= \sum_{s=1}^{m}\underbrace{\frac{\partial}{\partial X_k(1)}\left(\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right)}_{:=(\dagger)}\triangle_{sj}(n) + (D(n)G^{(j)})_{ik}.
\end{aligned}
$$

Consider the expression (†):

$$
\begin{aligned}
\frac{\partial}{\partial X_k(1)}\left[\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right] &= \sum_{t=1}^{m}\frac{\partial}{\partial X_t(n)}\frac{\partial X_t(n)}{\partial X_k(1)}\left(\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right) \\
&= \sum_{t=1}^{m}\frac{\partial X_t(n)}{\partial X_k(1)}\left(\frac{\partial^2}{\partial X_s(n)\partial X_t(n)}F_n^i(X(n),\sigma)\right) \\
&= \sum_{t=1}^{m}\triangle_{tk}(n)E_{st}^{(i)}(n) = \sum_{t=1}^{m}E_{st}^{(i)}(n)\triangle_{tk}(n).
\end{aligned}
$$

Substituting back into above equation

$$
\begin{aligned}
G_{ik}^{(j)}(n+1) &= \sum_{s=1}^{m}\sum_{t=1}^{m}E_{st}^{(i)}(n)\triangle_{tk}(n)\triangle_{sj}(n) + (D(n)G^{(j)})_{ik} \\
&= \sum_{s=1}^{m}(E^{(i)}(n)\triangle(n))_{sk}\triangle_{sj}(n) + (D(n)G^{(j)})_{ik} \\
&= \sum_{s=1}^{m}(\triangle(n)^t E^{(i)}(n)^t)_{ks}\triangle_{sj}(n) + (D(n)G^{(j)})_{ik} \\
&= (\triangle(n)^t E^{(i)}(n)^t\triangle(n))_{kj} + (D(n)G^{(j)})_{ik} \\
&= C_{ik}^{(j)} + (D(n)G^{(j)})_{ik}.
\end{aligned}
$$

Clearly, it follows that

$$
G_{ik}^{(j)}(1) = \frac{\partial^2 X_i(1)}{\partial X_j(1)\partial X_k(1)} = \frac{\partial}{\partial X_j(1)}(\delta_{ik}) = 0.
$$

This proves claim (4.20). Next, consider $U^{(j)} \in \mathbb{R}^{m\times 1}$, which are the vector sequences adjoint to the ARP (4.20), with the start vector given by $v := \nabla(g)(X(N)) \in \mathbb{R}^{1\times m}$. Through the result derived in (3.15), the result vectors $w_{(j)}$ are shown as

$$w_{(j)} = vG^{(j)}(N) = \sum_{n=1}^{N-1} U^{(j)}(n+1)^t C^{(j)}(n) = \bar{U}^{(j)}(1)^t \in \mathbb{R}^{1\times m}. \tag{4.21}$$

Therefore, $w_{(j)}$ are the vectors that comprise the rows of the matrix, $w \in \mathbb{R}^{m\times m}$. The matrix $Y := \triangle(N)^t H \triangle(N) \in \mathbb{R}^{m\times m}$ provides the desired Gamma, which is calculated as

$$\Gamma = w + Y. \tag{4.22}$$

To justify equation (4.22), the structure of the Gamma matrix is inferred by inspecting the individual entries Γ_{jk}, for any $1 \le j,k \le m$

$$\begin{aligned}\Gamma_{jk} &= \frac{\partial^2 g(X(N))}{\partial X_j(1)\partial X_k(1)} = \frac{\partial}{\partial X_j(1)}\left(\sum_{i=1}^{m} \frac{\partial g(X(N))}{\partial X_i(N)}\frac{\partial X_i(N)}{\partial X_k(1)}\right)\\ &= \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right)\triangle_{ik}(N)}_{:=(\star)} + \sum_{i=1}^{m} \frac{\partial g(X(N))}{\partial X_i(N)}\frac{\partial^2 X_i(N)}{\partial X_j(1)\partial X_k(1)}.\end{aligned}$$

The first line follows by applying the multivariable chain rule, whilst the second line is a result of the product rule. Next, consider the expression labeled $(\star)$ above:

$$\begin{aligned}\frac{\partial}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right)\triangle_{ik}(N) &= \sum_{l=1}^{m} \frac{\partial}{\partial X_l(N)}\frac{\partial X_l(N)}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right)\triangle_{ik}(N)\\ &= \sum_{l=1}^{m} \triangle_{lj}(N)\left(\frac{\partial^2 g(X(N))}{\partial X_l(N)\partial X_i(N)}\right)\triangle_{ik}(N)\\ &= \sum_{l=1}^{m} \left(\frac{\partial^2 g(X(N))}{\partial X_l(N)\partial X_i(N)}\right)\triangle_{lj}(N)\triangle_{ik}(N).\end{aligned}$$

Again, the first line follows from applying the multivariable chain rule and the last line follows by identifying that the elements are scalars.

Finally, substitute this expression back

$$\begin{aligned}
\Gamma_{jk} &= \sum_{i=1}^{m}\sum_{l=1}^{m}\left(\frac{\partial^2 g(X(N))}{\partial X_l(N)\partial X_i(N)}\right)\triangle_{lj}(N)\triangle_{ik}(N) + \sum_{i=1}^{m} v_i G_{ik}^{(j)}(N) \\
&= \sum_{l=1}^{m}\sum_{i=1}^{m} H_{il}\triangle_{ij}(N)\triangle_{lk}(N) + (vG^{(j)}(N))_k \\
&= \sum_{l=1}^{m}\sum_{i=1}^{m} H_{li}^{t}\triangle_{ij}(N)\triangle_{lk}(N) + (vG^{(j)}(N))_k \\
&= \sum_{l=1}^{m}(H^t\triangle(N))_{lj}\triangle_{lk}(N) + w_k^{(j)} \\
&= \sum_{l=1}^{m}(\triangle(N)^t H)_{jl}\triangle_{lk}(N) + w_k^{(j)} \\
&= (\triangle(N)^t H\triangle(N))_{jk} + w_k^{(j)} \\
&= Y_{jk} + w_{jk} \\
\Rightarrow \Gamma &= Y + w.
\end{aligned}$$

Therefore (4.22) is proven.

The implementation of an ARP to solve for Gamma is significantly more difficult than that of Delta or Vega. An initial problem is the discounted payoff function g of a derivative instrument generally fails to be twice differentiable, therefore a smoothed approximation of g is required (Giles and Glasserman, 2006). The second problem is that all matrices $\triangle(N)$ are needed for the computation of the matrices $C^{(j)}$. Therefore the forward method must be implemented to calculate the $\triangle(N)$ terms (Kienitz and Nowaczyk, 2011).

Chapter 5

Libor Market Model Caplets

In order to demonstrate the various methods for Delta sensitivity calculation, a caplet, which can typically be calculated by applying the Libor market model (LMM) is considered. This financial option is assumed to be struck at K and runs over the period $[T_m, T_{m+1})$. The function g represents the discounted payoff of the caplet and is given by

$$g = B_1(0)\left(\prod_{i=1}^{m} \frac{1}{1+\delta_i \tilde{L}_i(T_m)}\right)\delta_m \max(\tilde{L}_m(T_m) - K, 0). \tag{5.1}$$

The term $B_1(0) = \frac{1}{1+\delta_0 L_0(T_0)}$ represents the price at time T_0 of a zero coupon bond maturing at T_1 and the term is removed from the product expression as its value is known at T_0.

We are now able to compute the Deltas of this option through an affine recursion problem (ARP) by either implementing the forward or adjoint method solutions outlined in Chapter 4.

To proceed the start vector $v = \frac{\partial g}{\partial X(N)}$, where g is the discounted payoff function above and $X(N) = [L_0(T_m), L_1(T_m), ..., L_j(T_m), ..., L_m(T_m)]$, is derived as follows:

Firstly, consider the case $j < m$:

$$\begin{aligned}
\frac{\partial g}{\partial L_j(T_m)} &= \frac{\partial}{\partial L_j(T_m)}\left[B_1(0)\left(\prod_{i=1}^{m} \frac{1}{1+\delta_i \tilde{L}_i(T_m)}\right)\delta_m \max(\tilde{L}_m(T_m) - K, 0)\right] \\
&= B_1(0)\delta_m(\tilde{L}_m(T_m) - K)^+\left(\prod_{i=1, i\neq j}^{m} \frac{1}{1+\delta_i \tilde{L}_i(T_m)}\right) \times \left[\frac{\partial}{\partial \tilde{L}_j(T_m)}\left(\frac{1}{1+\delta_j \tilde{L}_j(T_m)}\right)\right] \\
&= B_1(0)\delta_m(\tilde{L}_m(T_m) - K)^+\left(\prod_{i=1}^{m} \frac{1}{1+\delta_i \tilde{L}_i(T_m)}\right) \times \frac{-\delta_j}{1+\delta_j \tilde{L}_j(T_m)}.
\end{aligned} \tag{5.2}$$

Secondly, consider the case $j = m$:

$$\begin{aligned}
\frac{\partial g}{\partial L_m(T_m)} &= \frac{\partial}{\partial L_m(T_m)}\left[B_1(0)\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\delta_m \max(\tilde{L}_m(T_m)-K,0)\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\frac{\partial}{\partial L_m(T_m)}\left[\frac{1}{1+\delta_m\tilde{L}_m(T_m)}(\tilde{L}_m(T_m)-K)^+\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right) \\
&\quad\times\left[\frac{-\delta_m}{(1+\delta_m\tilde{L}_m(T_m))^2}(\tilde{L}_m(T_m)-K)^+ + \frac{1}{1+\delta_m\tilde{L}_m(T_m)}\mathbb{1}(\tilde{L}_m(T_m)>K)\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right) \\
&\left[\mathbb{1}(\tilde{L}_m(T_m)>K) - \frac{\delta_m}{(1+\delta_m\tilde{L}_m(T_m))}(\tilde{L}_m(T_m)-K)^+\right]. \qquad (5.3)
\end{aligned}$$

Thus, combining the expressions (5.2) and (5.3) the form of v is given by:

$$v = \begin{cases} B_1(0)\delta_m(\tilde{L}_m(T_m)-K)^+\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\times\frac{-\delta_j}{1+\delta_j\tilde{L}_j(T_m)} & j<m; \\ B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\times & \\ \left[\mathbb{1}(\tilde{L}_m(T_m)>K) - \frac{\delta_m}{(1+\delta_m\tilde{L}_m(T_m))}(\tilde{L}_m(T_m)-K)^+\right] & j=m. \end{cases} \qquad (5.4)$$

It is worth highlighting Glasserman and Zhao (1999) propose the form (5.3) for all $j \leq m$, which is in disagreement with the derivation above. Further, they provide no motivation for this statement. The final set of matrices required are the factor matrices $D(n)$, which are propagated using the formulation in Section 4.3.

To compare relative computational efficiency, the MATLAB functions `tic` and `toc` are utilised to record times. The standard metric in sensitivity calculation for this comparison is the ratio of time taken for the calculation of the price of the financial option and the desired sensitivity, to the time required to calculate the price only (Griewank and Walther, 2008):

$$\frac{\text{Cost}(P+S)}{\text{Cost}(P)} \qquad (5.5)$$

In the above equation the computational time for the calculation is represented by Cost. Whilst, P represents the option price and S represents the desired sensitivity. Since MATLAB is an interpreted language the computational cost ratios for time period N = 5 and less produce nonsensical results and have therefore been excluded from all graphs.

5.1 Numerical Results: Delta

The ARP for Delta may now be solved, as all required components have been derived in closed form:

$$A(n) = \triangle(n), \quad A(1) = \triangle(1) = I, \quad D(n) = D(n), \quad C(n) = 0, \quad v = \frac{\partial g}{\partial X(N)}.$$

To proceed with the calculation of the vector of Deltas for the caplet a discretization scheme is implemented. The forward and adjoint mode, as well as a finite difference scheme are all considered to allow for computational comparisons. Kienitz and Wetterau (2012) provide much of the framework and procedures which are adapted for implementation below, with the code mentioned found in Appendix A.

The first step in the sensitivity calculation is the simulation of LMM realisations. Figure A.1 demonstrates this simulation using the `fullsim` function for the full factor model with time homogeneous volatilities. To implement the LMM for the other two versions slight modifications of this code are required. Secondly, the matrix $V(N)$ is created through implementation of the code in Figure A.2. The final set of matrices required are the factor matrices $D(n)$, which are propagated using the `MatrixDBuilder` function, given in Figure A.3. Subsequently, the code for the $D(n)$ matrices also requires slight adjustments for the other two versions of the LMM.

Consequently, all the required inputs have been derived for the various methods which are to be considered. To compute the forward method, the code found in Figure A.4 is applied, Figure A.5 contains the code for implementation of the adjoint method. Finally, Figure A.6 contains code to implement the finite difference scheme.

We considered all three versions of the LMM for caplets struck at-the-money with discounted payoff (5.1). The results provided below are for the full factor model with time-varying volatilities.

The parameters that were considered are a flat initial forward curve $L_i(0) = 0.07$; $\delta_i = 0.25$, $\forall i = 0, ..., m$. Furthermore, we set $Nh = m$ and $h = \delta_i$ to simplify implementation. The parameters which specify the volatility parametrisation function (4.5) are $a = -0.02$, $b = 0.3$, $c = 2$ and $d = 0.14$. The correlation parameter is $\beta = 0.01$ for equation (4.6). The model is tested over multiple values for N ranging from 1 to 40 and expiry dates of 0.25 years to 10 years, in order to test the relative efficiency of each proposed method for increasing levels of complexity.

The forward and adjoint approaches of the ARP, as well as the finite difference scheme, were implemented to determine the vector of Deltas. Each approach was

evaluated five times for each N with a sample size of 1000, to determine the computational efficiency of each method for increasing values of N. For all numerical estimate graphs an upper and lower three-standard deviation bound for the Monte Carlo estimates has been plotted. The Deltas calculated for all three methods, with an expiry of ten years ($N = 40$), are shown in Figure 5.1.

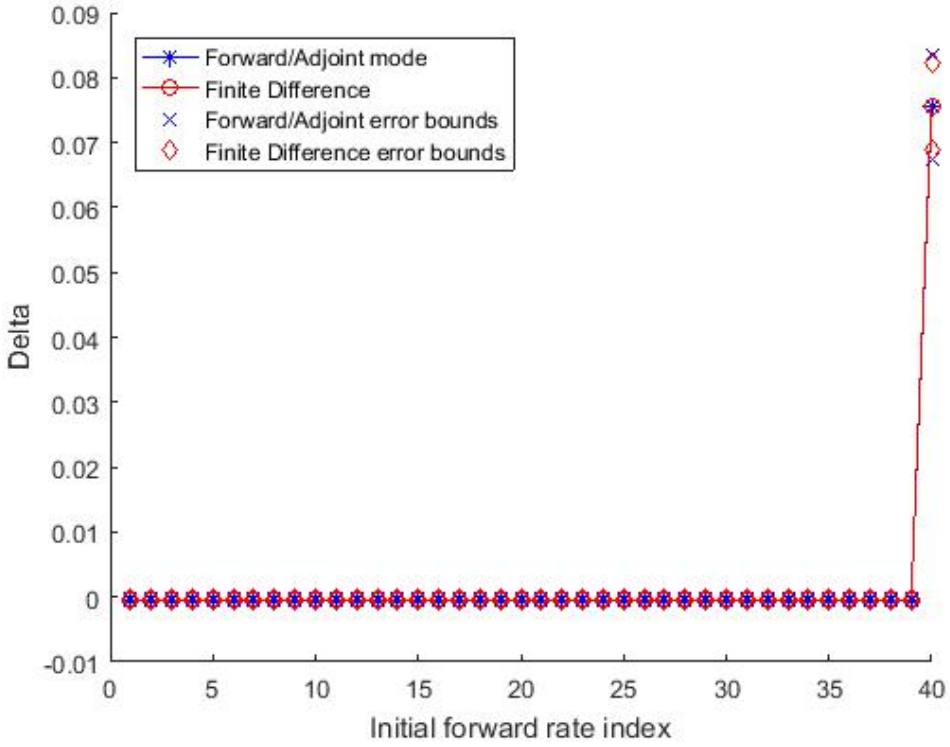

Fig. 5.1: Delta values for the LMM with expiry $N = 40$ via each method.

Figure 5.2 illustrates the relative computational expense as measured by the metric (5.5) for each approach. The graph demonstrates the significant computational expense of the finite difference scheme in comparison to the forward and adjoint mode approaches. Table B.1, found in Appendix B, contains the average runtimes utilised to compute the relative efficiency ratios displayed.

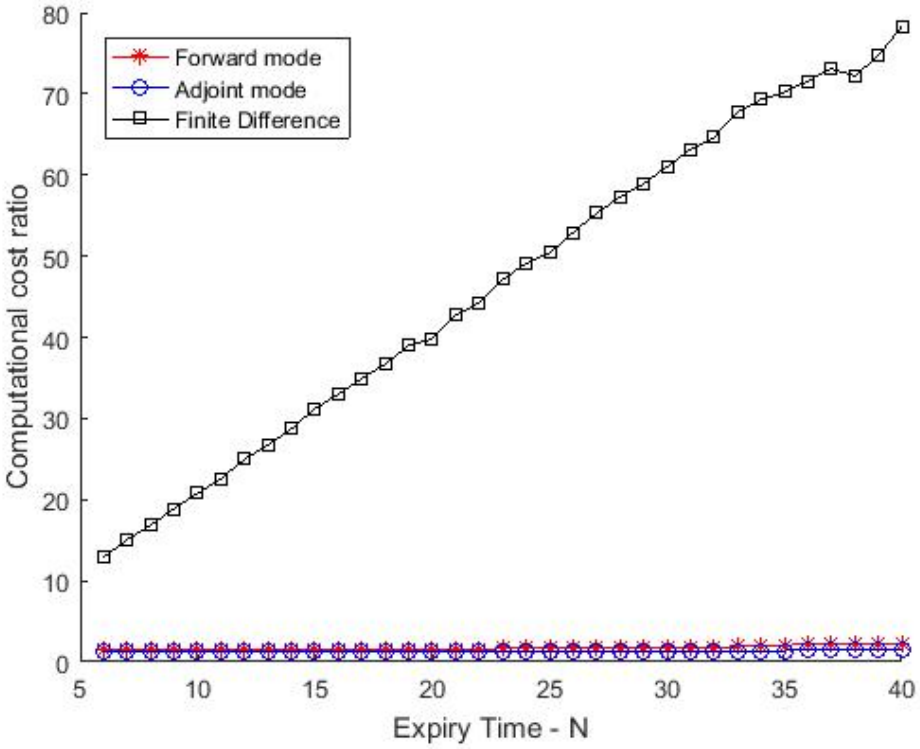

Fig. 5.2: Relative computational cost of each method for increasing tenor numbers N for Delta calculation.

Further, to demonstrate the superior efficiency of the adjoint method over the forward method we isolate these methods from the finite difference scheme in Figure 5.3. The graph illustrates the general improvement in performance of the adjoint method.

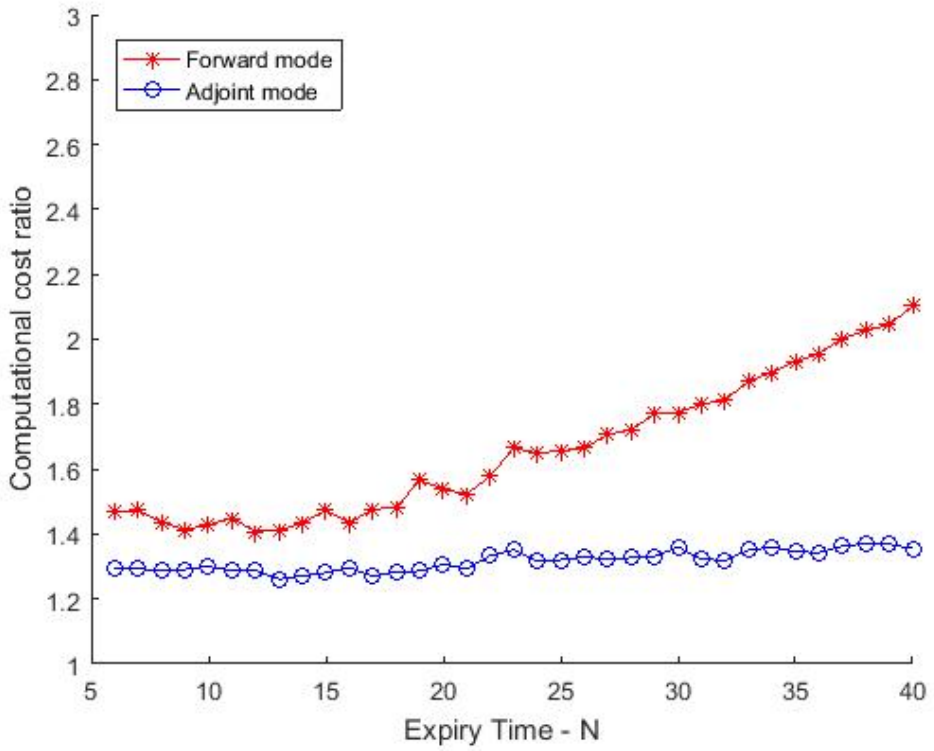

Fig. 5.3: Relative computational cost of forward and adjoint ARP approaches for increasing tenor numbers N for Delta calculation.

5.2 Numerical Results: Vega

Again, the first step in the sensitivity calculation is to simulate the LMM realisations through the code found in Figure A.1 for the full-factor model with time homogeneous volatilities. Secondly, the matrix $V(N)$ is constructed through implementation of the code in Figure A.2 and the factor matrices $D(n)$ are calculated using the code in Figure A.3. Finally, the translation matrices are propagated through the `MatrixBBuilder` function, with the code found in Figure A.7.

Consequently, all the required inputs have been derived for the various methods which are to be considered. Since the translation matrices have to be adapted

to the various methods, to compute the forward method, the code found in Figure A.8 is applied. Figure A.9 contains the modified code for computation of the adjoint method.

To compute the vector of Vegas we consider the same model specifications as outlined in Section 5.1. Again, for a caplet with expiry 10 years ($N = 40$) the values of Vega computed by each method are shown in Figure 5.4.

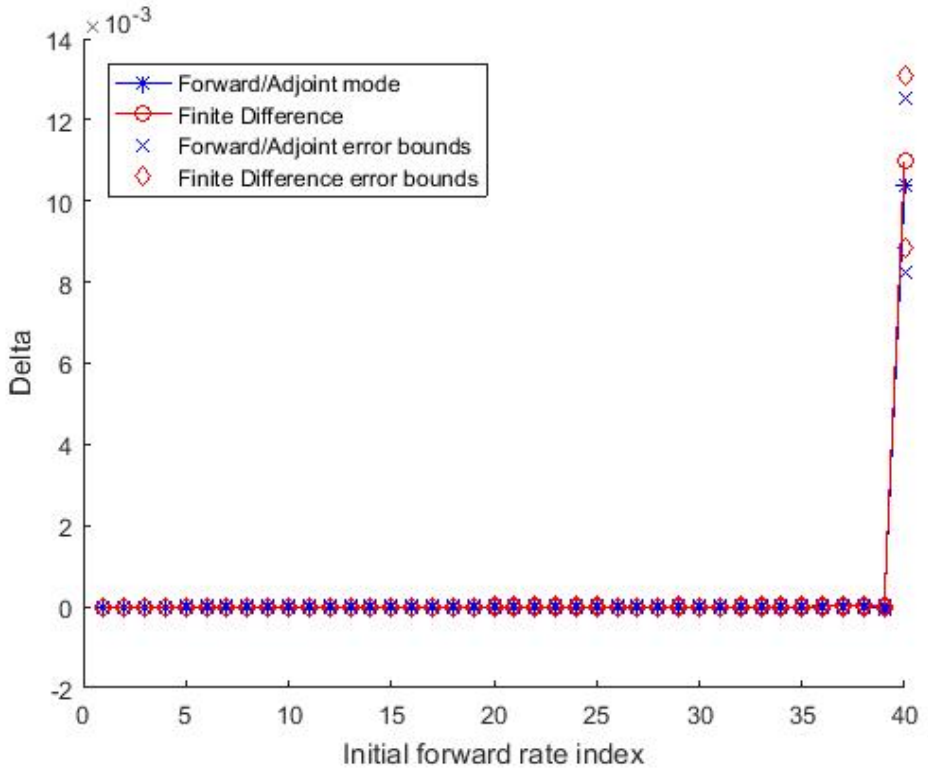

Fig. 5.4: Vega values for the LMM with expiry $N = 40$ via each method.

Figure 5.5 illustrates the relative computational expense as measured by the metric (5.5) for each approach. The graph demonstrates the significant computational expense of the finite difference scheme in comparison to the forward and adjoint mode approaches. Table B.2, found in Appendix B, contains the average run times utilised to compute the relative efficiency ratios displayed.

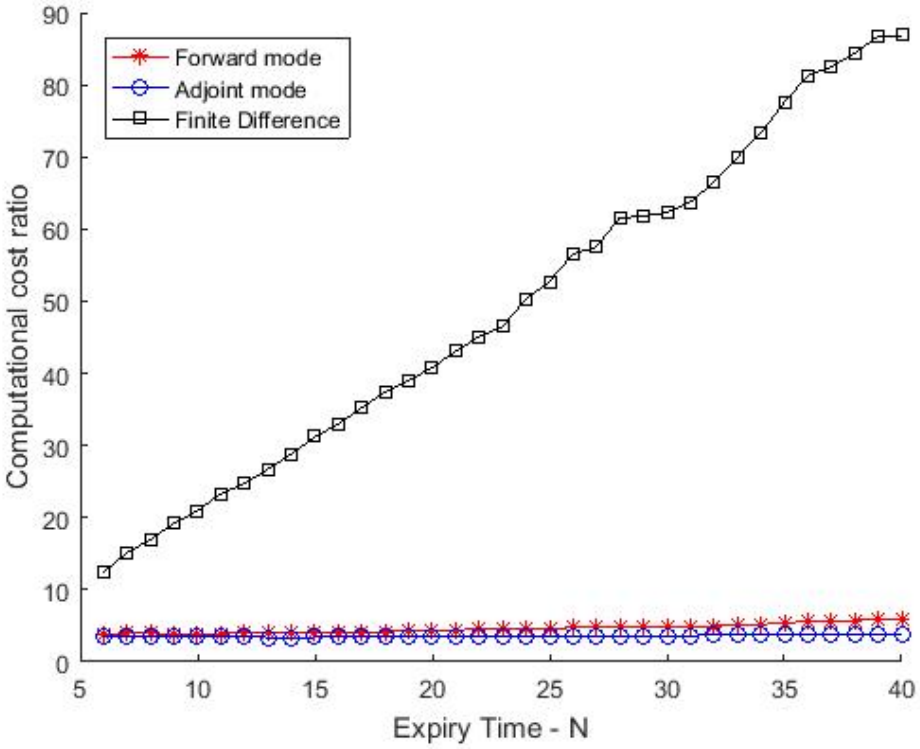

Fig. 5.5: Relative computational cost of each method for increasing tenor numbers N for Vega calculation.

Further, to demonstrate the superior efficiency of the adjoint method over the forward method we isolate these methods from the finite difference scheme in Figure 5.6. The graph illustrates the general improvement in performance of the adjoint method for Vega calculations.

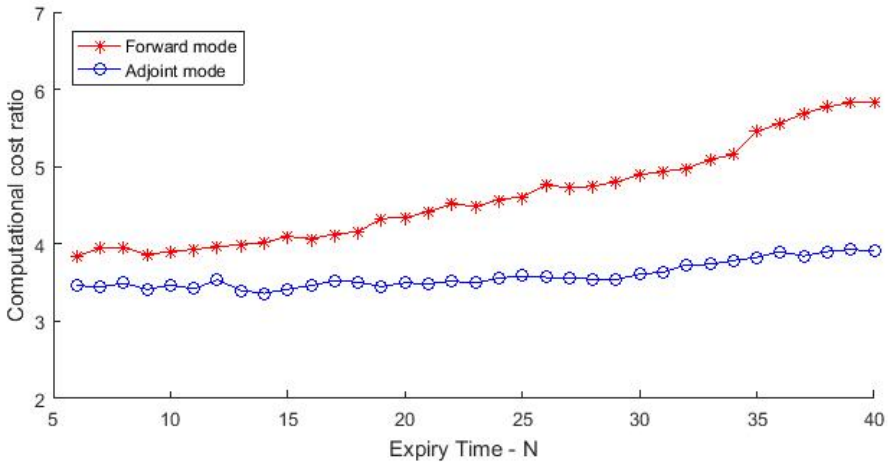

Fig. 5.6: Relative computational cost of forward and adjoint ARP approaches for increasing tenor numbers N for Vega calculation.

Chapter 6

Libor Market Model European Swaptions

A second financial instrument considered is a European swaption. This instrument grants the holder the right, but not the obligation, to enter into an interest rate swap (IRS) at a given future time, the swaption maturity T_r, at a certain strike rate K. The following section is guided by Brigo and Mercurio (2007) and Kienitz and Nowaczyk (2011).

Consider a tenor structure $T = T_0 < T_1 < ... < T_{m+1}$ with intervals $\delta_i = T_{i+1} - T_i$, which are typically a quarter or a half year. A nominal value $\mathcal{N}$ and a payer-or-receiver-factor $\phi \in \{-1, 1\}$ are also required, where ϕ indicates the type of payments the holder will make. If $\phi = 1$, it is a payer swaption and if exercised the holder will make fixed payments in exchange for floating payments. Whereas, $\phi = -1$, indicates a receiver swaption where floating payments will be made in exchange for fixed payments if exercise occurs.

Therefore, the swaption gives the right, but not the obligation, to receive payments at time index n of

$$X_n := X_n(L(n)) := X_n(L_n(n)) := \phi \mathcal{N} \delta_n (L_n(n) - K) \quad \text{for} \quad r \leq n \leq m.$$

If the swaption is exercised at time T_r, the payment X_n is calculated at time index n, but received at time index $m + 1$. Therefore a discount factor from from T_n to T needs to be considered,

$$PV_{n+1} := PV_{n+1}(L(n)) := \prod_{i=0}^{n} \frac{1}{1 + \delta_i L_i(i)}.$$

The payoff of a European swaption, which is exercised at time index r, is

$$g := g(L(m)) := \sum_{\mu=r}^{m} PV_{\mu+1} X_\mu,$$

therefore, for any $r \leq \mu \leq m$, define

$$g_\mu := g_\mu(L(\mu)) := PV_{\mu+1} X_\mu,$$

representing the discounted payoff at time index μ.

This swaption can be compared to entering a swap at time T_r, which would result in payments received at time index n of

$$X_n := X_n(L(n)) := \phi \mathcal{N} \delta_n (L_n(n) - S_{T_r}) \quad \text{for } r \leq n \leq m,$$

where S_{T_r} is the time-T_r swap rate for which the swap has initial value zero. To derive the value of the S_{T_r} swap rate we consider a portfolio which seeks to replicate a payer swap. The portfolio consists of a long position in a floating rate note (FRN), which makes payments of Libor on the tenor dates and a short position in a fixed coupon bond with coupon rate K — both are assumed to have nominal $\mathcal{N} = 1$.

Therefore the time T_r value of the swap is

$$IRS_{T_r} = 1 - (K \sum_{i=r}^{m+1} B_i(T_r)\delta_i + B_{m+1}(T_r)),$$

where $B_i(T_r)$ is the bond price for a specific tenor, time T_i at time T_r and the time T_r value of a FRN is its nominal value, $\mathcal{N} = 1$. The fair swap rate is then the value of K that sets the swap value to zero:

$$S_{T_r} = \frac{1 - B_{m+1}(T_r)}{\sum_{i=r}^{m+1} B_i(T_r)\delta_i}.$$

The net payoff of the swaption is therefore

$$\mathcal{N}\phi(S_T - K)^+ \delta_n \quad \text{at time } T_n \quad \Rightarrow \quad B_n(T_r)\mathcal{N}\phi(S_T - K)^+ \delta_n \quad \text{at time } T_r.$$

Hence the payoff of the swaption at the exercise date is

$$\mathcal{N}\phi(S_{T_r} - K)^+ \sum_{n=r}^{m+1} B_n(T_r)\delta_n.$$

Thus the holder of a payer swaption will choose to exercise if $S_{T_r} \geq K$ and discard the swaption otherwise.

We are now able to compute the Deltas of this option through an affine recursion problem (ARP) by implementing either the forward or adjoint mode solutions outlined in Chapter 4.

To proceed the start vector $v = \nabla(g)(L(m))$, where g is the discounted payoff function above and $\forall r \leq n \leq m : v_\mu := \nabla(g_\mu(L(\mu))$ such that $\nabla(g)(L(m)) = \sum_{\mu=r}^{m} \nabla(g_\mu(L(\mu))$, is derived as follows, assuming exercise has occurred:

Consider any $r \leq n \leq m$ and $1 \leq j \leq n-1$:

$$\begin{aligned} v^j_{(n)} = \frac{\partial g_n}{\partial L_j(j)} &= \frac{\partial}{\partial L_j(j)}\left[\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\phi\mathcal{N}\delta_n(L_n(n)-K)\right] \\ &= \phi\mathcal{N}\delta_n(L_n(n)-K)\prod_{i=1,i\neq j}^{n}\frac{1}{1+\delta_i L_i(i)}\left(\frac{-\delta_j}{(1+\delta_j L_j(j))^2}\right) \\ &= \phi\mathcal{N}\delta_n(L_n(n)-K)\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\left(\frac{-\delta_j}{1+\delta_j L_j(j)}\right) \\ &= \phi\mathcal{N}\delta_n\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\times\frac{\delta_j(K-L_n(n))}{1+\delta_j L_j(j)}. \end{aligned} \tag{6.1}$$

Secondly, consider the case $j = n$:

$$\begin{aligned} v^j_{(n)} = \frac{\partial g_n}{\partial L_n(n)} &= \frac{\partial}{\partial L_n(n)}\left[\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\phi\mathcal{N}\delta_n(L_n(n)-K)\right] \\ &= \frac{\partial}{\partial L_n(n)}\left[\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\right]\phi\mathcal{N}\delta_n(L_n(n)-K) \\ &\quad + \frac{\partial}{\partial L_n(n)}\left[\phi\mathcal{N}\delta_n(L_n(n)-K)\right]\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)} \\ &= \phi\mathcal{N}\delta_n(L_n(n)-K)\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\left(\frac{-\delta_n}{1+\delta_n L_n(n)}\right) + \phi\mathcal{N}\delta_n\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)} \\ &= \phi\mathcal{N}\delta_n\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}\times\left(1-\delta_n\frac{L_n(n)-K}{1+\delta_n L_n(n)}\right). \end{aligned} \tag{6.2}$$

Finally, when $n+1 \leq j \leq m$ then $v^j_{(n)} = 0$. Thus, combining this fact with expressions (6.1) and (6.2), the form of the start $v_{(n)} \in \mathbb{R}^{1\times m}$ is given by:

$$v = \alpha_n \cdot \begin{cases} \frac{\delta_j(K-L_n(n))}{1+\delta_j L_j(j)}, & 1 \leq j \leq n-1, \\ \left(1-\delta_n\frac{L_n(n)-K}{1+\delta_n L_n(n)}\right) & j = n, \\ 0, & n+1 \leq j \leq m, \end{cases} \tag{6.3}$$

where

$$\alpha_n := \phi\mathcal{N}\delta_n\prod_{i=1}^{n}\frac{1}{1+\delta_i L_i(i)}.$$

The final set of matrices required are the factor matrices $D(n)$, which are propagated using the formulation in Section 4.3. If the forward method is considered the following recursion is employed for the matrices $\triangle^\mu(n), r \leq \mu \leq m$:

$$\forall 1 \leq n \leq \mu-1 : \triangle^\mu(n+1) = D(n)\triangle^\mu(n), \quad \triangle^\mu(1) = I, \quad \triangle^\mu(n) = \triangle(n).$$

If the adjoint method is considered, let V^{μ} be the vector sequence adjoint to the recursion,

$$\forall 1 \leq n \leq \mu - 1 : V^{(\mu)}(n) = D(n)^t V^{(\mu)}(n+1), \quad V^{(\mu)}(\mu) := v_{(\mu)}^t := \nabla g_{(\mu)}(L_\mu(\mu))^t.$$

This leads to $\triangle^{(\mu)} = v_{(\mu)}\triangle^{(\mu)}(\mu) = V^{(\mu)}(1)^t$ by application of the definition of the adjoint ARP (3.15) to $\triangle^{(\mu)}$.

Further, for any $r \leq \mu \leq m,\ 1 \leq j \leq m,\ 1 \leq i \leq \mu,\ \triangle_{ij}(m) = \triangle_{ij}(\mu)$. Since the definition (4.3) implies

$$\triangle_{ij}(m) = \frac{\partial L_i(m)}{\partial L_j(1)} = \frac{\partial L_i(\mu)}{\partial L_j(1)} = \triangle_{ij}(\mu).$$

Finally let $\triangle$ represent the Delta of the payoff g of the European swaption. Then

$$\triangle = v\triangle(m) = \sum_{\mu=r}^{m} v_{(\mu)}\triangle_{(\mu)} = W(1)^t$$

where the sequence $W(n) = \sum_{\mu=\max(n,r)}^{N} V^{(\mu)}(n)$.

6.1 Numerical Results: Delta

To proceed with the calculation of the vector of Deltas for the European swaption a discretization scheme is selected. The forward and adjoint mode, as well as a finite difference scheme are all considered to allow for computational comparisons. Again, Kienitz and Wetterau (2012) provide much of the framework and procedures which are adapted for implementation below, with the code found in Appendix A.

The first step in the sensitivity calculation is to simulate the Libor market model (LMM) realisations. We again use the `fullsim` function found in Figure A.1 to simulate the full factor model with time homogeneous volatilities. Second, through the `CalcSwapRatesPayoff` function the payoffs and the swap rates at time T_r are calculated in Figure A.10. Next, after determining which paths are optimal to exercise the set of start vectors v are calculated in the `BuildStartVectorsV` function found in Figure A.12. The final set of matrices required are the factor matrices $D(n)$, which are again propagated using the `MatrixDBuilder` function, given in Figure A.3.

Consequently, all the required inputs have been derived for the various methods that are to be implemented. To compute the forward method, the code found in Figure A.13 is applied, Figure A.14 contains the code for implementation to calculate the adjoint method. Finally, Figure A.15 contains code to implement the finite difference scheme.

We consider all three versions of the LMM for European swaptions struck at-the-money with discounted payoff g. The results provided below are for the full factor model with time-varying volatilities.

The parameters that were used are the same as those specified in Section 5.1. The model is tested over multiple values for N ranging from 1 to 40 and expiries of 0.25 years to 10 years, in order to test the relative efficiency of each proposed method over increasing levels of complexity.

The forward and adjoint approaches of the ARP as well as the finite difference scheme were implemented to determine the vector of Deltas. Each approach was implemented five times for each N with a sample size of 1000, to determine the computational efficiency of each method over increasing values of N. The Deltas calculated for all three methods, with an expiry of ten years ($N = 40$), are shown in Figure 6.1.

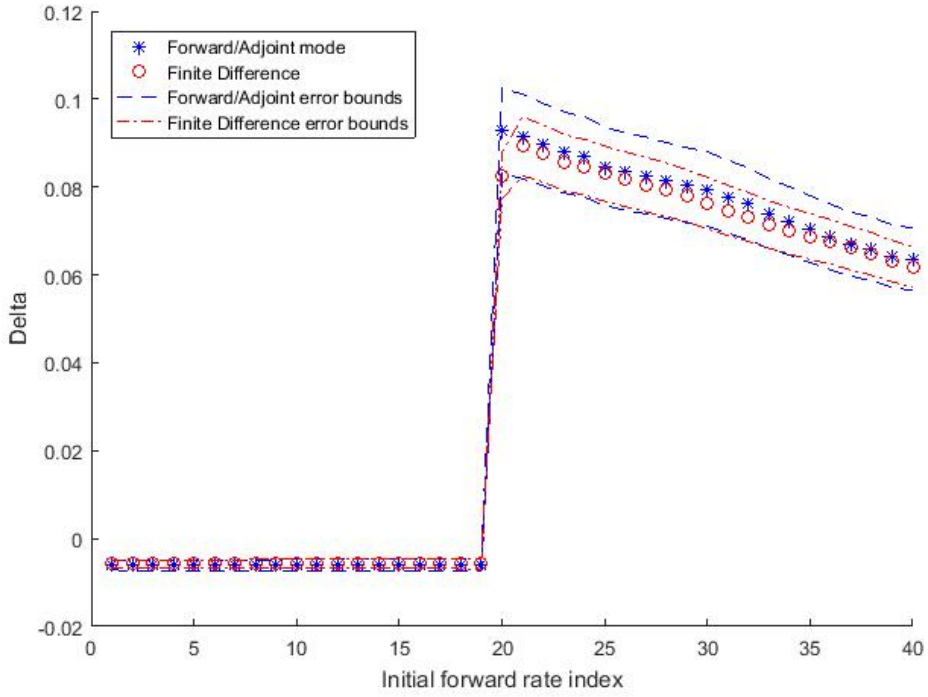

Fig. 6.1: Delta values for the LMM with expiry $N = 40$ via each method.

Figure 6.2 illustrates the relative computational expense as measured by the metric (5.5) for each approach. The graph demonstrates the significant computational expense of the finite difference scheme in comparison to the forward and adjoint mode approaches. Table B.3, found in Appendix B, contains the average run times utilised to compute the relative efficiency ratios displayed.

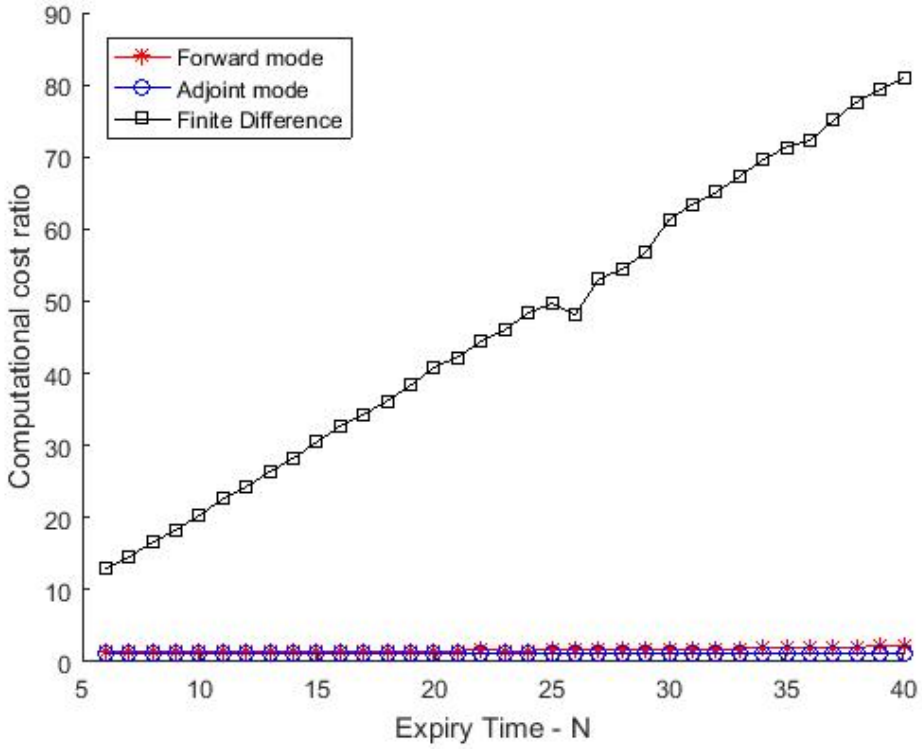

Fig. 6.2: Relative computational cost of each method for increasing tenor numbers N for Delta calculation.

Further, to demonstrate the superior efficiency of the adjoint method over the forward method we isolate these methods from the finite difference scheme in Figure 6.3. The graph illustrates the general improvement in performance of the adjoint method.

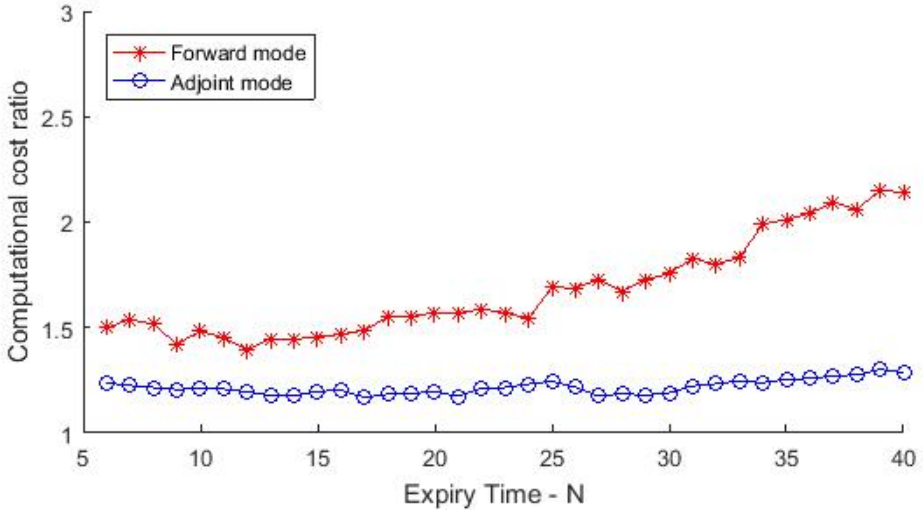

Fig. 6.3: Relative computational cost of forward and adjoint ARP approaches for increasing tenor numbers N for Delta calculation.

6.2 Numerical Results: Vega

To calculate the vector of Vegas for swaptions, the LMM, the payoffs, swap rates and factor matrices are all calculated as for Delta. The translation matrices $B(n)$ are computed through the `MatrixBBuilder` function, with the code found in Figure A.7.

Consequently, all the required inputs have been derived for the various methods that are to be implemented. Since the translation matrices have to be adapted to the various methods, to compute the forward method, the code found in Figure A.13 is applied. Figure (A.14) contains the modified code for computation of the adjoint method.

To compute the vector of Vegas we consider the same model specifications as outlined in Section 6.1. For a European swaption with expiry 10 years ($N = 40$) the values of Vega computed by each method are shown in Figure 6.4.

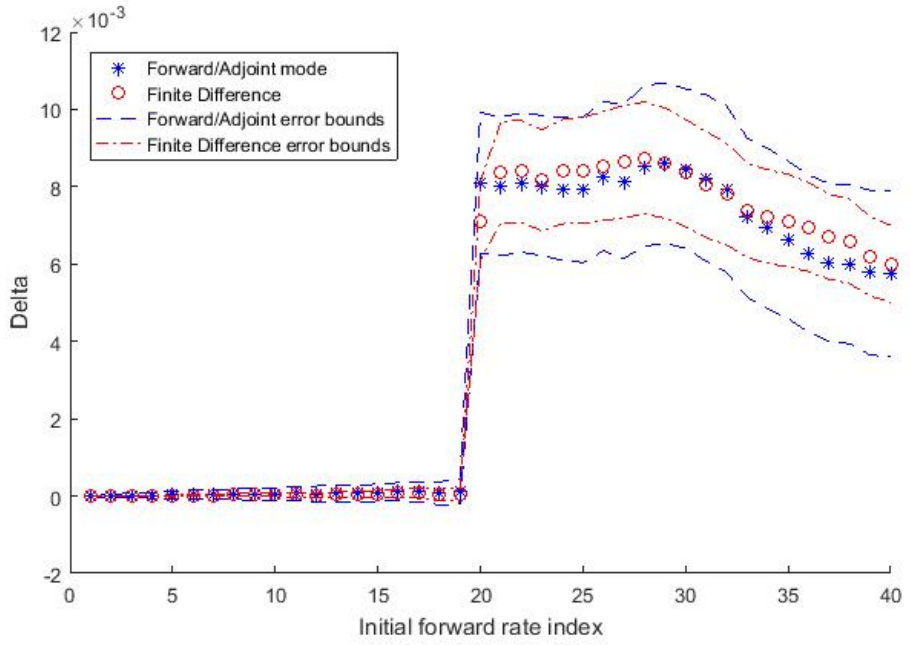

Fig. 6.4: Vega values for the LMM with expiry $N = 40$ via each method.

Figure 6.5 illustrates the relative computational expense as measured by the metric (5.5) for each approach. The graph demonstrates the significant computational expense of the finite difference scheme in comparison to the forward and adjoint mode approaches. Table B.4, found in Appendix B, contains the average run times utilised to compute the relative efficiency ratios displayed.

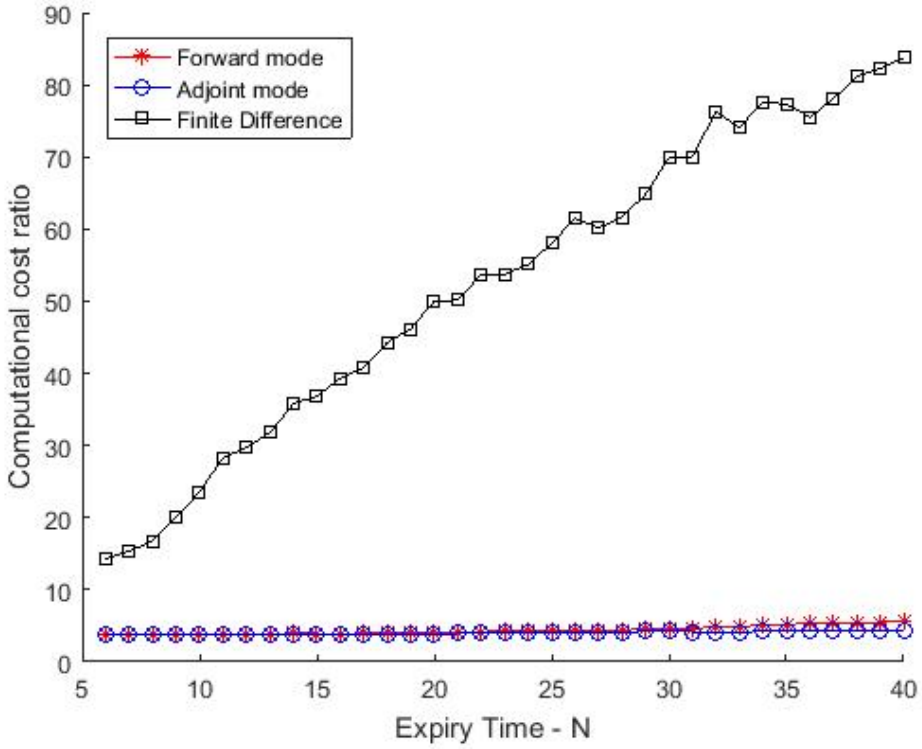

Fig. 6.5: Relative computational cost of each method for increasing tenor numbers N for Vega calculation.

Further, to demonstrate the superior efficiency of the adjoint method over the forward method we isolate these methods from the finite difference scheme in Figure 6.6. The graph illustrates the general improvement in performance of the adjoint method for Vega calculations.

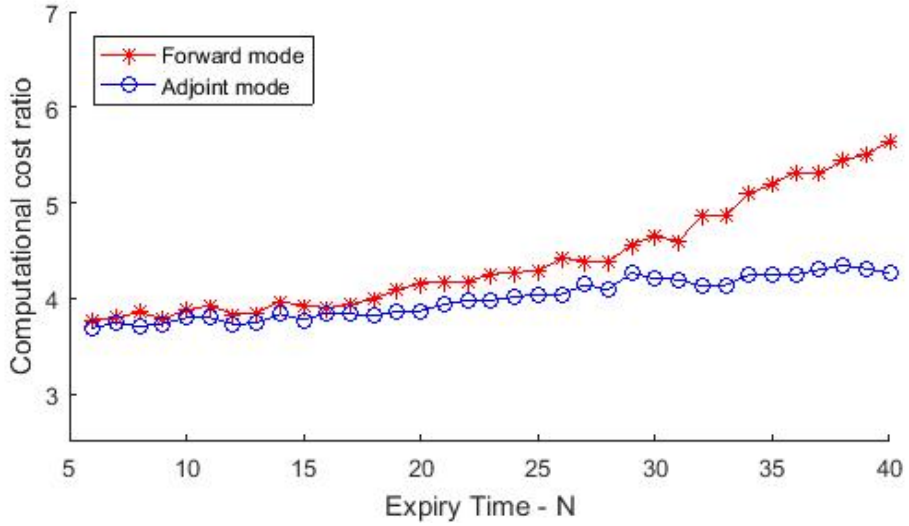

Fig. 6.6: Relative computational cost of forward and adjoint ARP approaches for increasing tenor numbers N for Vega calculation.

Chapter 7

Conclusion

This dissertation has sought to demonstrate how the accurate and efficient calculation of sensitivities of financial instruments can occur. The affine recursion problem (ARP) framework was introduced and explained, whilst demonstrating how an adjoint approach can be applied to solve the ARPs.

An introduction of the topic and discussion of the relevant literature provided the background significance of the challenge. Thereafter, the concept of the general ARP was outlined, with the two solution choices, the forward mode and the adjoint mode discussed. The key concept of the adjoint approach being a reversal of the order of differentiation from the forward approach, working backwards from expiry of the relevant option to the current time, instead of the chronological sequence of differentiation of the forward method. It was then shown how the intuitive forward approach relies on a matrix-matrix multiplication in its recursion. Whereas, through a mathematical manipulation, the adjoint approach implements a matrix-vector multiplication in its recursion. As a result significant computational saving is possible, since the forward method has a computational complexity of $\mathcal{O}(Nm^3)$ compared to $\mathcal{O}(Nm^2)$ of the adjoint approach, where the number of recursion steps is N and the factor matrices are $D(n) \in \mathbb{R}^{m\times m}$.

The Libor market model (LMM) was introduced where it was shown how a one-factor model with constant volatility could be transformed into a full factor model with time homogeneous volatilities. The discretization schemes of the model were presented as well as the construction of the appropriate volatility and correlation structures. A subset of the Greeks — namely Delta, Gamma and Vega were derived in the context of the LMM, with both the forward and adjoint approaches provided.

Two financial instruments were then presented to demonstrate empirically the relative computational expenses of the various methods discussed. The caplet and European swaption examples tested both showed the significant computational cost experienced by the finite difference scheme in comparison to the forward and adjoint modes. Further, it was shown how the adjoint method experienced signifi-

cant computational efficiencies compared to the forward method.

This dissertation sought to explore various versions of the LMM as well as provide a detailed introduction and understanding into how efficient sensitivity calculation can be achieved through the adjoint approach to solving the ARP. There remain many areas of further research, from the implementation of a factor reduction Libor scheme to applying adjoint methods to higher order sensitivities and financial derivatives with Bermudan-style payoffs. The adjoint approach has led to many interesting applications in its relative short existence in a financial context, with users adopting the techniques to solve a variety of interesting problems. It is expected that the adjoint approach will continue to be a source of keen interest to practitioners and researchers as the knowledge and powerful potential of the method continues to be understood.

Bibliography

Beveridge, C., Joshi, M. S. and Wright, W. M. (2010). Efficient Pricing and Greeks in the Cross-Currency Libor Market Model.
URL: *http://dx.doi.org/10.2139/ssrn.1662229*

Boyle, P., Broadie, M. and Glasserman, P. (1997). Monte Carlo Methods for Security Pricing, *Journal of Economic Dynamics and Control* **21**(8): 1267–1321.

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

Brigo, D. and Mercurio, F. (2007). *Interest Rate Models-Theory and Practice: With Smile, Inflation and Credit*, Springer.

Broadie, M. and Glasserman, P. (1996). Estimating Security Price Derivatives Using Simulation, *Management Science* **42**(2): 269–285.

Capriotti, L. (2011). Fast Greeks by Algorithmic Differentiation, *The Journal of Computational Finance* **14**(3): 3–35.

Capriotti, L. and Giles, M. (2012). Adjoint Greeks Made Easy, *Risk* **25**(9): 92.

Capriotti, L. and Giles, M. B. (2010). Fast Correlation Greeks by Adjoint Algorithmic Differentiation, *Risk* **23**(4): 79–83.

Capriotti, L., Jiang, Y. and Macrina, A. (2015). Real-time risk management: An AAD-PDE approach, *International Journal of Financial Engineering* **2**(4): 1550039.

Capriotti, L., Jiang, Y. and Macrina, A. (2017). AAD and least-square Monte Carlo: Fast Bermudan-style options and XVA Greeks, *Algoritmic Finance* **6**(1-2): 35–49.

Capriotti, L., Lee, S. J. and Peacock, M. (2011). Real Time Counterparty Credit Risk Management in Monte Carlo, *Risk* pp. 86–90.

Chan, J. H., Joshi, M. S. and Zhu, D. (2015). First-and second-order Greeks in the Heston model, *Risk* **17**(4): 19–69.

Denson, N. and Joshi, M. S. (2010). Fast Greeks for Markov-Functional Models Using Adjoint PDE Methods.
URL: *http://dx.doi.org/10.2139/ssrn.1618026*

Giles, M. and Glasserman, P. (2006). Smoking Adjoints: fast evaluation of Greeks in Monte Carlo calculations, *Risk* **19**(1): 88–92.

Glasserman, P. (2013). *Monte Carlo methods in financial engineering*, Springer.

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

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

Homescu, C. (2011). Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance.
URL: *http://dx.doi.org/10.2139/ssrn.1828503*

Jamshidian, F. (1997). Libor and swap market models and measures, *Finance and Stochastics* **1**(4): 293–330.

Joshi, M. and Pitt, D. (2010). Fast Sensitivity Computations for Monte Carlo Valuation of Pension Funds, *ASTIN Bulletin: The Journal of the IAA* **40**(2): 655–667.

Joshi, M. S. (2003). *The Concepts and Practice of Mathematical Finance*, Vol. 1, Cambridge University Press.

Joshi, M. and Yang, C. (2011a). Algorithmic Hessians and the fast computation of cross-gamma risk, *IIE Transactions* **43**(12): 878–892.

Joshi, M. and Yang, C. (2011b). Fast delta computations in the swap-rate market model, *Journal of Economic Dynamics and Control* **35**(5): 764–775.

Kallenberg, O. (2006). *Foundations of Modern Probability*, Springer Science & Business Media.

Kienitz, J. and Nowaczyk, N. (2011). Affine recursion problem and a general framework for adjoint methods for calculating sensitivities for financial instruments.
URL: *http://dx.doi.org/10.2139/ssrn.1957082*

Kienitz, J. and Wetterau, D. (2012). *Financial modelling: Theory, implementation and practice with MATLAB source*, John Wiley & Sons.

Leclerc, M., Liang, Q. and Schneider, I. (2009). Fast Monte Carlo Bermudan Greeks, *Risk* **22**(7): 84.

Xu, W., Chen, X. and Coleman, T. F. (2016). The Efficient Application of Automatic Differentiation for Computing Gradients in Financial Applications, *Journal of Computational Finance* **19**(3): 71–96.

Appendix A

MATLAB Code for the Implementation of the Forward and Adjoint ARP Approaches

A.1 Code for Implementation of Euler Approximations for Caplet Delta in the LMM

```
function [Libors] = fullsim(L_Init,m,path,Sig_mat,tau,Z_norm,rho,Choles)
Libors = zeros(m,m,path);
S = zeros(m,1);
for idx = 1:path
    Libors(:,1,idx)=L_Init;
end
for idx2 = 1:path
    for n = 1:m-1
        S = Sig_mat(:,n).*tau.*Libors(:,n,idx2)./(1+tau.*Libors(:,n,idx2));
        S(1:n)=0;
        mu = zeros(m,1);
        for k = n+1:m
            temp_S = S;
            correl = temp_S.*rho(:,k);
            sum_cor = cumsum(correl);
            mu(k) = sum_cor(k);
        end
        S = exp( Sig_mat(:,n).*((mu-Sig_mat(:,n)*0.5)*tau(n)+...
            Choles*Z_norm(:,n,idx2)*sqrt (tau(n))));
        Libors(n+1:m,n+1,idx2)=Libors(n+1:m,n,idx2).*S(n+1:m);
    end
end
for i = 2:m
    Libors(1:i-1,i,:)=Libors(1:i-1,i-1,:);
end
end
```

Fig. A.1: The `fullsim` function for the simulation of LMM realisations (Kienitz and Wetterau (2012)).

```
VN_matrix = zeros(m,No_paths);
MaxTest = max((reshape(Libors(m,m,:),1,No_paths))-K,0);
Indicate = ((reshape(Libors(m,m,:),1,No_paths))>K);
Disc = 1./(1+repmat(tau,1,No_paths).*(reshape(Libors(:,m,:),m,No_paths)));
Pres = prod(Disc);
VN_matrix(end,:) = tau(end).*Pres.*(Indicate-tau(end).*Disc(end,:).*MaxTest);
PriceV = mean(Pres.*MaxTest*tau(end));
for i = 1:m-1
    VN_matrix(i,:)=tau(end).*Pres.*MaxTest.*(-tau(i).*Disc(i,:));
end
```

Fig. A.2: Propagation of the $V(N)$ matrix

```
function D = MatrixDBuilder(LIBORs,m,Sig_mat,Tau,No_paths,rho)

D = zeros(m,m,m-1);
Time = Tau(1);

for n = 1:m-1

    v = LIBORs(:,n+1)./LIBORs(:,n) + ...
        Sig_mat(:,n).^(2).*(Time*Time).*...
        LIBORs(:,n+1)./((1+Time.*LIBORs(:,n)).^(2));
    v(1:n) = 1;
    D(:,:,n) = diag(v);

    x = Sig_mat(:,n).*Time*Time./((1+Time.*LIBORs(:,n)).^(2));
    y = LIBORs(:,n+1).*Sig_mat(:,n);
    A = rho(:,:).*y.*x.';
    port = n+1:m;
    D(port,port,n) = D(port,port,n) + tril(A(port,port),-1);
end

end
```

Fig. A.3: The `MatrixDBuilder` function for the construction of the factor $D(n)$ matrices (Kienitz and Wetterau (2012)).

```
For_Delta = zeros(m,1);
for p = 1:No_paths
    D = MatrixDBuilder(Libors,m,Z_mat,tau,p,rho);
    m = size(D,1);
    N = size(D,3)+1;
    A = zeros(m,m,N);
    Delta0 = eye(m);
    A(:,:,1) = Delta0;
    V = VN_matrix(:,p);
    for n = 1:N-1
        A(:,:,n+1) = D(:,:,n)*A(:,:,n);
    end
    w = V.'*A(:,:,N);
    For_Delta = For_Delta+w;
end
For_Delta = For_Delta/No_paths;
For_Delta = For_Delta';
```

Fig. A.4: Implementation of the ARP forward method (Kienitz and Wetterau (2012)).

```
Adj_Delta = zeros(m,1);
for p = 1:No_paths
    D = MatrixDBuilder(Libors,m,Z_mat,tau,p,rho);
    N = size(D,3)+1;
    V = VN_matrix(:,p);
    for n = N-1:-1:1
        V = D(:,:,n).'*V;
    end
    Adj_Delta = Adj_Delta+V;
end
Adj_Delta = Adj_Delta/No_paths;
```

Fig. A.5: Implementation of the ARP adjoint method (Kienitz and Wetterau (2012)).

```
Fin_Delta = zeros(m,1);
for i = 1:m
    Libor_Init_L=Libor_Init;
    Libor_Init_L(i)=Libor_Init_L(i)-eps;
    Libors_L=fullsim(Libor_Init_L,m,No_paths,Z_mat,tau,Z,rho,Choles);
    MaxTest_L = max((reshape(Libors_L(m,m,:),1,No_paths))-K,0);
    Disc_L = 1./(1+repmat(tau,1,No_paths).*...
    (reshape(Libors_L(:,m,:),m,No_paths)));
    Pres_L = prod(Disc_L);
    Value_L = mean(Pres_L.*MaxTest_L*tau(end));

    Libor_Init_R=Libor_Init;
    Libor_Init_R(i)=Libor_Init_R(i)+eps;
    Libors_R=fullsim(Libor_Init_R,m,No_paths,Z_mat,tau,Z,rho,Choles);
    MaxTest_R = max((reshape(Libors_R(m,m,:),1,No_paths))-K,0);
    Disc_R = 1./(1+repmat(tau,1,No_paths).*...
    (reshape(Libors_R(:,m,:),m,No_paths)));
    Pres_R = prod(Disc_R);
    Value_R = mean(Pres_R.*MaxTest_R*tau(end));

    Fin_Delta(i) = (Value_R-Value_L)/(2*eps);
end
Fin_Delta=Fin_Delta';
```

Fig. A.6: Implementation of a finite difference scheme (Kienitz and Wetterau (2012)).

A.2 Code for Implementation of Euler Approximations for Caplet Vega in the LMM

```
function B = MatrixBBuilder(LIBORs,Z,m,Z_mat,Tau,No_path, Correl, Choles)

B = zeros(m,m,m-1);
Time = Tau(1);

for n = 1:m-1

    A  = LIBORs(:,n+1) .* Z_mat(:,n) .* Time .* Time .*...
    LIBORs(:,n) ./ (1 + Time .* LIBORs(:,n)).';
    A = tril(A(n+1:m,n+1:m),-1);
    B(n+1:m,n+1:m,n) = A;
    B(:,:,n) = B(:,:,n).* Correl(:,:);

    mu = Time.*LIBORs(:,n).*Z_mat(:,n)./(1+Time.*LIBORs(:,n));
    mu(1:n) = 0;
    mu = mu.*Correl(:,:);
    mu = diag(cumsum(mu));

    for i=n+1:m
        B(i,i,n) = Time .* mu(i) -Z_mat(i,n)*Time +...
        Choles(i,:)*Z(:,n+1)*sqrt(Time) +...
        Z_mat(i,n)*1*Time*Time* LIBORs(i,n) / (1 + Time * LIBORs(i,n));
        B(i,i,n) = B(i,i,n) * LIBORs(i,n+1);
    end

end
end
```

Fig. A.7: The `MatrixBBuilder` function for the construction of the translation $B(n)$ matrices (Kienitz and Wetterau (2012)).

```
Vega0 = zeros(m,m);
For_Vega = zeros(m,1);
for p = 1:No_paths
    D = MatrixDBuilder(Libors(:,:,p),m,Z_mat,Tau,p,rho);
    q = size(D,1);
    N = size(D,3)+1;
    A = zeros(q,q,N);
    A(:,:,1) = Vega0;
    v = VN_matrix(:,p);
    B = MatrixBBuilder(Libors(:,:,p),Z(:,:,p),m,Z_mat,Tau,p,rho,Choles);
    for n = 1:N-1
        A(:,:,n+1) = D(:,:,n)*A(:,:,n) +B(:,:,n);
    end
    w = v.'*A(:,:,N);
    For_Vega = For_Vega+w;
end
For_Vega = For_Vega/No_paths;
For_Vega = For_Vega';
```

Fig. A.8: Implementation of the ARP forward method with the inclusion of a translation term (Kienitz and Wetterau (2012)).

```
Adj_Vega = zeros(m,1);
for p = 1: No_paths
    D = MatrixDBuilder(Libors(:,:,p),m,Z_mat,Tau,p,rho);
    B = MatrixBBuilder(Libors(:,:,p),Z(:,:,p),m,Z_mat,Tau,p,rho,Choles);
    N = size(D,3)+1;
    V = VN_matrix(:,p);
    Vbar = 0;
    for n = N-1:-1:1
        Vbar = Vbar + B(:,:,n).' * V;
        V = D(:,:,n).'*V;
    end
    Adj_Vega = Adj_Vega+Vbar';
end
Adj_Vega = Adj_Vega/No_paths;
```

Fig. A.9: Implementation of the ARP adjoint method with the inclusion of a translation term (Kienitz and Wetterau (2012)).

A.3 Code for Implementation of Euler Approximations for European Swaptions Delta in the LMM

```
function[Swaprate,Payoff]=CalcSwapRatesPayoff(m,paths,r,tau,phi,nom,K,Libors)

Swaprate = zeros(m,paths);
Payoff = zeros(m,paths);
for i = 1:paths
    for n = r:m
        sum = 0;
        prod = 1;
        for j = n:m
            prod = prod./(1+tau(n).*Libors(j,n,i));
            sum = sum + tau(n).*prod;
        end
        Swaprate(n,i) = (1-prod)./sum;
        Payoff(n,i) = phi.*nom.*sum.*max(Swaprate(n,i)-K,0);
    end
end
end
```

Fig. A.10: The `CalcSwapRatesPayoff` function for the calculation of swap rate and payoff values (Kienitz and Wetterau (2012)).

```
value = Payoff(r,:);
Price = 0;
for idx = 1:No_paths
    disc = 1;
    for n = r:-1:1
        disc = disc*(1+tau(1)*Libors(n,n,idx));
    end
    value(idx) = value(idx)/disc;
    Price = Price + value(idx);
end
Price_Final = Price/No_paths;
```

Fig. A.11: Algorithm for price calculation (Kienitz and Wetterau (2012)).

```
function [V] = BuildStartVectorsV(m,No_paths,r,tau,phi,nom,K,Libors,optimal)
check = optimal;
V = zeros(m,m,No_paths);
for path = 1:No_paths
    if(check(path))
        L = diag(Libors(:,:,path));
        disc = 1 + tau.*L;
        disc = 1./cumprod(disc);
        %prefactor values
        prefact = phi.*nom.*tau.*disc;
        %diagonal entries
        x = prefact.*(1- (tau.*(L-K))./(1+tau.*L));
        V(r:m,r:m,path) = diag(x(r:m));
        %subdiagonal entries
        a = prefact.*(K-L);
        b = tau./(1+tau.*L);
        Z = tril(a*b.',-1);
        V(r:m,1:m,path) = V(r:m,1:m,path) + Z(r:m,1:m);
    end
end
end
```

Fig. A.12: The `BuildStartVectorsV` function for the generation of the start vectors V (Kienitz and Wetterau (2012)).

```
v = sum(V);
For_Delta = zeros(m,1);
for p = 1:No_paths
    D = MatrixDBuilder(Libors(:,:,p),m,Z_mat,Tau,p,rho);
    m = size(D,1);
    N = size(D,3)+1;
    A = zeros(m,m,N);
    Delta0 = eye(m);
    A(:,:,1) = Delta0;
    if m == 1
        V = v;
        A(:,:,2) = D(:,:)*Delta0;
    else
        V = v(:,:,p).';
        for n = 1:N-1
            A(:,:,n+1) = D(:,:,n)*A(:,:,n);
        end
    end
    w = V'*A(:,:,N);
    For_Delta = For_Delta+w;
end
For_Delta = For_Delta/No_paths;
For_Delta = For_Delta';
```

Fig. A.13: Implementation of the ARP forward method for European swaptions (Kienitz and Wetterau (2012)).

```
v = sum(V);
Adj_Delta = zeros(m,1);
for p = 1:No_paths
    D = MatrixDBuilder(Libors(:,:,p),m,Z_mat,Tau,p,rho);
    N = size(D,3)+1;
    if m == 1
        V = v;
    else
        V = v(:,:,p).';
    end
    for n = N-1:-1:1
        V = D(:,:,n).'*V;
    end
    Adj_Delta = Adj_Delta+V;
end
Adj_Delta = Adj_Delta/No_paths;
```

Fig. A.14: Implementation of the ARP adjoint method for European swaptions (Kienitz and Wetterau (2012)).

```
v = sum(V);
Fin_Diff = zeros(m,1);
for i = 1:m
    L_Init_L=L_Init;
    L_Init_L(i)=L_Init_L(i)-eps;
    Libors_L=fullsim(L_Init_L,m,No_paths,Z_mat,tau,Z_Norm,rho,Choles);
    [SwaprateL,PayoffL] = CalcSwapRatesPayoff(m,No_paths,r,tau,phi,nom,K,Libors_L);
     valueL = PayoffL(r,:);
    PriceL = 0;
    for idx = 1:No_paths
        disc = 1;
        for n = r:-1:1
            disc = disc*(1+tau(1)*Libors_L(n,n,idx));
        end
        valueL(idx) = valueL(idx)/disc;
        PriceL = PriceL + valueL(idx);
    end
    Price_Final_L(i) = PriceL/No_paths;

    L_Init_R=L_Init;
    L_Init_R(i)=L_Init_R(i)+eps;
    Libors_R=fullsim(L_Init_R,m,No_paths,Z_mat,tau,Z_Norm,rho,Choles);
    [SwaprateR,PayoffR] = CalcSwapRatesPayoff(m,No_paths,r,tau,phi,nom,K,Libors_R);
    valueR = PayoffR(r,:);
    PriceR = 0;
    for idx = 1:No_paths
        disc = 1;
        for n = r:-1:1
            disc = disc*(1+tau(1)*Libors_R(n,n,idx));
        end
        valueR(idx) = valueR(idx)/disc;
        PriceR = PriceR + valueR(idx);
    end
    Price_Final_R(i) = PriceR/No_paths;

    Fin_Diff(i) = (Price_Final_R(i)-Price_Final_L(i))/(2*eps);
end
Fin_Diff=Fin_Diff';
```

Fig. A.15: Implementation of a finite difference scheme for European swaptions (Kienitz and Wetterau (2012)).

Appendix B

Average Runtimes for Calculations of Various Methods

B.1 Average Runtimes for LMM Caplets: Delta

Tab. B.1: Average Runtimes for Delta of Various Methods

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Delta	Price	Price + Delta	Price	Price + Delta	Price
6	0.096	0.066	0.086	0.064	0.836	0.065
7	0.118	0.080	0.103	0.079	1.206	0.080
8	0.135	0.094	0.119	0.094	1.595	0.095
9	0.157	0.111	0.140	0.111	2.089	0.111
10	0.181	0.127	0.160	0.127	2.662	0.128
11	0.210	0.145	0.183	0.146	3.317	0.147
12	0.227	0.162	0.205	0.163	4.043	0.162
13	0.259	0.183	0.226	0.180	4.850	0.182
14	0.286	0.200	0.251	0.198	5.741	0.200
15	0.324	0.220	0.279	0.219	6.791	0.218
16	0.341	0.238	0.304	0.235	7.823	0.237
17	0.383	0.259	0.331	0.260	9.009	0.258
18	0.418	0.282	0.360	0.280	10.310	0.281
19	0.478	0.305	0.388	0.302	11.771	0.302
20	0.509	0.331	0.420	0.321	13.167	0.331
21	0.559	0.367	0.449	0.348	14.892	0.349
22	0.602	0.381	0.493	0.369	16.626	0.376
23	0.679	0.408	0.534	0.395	18.524	0.393
24	0.701	0.425	0.553	0.420	20.469	0.417

Continued on next page

Tab. B.1: Continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Delta	Price	Price + Delta	Price	Price + Delta	Price
25	0.745	0.450	0.591	0.447	22.627	0.449
26	0.797	0.478	0.626	0.471	24.957	0.472
27	0.860	0.504	0.672	0.508	27.503	0.497
28	0.901	0.523	0.706	0.532	29.956	0.523
29	0.984	0.556	0.764	0.557	32.728	0.556
30	1.038	0.585	0.818	0.587	35.400	0.580
31	1.125	0.625	0.860	0.631	38.673	0.612
32	1.167	0.644	0.888	0.651	41.617	0.644
33	1.267	0.678	0.945	0.677	45.886	0.677
34	1.451	0.712	0.994	0.716	49.144	0.709
35	1.540	0.745	1.055	0.754	52.669	0.750
36	1.683	0.779	1.087	0.776	56.142	0.784
37	1.728	0.813	1.161	0.815	61.379	0.840
38	1.827	0.852	1.217	0.855	65.069	0.901
39	1.906	0.884	1.280	0.884	69.463	0.930
40	1.973	0.916	1.336	0.912	73.940	0.948

B.2 Average Runtimes for LMM Caplets: Vega

Tab. B.2: Average Runtimes for Vega of Various Methods

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Vega	Price	Price + Vega	Price	Price + Vega	Price
6	0.331	0.086	0.289	0.083	1.073	0.086
7	0.400	0.101	0.349	0.101	1.538	0.101
8	0.480	0.122	0.424	0.121	2.062	0.122
9	0.548	0.142	0.482	0.141	2.723	0.142
10	0.643	0.165	0.560	0.162	3.440	0.164
11	0.737	0.187	0.637	0.186	4.382	0.188
12	0.823	0.208	0.718	0.203	5.126	0.207
13	0.937	0.235	0.794	0.233	6.270	0.235
14	1.040	0.259	0.869	0.258	7.460	0.259

Continued on next page

Tab. B.2: Continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Vega	Price	Price + Vega	Price	Price + Vega	Price
15	1.167	0.285	0.964	0.282	8.860	0.284
16	1.253	0.308	1.062	0.307	10.173	0.308
17	1.393	0.338	1.183	0.335	11.896	0.337
18	1.512	0.363	1.273	0.363	13.646	0.364
19	1.744	0.403	1.360	0.394	15.579	0.399
20	1.854	0.427	1.474	0.422	17.367	0.426
21	2.007	0.454	1.588	0.455	19.646	0.456
22	2.193	0.485	1.704	0.483	21.940	0.487
23	2.356	0.526	1.831	0.524	24.435	0.526
24	2.513	0.550	1.955	0.549	27.672	0.550
25	2.725	0.591	2.090	0.583	31.061	0.589
26	2.957	0.621	2.215	0.621	35.121	0.621
27	3.150	0.667	2.346	0.660	38.286	0.665
28	3.295	0.694	2.471	0.697	42.952	0.697
29	3.529	0.735	2.639	0.745	45.964	0.744
30	3.780	0.772	2.806	0.777	48.412	0.777
31	4.029	0.816	2.970	0.817	52.061	0.817
32	4.217	0.847	3.186	0.854	56.686	0.853
33	4.548	0.894	3.383	0.905	62.968	0.900
34	4.821	0.934	3.595	0.949	69.180	0.944
35	5.281	0.968	3.803	0.994	76.482	0.986
36	5.709	1.027	4.006	1.027	83.450	1.027
37	6.145	1.080	4.239	1.103	90.409	1.096
38	6.482	1.123	4.471	1.146	96.166	1.139
39	6.903	1.184	4.672	1.189	103.021	1.188
40	7.130	1.224	4.872	1.245	107.504	1.238

B.3 Average Runtimes for LMM European Swaptions: Delta

Tab. B.3: Average Runtimes for Delta of Various Methods

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Delta	Price	Price + Delta	Price	Price + Delta	Price
6	0.098	0.065	0.086	0.069	0.838	0.065
7	0.121	0.079	0.104	0.085	1.161	0.080
8	0.141	0.093	0.122	0.101	1.578	0.095
9	0.158	0.111	0.143	0.119	2.067	0.113
10	0.187	0.126	0.165	0.136	2.619	0.129
11	0.210	0.145	0.188	0.155	3.263	0.144
12	0.227	0.162	0.207	0.173	3.953	0.162
13	0.261	0.185	0.230	0.190	4.805	0.182
14	0.286	0.198	0.253	0.215	5.671	0.201
15	0.323	0.222	0.282	0.236	6.718	0.221
16	0.351	0.239	0.306	0.254	7.761	0.237
17	0.390	0.262	0.333	0.285	9.041	0.264
18	0.434	0.280	0.360	0.304	10.302	0.285
19	0.480	0.309	0.389	0.327	11.726	0.307
20	0.514	0.327	0.422	0.353	13.286	0.325
21	0.557	0.355	0.453	0.386	15.021	0.356
22	0.600	0.378	0.496	0.409	16.806	0.378
23	0.654	0.416	0.521	0.429	18.613	0.405
24	0.689	0.445	0.563	0.457	20.636	0.427
25	0.775	0.459	0.607	0.488	22.852	0.460
26	0.816	0.483	0.635	0.522	25.175	0.524
27	0.889	0.515	0.677	0.576	27.750	0.523
28	0.909	0.544	0.717	0.604	30.208	0.555
29	0.989	0.573	0.760	0.645	33.166	0.585
30	1.051	0.599	0.805	0.677	36.030	0.588

Continued on next page

Tab. B.3: Continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Delta	Price	Price + Delta	Price	Price + Delta	Price
31	1.154	0.631	0.856	0.700	39.220	0.618
32	1.185	0.658	0.903	0.732	42.154	0.648
33	1.286	0.702	0.961	0.771	46.259	0.687
34	1.458	0.732	1.014	0.819	50.011	0.719
35	1.555	0.773	1.082	0.862	53.806	0.754
36	1.648	0.807	1.118	0.889	57.502	0.796
37	1.811	0.865	1.181	0.929	62.173	0.828
38	1.829	0.887	1.238	0.971	66.321	0.854
39	1.967	0.915	1.325	1.018	70.900	0.894
40	2.024	0.945	1.349	1.048	75.399	0.931

B.4 Average Runtimes for LMM European Swaptions: Vega

Tab. B.4: Average Runtimes for Vega of Various Methods

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Vega	Price	Price + Vega	Price	Price + Vega	Price
6	0.323	0.086	0.320	0.087	1.237	0.086
7	0.396	0.104	0.403	0.108	1.643	0.107
8	0.485	0.126	0.464	0.125	2.102	0.125
9	0.544	0.144	0.537	0.144	2.873	0.144
10	0.626	0.161	0.616	0.162	3.817	0.162
11	0.721	0.184	0.699	0.184	5.211	0.184
12	0.806	0.210	0.788	0.211	6.282	0.211
13	0.901	0.234	0.883	0.236	7.498	0.236
14	1.005	0.253	0.979	0.255	9.115	0.255
15	1.118	0.284	1.097	0.290	10.649	0.289
16	1.216	0.312	1.201	0.312	12.249	0.312
17	1.368	0.348	1.315	0.343	14.059	0.344
18	1.475	0.369	1.431	0.375	16.429	0.371

Continued on next page

Tab. B.4: Continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price + Vega	Price	Price + Vega	Price	Price + Vega	Price
19	1.618	0.396	1.564	0.405	18.708	0.405
20	1.758	0.423	1.689	0.436	21.390	0.428
21	1.901	0.455	1.813	0.460	23.196	0.462
22	2.043	0.489	1.946	0.489	26.290	0.489
23	2.227	0.524	2.094	0.526	28.262	0.526
24	2.373	0.555	2.239	0.558	30.779	0.559
25	2.549	0.595	2.400	0.594	34.597	0.596
26	2.728	0.617	2.557	0.633	38.616	0.628
27	2.939	0.670	2.781	0.671	40.469	0.672
28	3.098	0.707	2.910	0.711	43.912	0.713
29	3.341	0.732	3.192	0.748	48.174	0.743
30	3.562	0.834	3.293	0.783	54.356	0.778
31	3.836	0.856	3.541	0.843	58.485	0.835
32	4.174	0.909	3.621	0.877	66.338	0.870
33	4.434	0.947	3.804	0.919	67.847	0.915
34	4.824	0.994	4.064	0.957	73.953	0.954
35	5.165	1.008	4.273	1.008	77.588	1.003
36	5.589	1.051	4.487	1.056	79.648	1.055
37	5.807	1.095	4.730	1.100	86.050	1.102
38	6.196	1.138	4.950	1.140	92.552	1.140
39	6.547	1.190	5.253	1.195	98.429	1.196
40	7.015	1.243	5.422	1.248	104.467	1.248