Adjoint Venture: Fast Greeks with Adjoint Algorithmic Differentiation

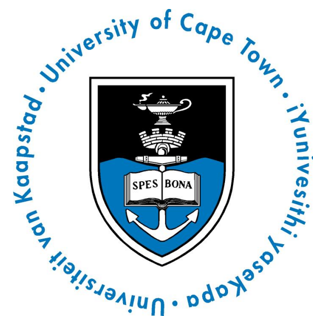
Chris McPetrie

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.

University of Cape Town

November 20, 2016

MPhil in Mathematical Finance,
University of Cape Town.



The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

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 in the University of the Cape Town. It has not been submitted before for any degree or examination in any other University.

Signed by candidate

Christopher Lindsay McPetrie
Signature removed

November 20,2016

Abstract

This dissertation explores the adjoint approach to solving affine recursion problems (ARPs) in the context of computing sensitivities of financial instruments. It is shown how, by moving from an intuitive 'forward' approach to solving a recursion to an 'adjoint' approach, one might dramatically increase the computational efficiency of algorithms employed to compute sensitivities via the pathwise derivatives approach in a Monte Carlo setting. Examples are illustrated within the context of the Libor Market Model. Furthermore, these ideas are extended to the paradigm of Adjoint Algorithmic Differentiation, and it is illustrated how the use of sophisticated techniques within this space can further improve the ease of use and efficiency of sensitivity calculations.

Acknowledgements

I would firstly like to thank my supervisor, Professor Tom McWalter, for his assistance and guidance during the compilation of this dissertation. Additionally, Dr Jörg Kienitz, despite not officially being a supervisor to my work, was very forthcoming with assistance, and I thank him.

I would also like to thank my parents, John and Sue McPetrie, for their encouragement and financial support during all my years of study. In this regard I am most indebted to my grandmother, Grace van Breda, whose financial contributions allowed me to pursue the studies of my choice, and this Masters in particular.

Thanks goes to all the academic staff involved in the administration of the MPhil in Mathematical Finance programme, for creating a fun and friendly environment in which to learn.

Lastly, a huge thanks goes to Pascal Despard, for her endless patience and support.

On a more sombre note, I would like to acknowledge my brother, Allan John McPetrie, who tragically passed away while this dissertation was being completed. Though he had little interest in Financial Mathematics, he was always very proud and supportive of my studies. He will be missed greatly.

Contents

1. Introduction 1

2. Literature Review 4

3. Theory 6
- 3.1 Diffusion Processes 6
  - 3.1.1 The Greeks 6
- 3.2 The Pathwise Derivatives Method 7
- 3.3 Approximating Diffusions Using Euler Schemes 8
- 3.4 The Affine Recursion Problem 9
  - 3.4.1 Adjoint Method 11

4. ARP Methods Applied to the LIBOR Market Model 13
- 4.1 The LIBOR Market Model 13
- 4.2 Approximating Delta Using an ARP 14
- 4.3 Pathwise Delta within the LMM 16
  - 4.3.1 The Adjoint Method 17
  - 4.3.2 An Example: Caplets within the LMM 18
- 4.4 Estimating Vega Using an ARP 22
- 4.5 Pathwise Vega within the LMM 24
  - 4.5.1 Numerical Results 25
- 4.6 Approximating Gamma Using an ARP 28

5. Algorithmic Differentiation 33
- 5.1 A Simple Example of Algorithmic Differentiation 33
  - 5.1.1 Forward Mode 35
  - 5.1.2 Adjoint Mode 39

6. Implementing AD in MATLAB 42
- 6.1 Implementing the Forward Mode Using Operator Overloading . . . 42
- 6.2 Implementing Adjoint Mode AD Using ADMAT 2.0 45

7. Conclusion 47

Bibliography 49

A. Code to Implement Forward and Adjoint ARP Approaches 51
A.1 Code to Implement Euler Approximations for Delta in the Libor Market Model 51
A.2 Code to Implement Euler Approximations for Vega in the Libor Market Model 55

B. Average Runtimes 57
B.1 Average Runtimes - Delta 57
B.2 Average Runtimes - Vega 58

C. Code to Implement AD in Matlab 60
C.1 The `Solder` Object Class 60
C.2 MATLAB Code Implementing AD Evaluation of Basket Option, Using ADMAT 2.0 65

List of Figures

4.1 Deltas calculated via each method for the model with $N = 40$. 20
4.2 Relative computational cost of each approach to find Delta for increasing number of tenors N. 21
4.3 Relative computational cost of forward and adjoint ARP approaches to finding Delta for increasing number of tenors N. 22
4.4 Vegas calculated via each method for the model with $N = 40$. Figures above on the y-axis are $\times 10^{-3}$. 26
4.5 Relative computational cost of each approach to finding Vega for increasing number of tenors N. 27
4.6 Relative computational cost of forward and adjoint ARP approaches to finding Vega for increasing number of tenors N. 28

5.1 Computational graph for function (5.1), corresponding to the instructions in (5.4). 36
5.2 Computational graph for the forward mode of differentiation of function (5.1). 38
5.3 Computational graph for the adjoint mode of differentiation of function (5.1). 41

6.1 Computational efficiency of the forward and adjoint (reverse) modes of AD in comparison to the finite-difference approach, for the European Basket Option. 46

A.1 The `LMM_Sim` function, used to simulate Libor Market Model realisations (Kienitz and Wetterau, 2012). 51
A.2 The `MatrixDBuilder` function, used to build the $D(n)$ matrices (Kienitz and Wetterau, 2012). 52
A.3 An algorithm to construct $V(N)$. 52
A.4 Implementing the forward method (Kienitz and Wetterau, 2012). . . . 53
A.5 Implementing the adjoint method (Kienitz and Wetterau, 2012). . . . 53
A.6 Implementing a finite difference scheme (Kienitz and Wetterau, 2012). 54
A.7 The `MatrixBBuilder` function, used to build the $B(n)$ matrices (Kienitz and Wetterau, 2012). 55
A.8 Implementing the forward method where the ARP includes a translation term (Kienitz and Wetterau, 2012). 56
A.9 Implementing the adjoint method where the ARP includes a translation term (Kienitz and Wetterau, 2012). 56

C.1 Full code defining the `solder` object class (for the functions required in the examples provided). 60
C.2 Full code defining the `solder` object class (continued). 61
C.3 Full code defining the `solder` object class (continued). 62
C.4 MATLAB code implementing forward mode AD for the European call option using the `solder` object class. 63
C.5 The BasketOptionADMAT function, which can be evaluated using ADMAT 2.0. 64
C.6 Implementing AD to find the Greeks of the basket option, using ADMAT 2.0. 65

List of Tables

6.1 Comparison of price and Greeks calculated with forward-mode AD and respective analytical solutions 44

B.1 Average Runtimes of the Various Approaches; Delta. 57
B.2 Average Runtimes of the Various Approaches; Vega. 58

Chapter 1

Introduction

Monte Carlo simulation has proven to be the most computationally feasible pricing technique for a large proportion of the pricing models employed by practitioners, many of which are too complex to be evaluated via deterministic numerical or analytical methods. This has given Monte Carlo methodology a lofty position in the suite of tools available to practitioners in the pricing and hedging of complex derivative securities (Capriotti, 2011).

Monte Carlo methods are still, generally speaking, computationally expensive, and these efficiency issues become even more problematic when calculating the Greeks: the sensitivities of the price of a contingent claim to underlying model parameters. These sensitivities are essential to hedging and risk management and it is vital to practitioners that they can be computed efficiently and accurately.

The standard method of price sensitivity calculation, known as the finite difference scheme or 'bumping', involves adjusting the underlying model parameters one-by-one, repeating the simulation for each perturbed parameter, and calculating finite-difference approximations. The computational overhead of this methodology thus increases linearly with the number of sensitivities computed. This leads to severe computational expense when the number of computed sensitivities is large. For example, in the case of an interest rate derivative, this method would involve adjusting each initial forward rate and repeating the Monte Carlo simulation. If the derivative in question is long-dated, this may become computationally impractical. In addition, sensitivity estimates produced via finite difference methods often have poor bias and variance properties (Giles and Glasserman, 2005).

By using information pertaining to the underlying model dynamics used in a Monte Carlo simulation, practitioners may derive better estimates of price sensitivities than by bumping. A number of alternative methods which incorporate such information have been proposed: for example, the likelihood ratio method (Boyle *et al.*, 1997) and the pathwise derivative method (Broadie and Glasserman, 1996). The likelihood ratio method differentiates the transition density of the underlying

assets or state variables, while the latter technique differentiates the evolution of the underlying assets or state variables along each path. These techniques require additional model analysis and programming in comparison to finite difference estimates, but this burden is usually justified by an improvement in the quality of Greeks calculated (Giles and Glasserman, 2005).

In a notable paper, Giles and Glasserman presented the adjoint method to accelerate the calculation of Greeks by Monte Carlo simulation, in the context of the LIBOR Market Model. This approach uses ideas which have previously been employed in fields such as computational fluid dynamics to increase the computational efficiency of pathwise estimates of the Greeks. The estimates produced by this method are identical to those generated via ordinary pathwise methods: the appeal lies then in improved computational efficiency.

Kienitz and Nowaczyk (2011) expand on these ideas by exploring the adjoint method in the broader context of the 'affine recursion problem' (ARP) and illustrate how an ARP can be resolved using both forward and adjoint methods. These approaches are analogous to those discussed by Giles and Glasserman, and Kienitz and Nowaczyk show how they can be implemented to estimate sensitivities, using Euler schemes. The more general case explored allows the extension of the method to other models, although the LIBOR Market Model is similarly used as an illustrative example.

One drawback of the pathwise derivatives method is the need to differentiate the payoff function, as well as the evolution equation — the equation which propogates the approximated diffusion from one step to another. These derivatives may prove challenging to evaluate analytically, or again, computationally inefficient to evaluate numerically. Furthermore, the payoff functions of financial derivatives often fail to be twice differentiable, and a smoothed approximation must be used in such cases. Capriotti (2011) illustrates how algorithmic differentiation (AD) can be used to overcome the issue of efficient calculation of the derivatives of the payoff function. In fact, AD enables the automatic generation of efficient code to implement the derivatives of the payout function. He shows that in the case where the number of observations of the underlying assets is larger than the number of securities simultaneously evaluated in the payoff function, or when we are concerned with the aggregated risk of a portfolio, implementation of the pathwise derivatives method using the adjoint mode of AD — adjoint algorithmic differentiation (AAD) — can improve the computational efficiency of the calculations by several orders of magnitude in comparison to the other methods discussed.

AAD has been shown to be useful in a number of extended applications, such as in the calibration of interest rate models (Henrard, 2013) and in credit valuation

adjustment (CVA) sensitivity estimations (Savickas *et al.*, 2014).

As should be apparent, there are two similar paradigms within this space — namely adjoint algorithmic differentiation and the adjoint approach to evaluating affine recursion problems (as it pertains to the calculation of sensitivities for financial instruments). These approaches overlap to some extent and the distinction between the two can be difficult to make. At a high level, adjoint algorithmic differentiation involves exploiting various structures in computer code to efficiently evaluate derivatives — it relies on specific software and embedded computer functionality — while the adjoint approach to evaluating ARPs deals more with the manipulation of matrices so as to supplant matrix-matrix multiplication with matrix-vector products and in that way enable more efficient calculation of sensitivities for financial instruments. AAD is more of a 'technological' manipulation while the adjoint approach to evaluating ARPs is more of a mathematical manipulation; the former falls under the study of computer science while the latter, as it applies in the context of financial instruments, sits neatly within the mathematical finance discipline. However, as mentioned, these topics do overlap and as such they are worth discussing together — ideally, however, avoiding further confusion regarding their distinction from one another.

This dissertation begins by providing a brief overview of the relevant literature. Thereafter some of the theory underpinning the principal concepts applicable to this topic is discussed, namely the general context of diffusion processes, the Greeks, the pathwise derivatives method, and the use of Euler schemes to approximate diffusions, before the concept of the Affine Recursion Problem is introduced and the forward and adjoint methods are discussed. Thereafter, it is illustrated how these concepts can be employed to evaluate sensitivities of financial instruments within the context of the Libor Market Model, and it is shown how both approaches provide large computational savings over the finite difference approximation. The concept of algorithmic differentiation is then introduced and illustrated via a simple example. An attempt to implement AD in MATLAB is then discussed. Finally, it is illustrated how one might use an AD package to implement algorithmic differentiation in MATLAB.

Chapter 2

Literature Review

The paper of Giles and Glasserman (2005) is regarded as seminal in the field of algorithmic differentiation as applied to computational finance, in that it was the first to introduce adjoint and algorithmic differentiation methodology to an audience of computational finance academics. These techniques had however been applied for decades in various other fields, namely computational fluid dynamics, meteorology, atmospheric sciences, and engineering design optimization. Homescu (2011) provides a comprehensive list of references illustrating earlier applications of the methodology in other fields, in addition to a useful discussion of its development within the computational finance literature. Griewank and Walther (2008) is considered the standard reference for the general theory and application of algorithmic differentiation, outside of a computational finance context.

Giles and Glasserman explore how the adjoint method can be applied to calculate sensitivities within the context of the Libor Market Model using the pathwise derivatives method. The instruments considered in that paper are caplets and a portfolio of swaptions. They illustrate that the adjoint approach offers substantial computational savings over the central difference approach to sensitivity calculation: specifically, computational costs were reduced by a factor of 5 for the computation of Deltas only, and by a factor of 25 for the computation of both Deltas and Vegas.

Leclerc *et al.* (2009) extended the method to calculate sensitivities of Bermuda-style financial derivatives. This paper further documents the computational efficiency of adjoint approaches.

Capriotti and Giles (2010) considered the application of adjoint methodology and algorithmic differentiation to the calculation of correlation Greeks. This paper considers the pricing of an instrument via Monte Carlo in a Gaussian copula framework. A Cholesky factorization of the correlation matrix is used to efficiently sample jointly normal random variables, and correlation risk is calculated by using the adjoint of the Cholesky factorization to implement the pathwise differentiation

approach. Again, vast computational savings are documented.

Capriotti (2011) further illustrates how algorithmic differentiation can be implemented in conjunction with the pathwise derivatives method to offer substantial computational savings over finite difference approaches. This paper applies the methodology to two options, namely a Basket Option and a "Best Of" Asian option. This paper also provides a simple, graphical explanation of some of the core ideas underlying algorithmic differentiation.

Capriotti *et al.* (2011) explores how adjoint algorithmic differentiation can be implemented to optimise the calculation of counterparty credit risk — specifically, the calculation of an appropriate credit valuation adjustment (CVA). This paper documents computational savings in the calculation of CVA by more than two orders of magnitude.

Other areas where adjoint methods have been shown to reduce the computational cost of sensitivity calculations include: the co-terminal swap-rate market model, as studied by Joshi and Yang (2011); the displaced-diffusion LIBOR market model, as covered by Joshi and Pitt (2010); the cross-currency displaced-diffusion LIBOR market model, reviewed by Beveridge *et al.* (2010); the framework of Markov-functional models using adjoint partial differential equation methods, as discussed in Denson and Joshi (2010); and the Heston model, as detailed in Chan *et al.* (2010). Homescu (2011) provides a more detailed review of these contributions.

Adjoint methods and the application of algorithmic differentation within the context of computational finance of areas of deep and ongoing interest for researchers. There is a wealth of literature within this field, and the influence of this knowledge continues to grow as greater numbers of practitioners begin to adopt adjoint and algorithmic techniques in favour of traditional methods.

Chapter 3

Theory

This chapter serves to briefly cover the theoretical framework upon which the methods to be discussed are built, and to establish the notation to be used throughout the dissertation.

3.1 Diffusion Processes

The following terminology and notation are largely consistent with that used in Kienitz and Nowaczyk (2011). Consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ and let

$$\bar{X} : \Omega \times \mathbb{R}^+ \to \mathbb{R}^m$$

be a stochastic process, adapted to the filtration $\mathcal{F}_t$. Let $\bar{X}$ be a diffusion: it satisfies the stochastic differential equation (SDE):

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

Here $a \in \mathcal{C}^2(\mathbb{R}^m \times \mathbb{R}^+, \mathbb{R}^m)$ is the drift, $b \in \mathcal{C}^2(\mathbb{R}^m \times \mathbb{R}^+, \mathbb{R}^{m\times d})$ is the diffusion matrix, and W is a d-dimensional Brownian motion. It may also be the case that $a = a(x, \sigma)$ and $b = b(x, \sigma)$ are functions of a set of parameters $\sigma \in \mathbb{R}^q$.

Additionally, assume $T > 0$ and consider a function $g \in \mathcal{C}^2(\mathbb{R}^m, \mathbb{R})$ such that

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

3.1.1 The Greeks

Considering g above, its derivatives can be abbreviated by:

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

Consider the function

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

where g could be, for example, the discounted payoff of a contingent claim at time T, the underlying of which is $\bar{X}$. In this way $\tilde{p}$ may be considered the fair price of the claim.

The function $\tilde{p}$ can be considered a function of a number of variables, namely $\bar{X}_i(1)$, $i = 1, \dots m$, and σ. Consider the sensitivities

$$\begin{aligned}
\tilde{\Delta} :=& \nabla_{\bar{X}(1)}(\tilde{p}) \in \mathbb{R}^m, \\
\tilde{\Gamma} :=& \text{Hess}_{\bar{X}(1)}(\tilde{p}) \in \mathbb{R}^{m \times m}, \\
\tilde{\mathcal{V}} :=& \nabla_{\sigma}(\tilde{p}) \in \mathbb{R}^m .
\end{aligned} \tag{3.2}$$

These are known as Delta, Gamma and Vega respectively, and collectively they form a subset of the group of sensitivities known as the Greeks. Sensitivities with respect to other variables or of a higher order can be considered, but focus is restricted to those given here.

3.2 The Pathwise Derivatives Method

Algorithmic differentiation is useful in a Monte Carlo setting in that it further enables the evaluation of sensitivities via the pathwise derivatives method. A review of this method, similarly given by Giles and Glasserman (2005) and Capriotti (2011), is thus helpful. The following review draws from those two papers.

Consider a diffusion process, with risk-neutral dynamics as per (3.1). A derivative security maturing at time T_N with discounted payoff $g(\bar{X}(T_N))$ has price $V = \mathbb{E}_{\mathbb{Q}}[g(\bar{X}(T_N))]$ — the expected value of the discounted payoff under the risk-neutral measure $\mathbb{Q}$. Alternatively, the discounted payoff g may depend on earlier values in the evolution of $\bar{X}$, in which case the price would be given by $V = \mathbb{E}_{\mathbb{Q}}[g(\bar{X}(T_1), \dots, \bar{X}(T_N))]$, with $T_1, \dots, T_N$ the relevant reference dates.

These expectations can be calculated using Monte Carlo by simulating a number N_{MC} of random samples of the state vector $X = (X(T_1), \dots, X(T_M))^\top$, giving $X[1], \dots, X[N_{MC}]$ (where $X[i]$ refers to the ith sample), and computing the associated payout $g(X)$ for each simulation. Note that the notation changes from $\bar{X}$ to X to indicate that X is an approximation to $\bar{X}$. The value of the derivative security can then be estimated as:

$$V \approx \frac{1}{N_{MC}} \sum_{i_{MC}=1}^{N_{MC}} g(X[i_{MC}]) \, . \tag{3.3}$$

Now consider the problem of calculating the sensitivity of the option price V to a set of N_θ parameters $\theta = (\theta_1, \dots \theta_{N_\theta})$, with $\frac{\partial}{\partial \theta_k}\mathbb{E}_{\mathbb{Q}}[g(X)]$ the sensitivity of V with respect to the kth parameter θ_k. The pathwise method estimates this sensitivity using

$$\frac{\partial}{\partial \theta_k} g(X[i_{MC}]),$$

the sensitivity of the discounted payoff along the ith path of the Monte Carlo simulation. This is an unbiased estimate if

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

that is, if the derivative and expectation can be interchanged. Conditions which permit this interchange are discussed in Glasserman (2003) — most pertinently, that the discounted payoff function g must be Lipschitz continuous. We now ascribe to g the subscript $_\theta$ in order to denote its dependence on the parameter set θ. If the relevant conditions are satisfied, we may then write:

$$\bar{\theta}_k = \frac{\partial g_\theta(X)}{\partial \theta_k} = \sum_{j=1}^{m} \frac{\partial g_\theta(X)}{\partial X_j}\frac{\partial X_j}{\partial \theta_k} + \frac{\partial g_\theta(X)}{\partial \theta_k}\,. \tag{3.4}$$

This is the pathwise derivatives estimator, and by computing and averaging this value over each Monte Carlo path we may estimate the desired sensitivity of V to θ_k. Capriotti (2011) notes that g_θ may depend on θ not only implicitly through the state vector X, but also explicitly. For this reason the second term in (3.4) must be considered when implementing the pathwise derivatives method — although it is often overlooked in the academic literature. In the case where the state vector $X = (X(T_1), \dots, X(T_N))$ is a path of a m-dimensional diffusion process, the pathwise derivatives estimator (3.4) may instead be written as:

$$\bar{\theta}_k = \sum_{l=1}^{N}\sum_{j=1}^{m} \frac{\partial g_\theta(X(T_1), \dots, X(T_N))}{\partial X_j(T_l)}\frac{\partial X_j(T_l)}{\partial \theta_k} + \frac{\partial g_\theta(X)}{\partial \theta_k}\,. \tag{3.5}$$

3.3 Approximating Diffusions Using Euler Schemes

Consider a discrete time grid

$$0 = T_1 < \dots < T_N = T$$

and define $\delta_n := T_{n+1} - T_n$, $n = 1, \dots, N-1$. Consider the sequence

$$X(n+1) := F_n(X(n), \sigma), \quad X(1) := \bar{X}_{T_1} = \bar{X}_0, \quad \forall\, 1 \le n \le N-1\,. \tag{3.6}$$

Equation (3.6) is known as the evolution equation. The function F_n is a $\mathcal{C}^2$-map such that:

$$\begin{aligned} F_n &: \mathbb{R}^m \times \mathbb{R}^q \to \mathbb{R}^m \\ (x, \sigma) &\mapsto F_n(x, \sigma)\,. \end{aligned} \tag{3.7}$$

F_n is usually selected so that $X(n) \approx \bar{X}(T_n)$. This is typically achieved using an Euler scheme:

$$F_n(y, \sigma) := y + a(y, \sigma)\delta_n + b(y, \sigma)Z(n+1)\sqrt{\delta_n}\,. \tag{3.8}$$

Here $Z(n+1) \in \mathbb{R}^d$ are a sequence of random vectors sampled from the Standard Normal distribution.

3.4 The Affine Recursion Problem

Kienitz and Nowaczyk provide a detailed description of the affine recursion problem and its relevance to adjoint methods and the calculation of sensitivities for financial instruments. This largely guides the following section.

Let $A(n) \in \mathbb{R}^{m\times q}$ be a sequence of matrices satisfying the forward recursion

$$A(n+1) = D(n)A(n) + C(n), \quad \forall 1 \le n \le N-1, \quad A(1) = A_1 \tag{3.9}$$

where $N, m, q \in \mathbb{N}$, and for any $n = 1, \ldots, N-1$

$$A_1 \in \mathbb{R}^{m\times q}, \qquad D(n) \in \mathbb{R}^{m\times m}, \qquad C(n) \in \mathbb{R}^{m\times q}, \qquad v \in \mathbb{R}^{1\times m}\,. \tag{3.10}$$

The affine recursion problem (ARP) is then

$$w := vA(N) \in \mathbb{R}^{1\times q}\,. \tag{3.11}$$

Here A is the recursing matrix, A_1 is the initial matrix, D are the factors, C are the translations, v is the start vector and w is the result vector.

Kienitz and Nowaczyk show that any ARP as given in (3.9) is uniquely solvable, and in fact

$$A(n) = \left(\sum_{j=1}^{n-1} \left(\prod_{k=1}^{j-1} D(n-k) \right) C(n-j) \right) + \left(\prod_{j=1}^{n-1} D(n-j) \right) A_1, \quad \forall 1 \le n \le N\,. \tag{3.12}$$

As a result,

$$w = vA(N) = \left(\sum_{j=1}^{N-1} v \left(\prod_{k=1}^{j-1} D(n-k) \right) C(n-j) \right) + v \left(\prod_{j=1}^{N-1} D(N-j) \right) A_1\,. \tag{3.13}$$

The proof of this result is derived using induction over n. Consider the case $n = 1$:

$$A(1) = \underbrace{\left(\sum_{j=1}^{0}\left(\prod_{k=1}^{j-1} D(1-k)\right) C(1-j)\right)}_{\text{sums to 0}} + \underbrace{\left(\prod_{j=1}^{0} D(n-j)\right)}_{=1} A_1 = A_1$$

Now, by the standard algorithm of mathematical induction: assume that (3.12) holds true for n, and show that it is true for $(n+1)$:

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

Solving the above requires nothing more than algebraic manipulation. Additionally, the solution employs the fact that a summation running from $j = 2$ to n over $(j-1)$ is equivalent to one running from $j = 1$ to n over j. Ultimately, the above proves, by mathematical induction, that (3.12) is true for all $1 \leq n \leq N$.

In order to calculate w, an intuitive algorithm would be to simply implement the forward recursion as per (3.9). The computational cost of calculating $A(n+1)$ from $A(n)$ comprises the cost of the matrix multiplication $D(n)A(n)$ and the matrix addition $D(n)A(n) + C(n)$. A naive implementation of matrix multiplication has complexity $\mathcal{O}(m^3)$ and the matrix addition has complexity $\mathcal{O}(m^2)$, which is negligible in comparison. Since the calculation of $A(N)$ requires $N-1$ of these recursions and a final matrix-vector multiplication $A(N)v$ — also of comparatively negligible complexity $\mathcal{O}(m^2)$ — the total computational cost to calculate w is $\mathcal{O}(Nm^3)$. This method is known as the forward method (Kienitz and Nowaczyk).

3.4.1 Adjoint Method

Kienitz and Nowaczyk provide an alternative approach which reduces the high computational cost of the forward method.

Consider the vectors $V(n) \in \mathbb{R}^{m\times 1}$, defined by

$$V(n) := D(n)^{\top} V(n+1), \quad \forall 1 \le n \le N-1, \quad V(N) := v^{\top}. \tag{3.14}$$

This is the sequence *adjoint* to the ARP in (3.9). The vectors $\bar{V}(n) \in \mathbb{R}^{q\times 1}$, defined by

$$\bar{V}(n) := C(n)^{\top} V(n+1) + \bar{V}(n+1), \quad \forall 1 \le n \le N-1, \quad \bar{V}(N) := 0, \tag{3.15}$$

are known as the total adjoint sequence of the ARP in (3.9). These sequences enable the development of the adjoint, or reverse, method. The ARP in (3.9), as an alternative to the forward approach in (3.12), may instead be solved as

$$w = vA(N) = \sum_{n=1}^{N-1} V(n+1)^{\top} C(n) + V(1)^{\top} A_1 = \bar{V}(1)^{\top} + V(1)^{\top} A_1 . \tag{3.16}$$

The sequences $V(n)$ and $\bar{V}(n)$ are explicitly given by

$$\forall 1 \le n \le N-1 : V(n) = \left(\prod_{k=n}^{N-1} D(k)^{\top} \right) v^{\top}, \tag{3.17}$$

$$\forall 1 \le n \le N-1 : \bar{V}(n) = \sum_{j=n}^{N-1} C(j)^{\top} V(j+1) . \tag{3.18}$$

Justification for these results is provided by Kienitz and Nowaczyk. The claim in (3.17) follows directly from the definition in (3.14):

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

Similarly, (3.18) follows directly from (3.15):

$$\begin{aligned}
\bar{V}(N-1) =& C(N-1)^\top V(N) + 0 \\
\bar{V}(N-2) =& C(N-2)^\top V(N-1) + C(N-1)^\top V(N) \\
\bar{V}(N-3) =& C(N-3)^\top V(N-2) + C(N-2)^\top V(N-1) + C(N-1)^\top V(N) \\
& \cdots \\
\bar{V}(n) =& \sum_{j=n}^{N-1} C(j)^\top V(j+1)
\end{aligned}$$

Proving the claim made in (3.16) requires the calculation of $w^\top$ rather than w, and involves nothing more than some thoughtful re-indexation and algebraic manipulation:

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

The first line follows from (3.13), while the third- and second-last lines follow from (3.17) and (3.18) respectively.

The considerable numerical advantage of the adjoint method lies in the fact that while (3.9) involves matrix recursion, (3.14) and (3.15) are vector recursions only. As a result, the computational cost of calculating the sequences of matrices $V(n)$ and $\bar(V)(n)$ is of complexity $\mathcal{O}(Nm^2)$, as is the complexity of the entire algorithm. This offers a substantial computational savings over the forward method, which has complexity $\mathcal{O}(Nm^3)$ (Kienitz and Nowaczyk).

Chapter 4

ARP Methods Applied to the LIBOR Market Model

4.1 The LIBOR Market Model

Giles and Glasserman (2005) introduced the application of the adjoint method in the context of the LIBOR Market Model. The discussion which follows is largely guided by this paper. This section serves as a useful example of the application of the ideas surrounding ARPs to the calculation of sensitivities of financial instruments.

Consider the LIBOR Market Model as specified by Brace *et al.* (1997). For a fixed set of $m+1$ bond maturities T_i, $i = 1, \ldots, m+1$, with spacings $T_{i+1} - T_i = \delta_i$, let $\tilde{L}_i(t)$ denote the forward LIBOR rate fixed at time t for the interval $[T_i, T_{i+1}), \ldots i = 1, \ldots, m$. Let $\mathcal{I}(t)$ represent the index of the next maturity date as of time t, $T_{\mathcal{I}(t)-1} \leq t < T_{\mathcal{I}(t)}$. The arbitrage-free dynamics of the forward rates are

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

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

$$\mu_i(\tilde{L}(t)) = \sum_{j=\mathcal{I}(t)}^{i} \frac{\sigma_i^\top \sigma_j \delta_j \tilde{L}_j(t)}{1 + \delta_j \tilde{L}_j(t)}.$$

An Euler scheme is applied to the logarithms of the forward rates in order to simulate the process. For the evolution of the Euler scheme, a fixed time step h is used. A useful choice of h is one such that $Nh = m$. Note that although h is used in the Euler scheme, the terms δ_i persist in the model, as these are determined by the maturities of the bonds used to construct the initial forward curve and are thus model inputs rather than flexible modelling parameters. In the most simple exercise, one

in which bond prices are generated artificially and whose maturities can thus be chosen at the discretion of the modeller, it may be convenient to make δ_i constant and tie up directly with h. This yields

$$L_i(n+1) = L_i(n)\exp\left(\left[\mu_i(L(n)) - \frac{||\sigma_i||^2}{2}\right]h + \sigma_i^\top Z(n+1)\sqrt{h}\right), \quad i = \mathcal{I}(nh), \ldots, m. \tag{4.1}$$

A rate remains fixed once it settles at its maturity, and thus $L_i(n+1) = L_i(n)$ if $i < \mathcal{I}(nh)$. One can minimize the computational cost by first evaluating the summations

$$S_i(n) = \sum_{j=\mathcal{I}(t)}^{i} \frac{\sigma_j \delta_j L_j(n)}{1+\delta_j L_j(n)}, \quad i = \mathcal{I}(nh), \ldots, m. \tag{4.2}$$

This then gives $\mu_i = \sigma_i^\top S_i$.

4.2 Approximating Delta Using an ARP

The following is guided by Kienitz and Nowaczyk. Recall the Greeks discussed in Section 3.1.1, and Delta in particular. Let X be an approximation of $\bar{X}$, computed via an Euler scheme: then, $p := \mathbb{E}[g(X(N)]$ is an approximation of $\tilde{p}$. Furthermore

$$\Delta := \nabla_{X(1)}(p) \in \mathbb{R}^m$$

is the approximated Delta. It can be calculated as the result vector

$$\Delta = v\Delta(N) \tag{4.3}$$

of an ARP with matrix recursion of the form

$$\Delta(n+1) = D(n)\Delta(n), \quad \forall\, 1 \le n \le N-1, \quad \Delta(1) = I\,. \tag{4.4}$$

Here, the factor matrix $D(n)$ is given as:

$$D_{ik}(n) := \frac{\partial F_n^i}{\partial y_k}(X(n), \sigma), \quad \forall\, 1 \le i, k \le m,\ \forall 1 \le n \le N-1, \tag{4.5}$$

the recursing matrix $\Delta(n)$ is defined by:

$$\Delta_{ij} := \frac{\partial X_i(n)}{\partial X_j(1)}, \quad \forall\, 1 \le i, j \le m, \forall\, 1 \le n \le N, \tag{4.6}$$

and the start vector is defined by

$$v := \nabla g(X(N)) \in \mathbb{R}^{1\times m}\,. \tag{4.7}$$

Recalling the general form of an ARP from (3.9), this equates to:

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

The above pertains to the forward mode of solving an ARP. The solution may equally be obtained via the adjoint mode. If $V(n)$ is the vector sequence adjoint to the recursion in (4.4):

$$V(n) := D(n)^\top V(n+1), \quad \forall\, 1 \le n \le N-1, \quad V(N) := \nabla g(X(N))^\top \tag{4.8}$$

as per Section 3.4.1, then Δ can alternatively be calculated by

$$\Delta = V(1)^\top\,. \tag{4.9}$$

Proving these claims requires nothing more than an application of the chain rule of differentiation, as discussed in Section 3.2, in the context of ARPs as laid out in Section 3.4. Recalling the evolution equation in (3.6):

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

we apply the the chain rule:

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

This verifies the claim made in (4.4). To verify the claim made in (4.9), consider (3.16), and recall that for the ARP in question here, $C(n)$ (the translations) are all equal to 0:

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

In order to explicitly calculate the entries $D_{ik}(n)$ of the factor matrices $D(n)$, one is required to differentiate the evolution equation. This can be performed on the general form of the Euler scheme (3.8) but it is perhaps more useful to do so on the evolution equation as prescribed by a particular model. In that vein we now apply these ideas within the particular context of the Libor Market Model.

4.3 Pathwise Delta within the LMM

We now restrict our focus to the Libor Market Model and consider the problem of computing deltas within this context.

We are required to find the form of the factor matrices $D(n)$. These denote the derivative of the transformation from $L_i(n)$ to $L_i(n+1)$ as prescribed by the evolution equation (4.1) and are given by:

$$D_{ii}(n) = \begin{cases} 1 & i < \mathcal{I}(nh); \\ \dfrac{L_i(n+1)}{L_i(n)} + \dfrac{L_i(n+1)||\sigma||^2\delta_i h}{(1+\delta_i L_i(n))^2}, & i \geq \mathcal{I}(nh); \end{cases}$$

and, for j $\neq$ i, (4.10)

$$D_{ij}(n) = \begin{cases} \dfrac{L_i(n+1)\sigma_i^\top \sigma_j \delta_j h}{(1+\delta_j L_j(n))^2}, & i > j \geq \mathcal{I}(nh); \\ 0 & \text{otherwise}. \end{cases}$$

These results follow directly from the evolution equation in (4.1). Firstly consider the diagonal elements of the matrix $D(n)$, i.e. where $j = i$, and consider the case $i < \mathcal{I}(nh)$. Since $L_i(n+1) = L_i(n)$ for $i < \mathcal{I}(nh)$, it is clear that $D_{ii}(n) = 1$ in this case. Now consider the case $i < \mathcal{I}(nh)$:

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

Now consider the case where $j < i < \mathcal{I}(nh)$. Since $L_i(n+1) = L_i(n)$ for $i < \mathcal{I}(nh)$, clearly $D_{ij}(n) = 0$ in this case. Similarly, in the case where $\mathcal{I}(nh) < i < j$, $D_{ij}(n) = 0$ again, since the summation used to construct μ_i only runs from $\mathcal{I}(t)$ to i and thus no $L_j(n)$ for $j > i$ will be included in the evolution equation.

Finally, consider the case where $i > j \geq \mathcal{I}(nh)$:

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

This confirms the result stated in (4.10). Giles and Glasserman propose an efficient implementation of this algorithm, which uses $\Delta_{ij}(n+1) = \Delta_{ij}(n)$ for $i < \mathcal{I}(nh)$, and for $i \geq \mathcal{I}(nh)$

$$
\Delta_{ij}(n+1) = \frac{L_i(n+1)}{L_i(n)} \Delta_{ij}(n) + L_i(n+1)\sigma_i^\top \sum_{k=\mathcal{I}(nh)}^{i} \frac{\sigma_k \delta_k h \Delta_{kj}(n)}{1+\delta_k L_k(n))^2}
$$

However, due to the potentially large number of forward rates m used in the Libor Market Model, the numerical evaluation of $\Delta_{ij}(n)$ via this scheme can be computationally expensive. To this end, the adjoint method to solving ARPs can be employed, offering computational savings without including any further approximations.

4.3.1 The Adjoint Method

Consider again the desired vector of deltas $\frac{\partial g(X(N))}{\partial X(1)}$

$$
\begin{aligned}
\frac{\partial g}{\partial X(1)} =& \frac{\partial g}{\partial X(N)} \Delta(N) \\
=& \frac{\partial g}{\partial X(N)} D(N-1)D(N-2)\ldots D(1)\Delta(1) \\
=& V(1)^\top \Delta(1)
\end{aligned}
$$

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

$$
V(n) = D(n)^\top V(n+1), \quad V(N) = \left(\frac{\partial g}{\partial X(N)} \right)^\top \tag{4.11}
$$

The major insight here is that, as in the general case of ARPs discussed in Section 3.4.1, the adjoint relation in (4.11) uses matrix-vector products, while (4.4) uses matrix-matrix products. As a result, one needs only to update the m entries of the adjoint variables $V(n)$, rather than the m^2 variables required in the forward method. This can lead to considerable computational savings (Giles and Glasserman).

4.3.2 An Example: Caplets within the LMM

In order to further illustrate these ideas, we specify a function g which pertains to a financial instrument typically evaluated using the Libor Market Model. Consider a caplet for the interval $[T_m, T_{m+1})$ struck at K. Such an instrument has discounted payoff

$$g = B_1(0)\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\delta_m \max\left(0,\, \tilde{L}_m(T_m)-K\right)\,. \tag{4.12}$$

Here $B_1(0) = \frac{1}{1+\delta_0 L_0(T_0)}$ is the T_0 price of a bond maturing at T_1. This is included separately in the equation simply because it is known at T_0.

Applying the ideas discussed in the preceding sections, we are well positioned to implement both forward and adjoint mode solutions to the calculation of the deltas of this instrument via an ARP. The matrices $D(n)$ can be constructed using the format discussed in (4.10) and the start vector $v = \frac{\partial g}{\partial X(N)}$ can be found using the following analysis:

We seek $\frac{\partial g}{\partial X(N)}$, where g is the discounted payoff above and

$$X(N) = [L_0(T_m),\, L_1(T_m), \ldots,\, L_j(T_m), \ldots,\, L_m(T_m)]\,.$$

Firstly, consider the case $j = m$:

$$\begin{aligned}
\frac{\partial g}{\partial L_m(T_m)} &= \frac{\partial}{\partial L_m(T_m)}\left[B_1(0)\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\delta_m \max(0, \tilde{L}_m(T_m)-K)\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\frac{\partial}{\partial L_m(T_m)}\left[\frac{1}{1+\delta_m\tilde{L}_m(T_m)}(\tilde{L}_m(T_m)-K)^+\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m-1}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right) \\
&\quad\times\left[\frac{-\delta_m}{(1+\delta_m\tilde{L}_m(T_m))^2}(\tilde{L}_m(T_m)-K)^+ + \frac{1}{1+\delta_m\tilde{(L)}_m(T_m)}\mathbb{1}(\tilde{L}_m(T_m)>K)\right] \\
&= B_1(0)\delta_m\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\left[\mathbb{1}(\tilde{L}_m(T_m)>K) - \frac{\delta_m}{(1+\delta_m\tilde{L}_m(T_m))}(\tilde{L}_m(T_m)-K)^+\right]\,.
\end{aligned} \tag{4.13}$$

Next, consider the case $j < m$:

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

$$=B_1(0)\delta_m(\tilde{L}_m(T_m)-K)^+\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\times\frac{-\delta_j}{1+\delta_j\tilde{L}_j(T_m)}\,. \tag{4.14}$$

Thereafter, (4.13) and (4.14) give the form of v:

$$v=\begin{cases}B_1(0)\delta_m\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\times & \\ \left[\mathbb{1}(\tilde{L}_m(T_m)>K)-\frac{\delta_m}{(1+\delta_m\tilde{L}_m(T_m)}(\tilde{L}_m(T_m)-K)^+\right] & j=m; \\ B_1(0)\delta_m(\tilde{L}_m(T_m)-K)^+\left(\prod_{i=1}^{m}\frac{1}{1+\delta_i\tilde{L}_i(T_m)}\right)\times\frac{-\delta_j}{1+\delta_j\tilde{L}_j(T_m)} & j<m\end{cases} \tag{4.15}$$

It must be noted that this does not agree with Glasserman and Zhao (1999), who suggest that the form (4.13) applies to all $j \leq m$. However no justification is provided for this claim.

All the components required to solve the ARP are now available in close form:

$$v \overset{(4.15)}{=} \frac{\partial g}{\partial X(N)}, \text{ with } A(n)=\Delta(n), \quad D(n)\overset{(4.10)}{=}D(n), \quad C(n)=0, \quad A(1)=\Delta(1)=I\,.$$

Consider a scheme to calculate the vector of deltas for this caplet, implemented in forward and adjoint modes, as well as via finite difference methods for comparative purposes. A great deal of the implementation which follows is adapted from Kienitz and Wetterau (2012). All the code referred to here is found in Appendix A.1.

Firstly, one must simulate the Libor Market Model realisations. This can be achieved using the `LMM_Sim` function, found in Figure A.1. Next, $V(N)$ can be constructed using the code in Figure A.3.

The forward method can be implemented using the code in Figure A.4, while the adjoint method can be implemented using the code in Figure A.5. Within each of these implementations, the $D(n)$ matrices are constructed using the `MatrixDBuilder` function, as shown in Figure A.2. Finally, one can implement a finite difference scheme for comparative computational performance using the code in Figure A.6.

The standard metric for comparing relative computational efficiency in this context is the ratio of the time taken to compute the price of an instrument and the desired derivatives, to the time taken to compute the price only:

$$\frac{\text{Cost}(P+D)}{\text{Cost}(P)}\,. \tag{4.16}$$

Here P represents the price of an instrument, D represents the desired derivatives and Cost represents the time taken to compute the respective calculations. These times can be recorded using the MATLAB `tic` and `toc` functions.

Numerical Results

We consider a flat initial forward curve with $L_i(0) = 0.07$, $\sigma_i = 0.2$, $\delta_i = 0.5$, $i = 0, \ldots m$. For simplicity we set $h = \delta_i$ and $Nh = m$. In order to test the efficacy of the proposed methods across increasing degrees of complexity we consider multiple iterations of the model with N ranging from 1 to 40. We consider a caplet with discounted payoff (4.12) struck at-the-money and compute the vector of deltas for each of these models using the forward and adjoint modes of ARP resolution, as well as via a finite difference scheme. A Monte Carlo sample of size 1000 was used and each approach was run five times for each N. An average computational efficiency could then be determined for each approach, over increasing N. Figure 4.1 shows the Deltas computed using each of these methods for the model with $N = 40$. This Delta profile is consistent with the fact that a one-factor model has been implemented.

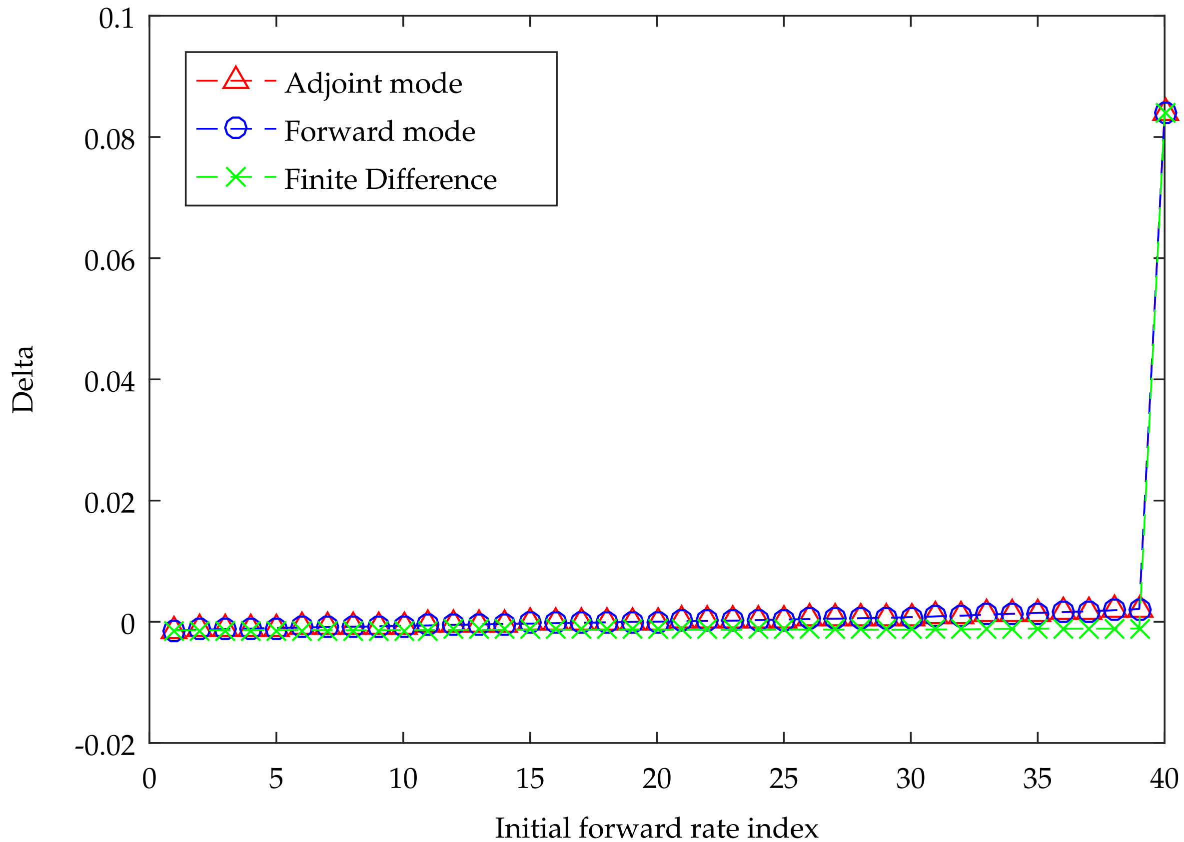


Fig. 4.1: Deltas calculated via each method for the model with $N = 40$.

The relative computational cost of each approach as proxied by (4.16) is shown

in Figure 4.2. As the graph shows, the forward and adjoint ARP approaches offer large computational savings in comparison to a finite difference scheme. The average runtimes used to calculate these relative efficiency ratios can be found in Table B.1 in Appendix B.1.

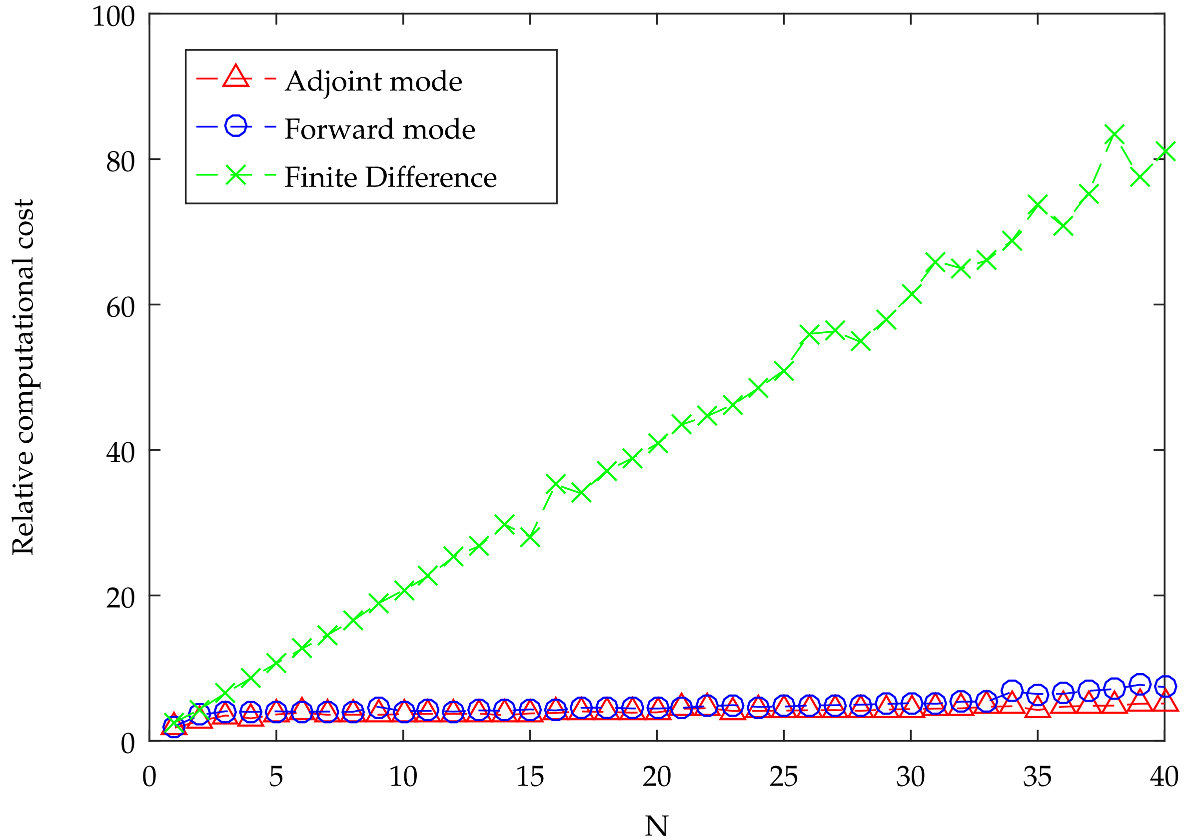


Fig. 4.2: Relative computational cost of each approach to find Delta for increasing number of tenors N.

Removing the finite difference scheme from the comparison and zooming in on the forward and adjoint ARP approaches, Figure 4.3 shows that the adjoint mode performs better across all values of N:

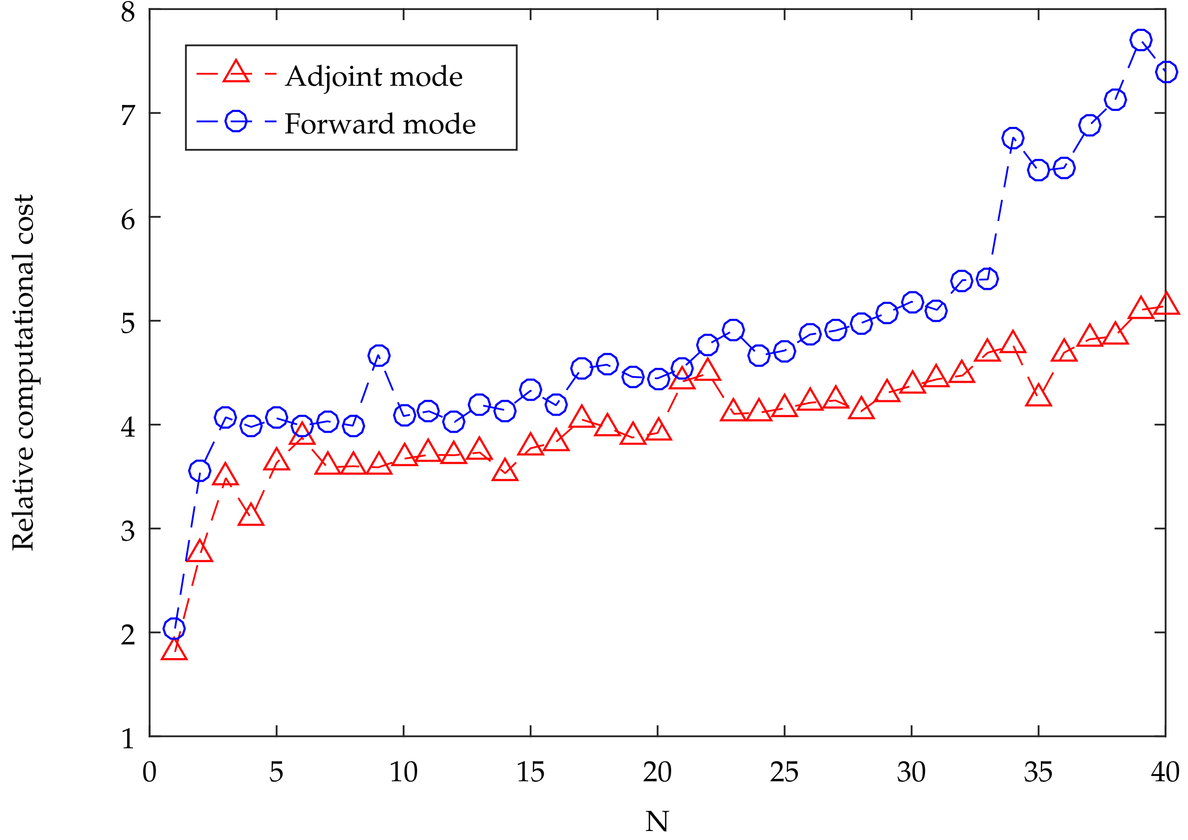


Fig. 4.3: Relative computational cost of forward and adjoint ARP approaches to finding Delta for increasing number of tenors N.

4.4 Estimating Vega Using an ARP

The following is guided by Kienitz and Nowaczyk. Recall the Greeks discussed in Section 3.1.1, and Vega in particular. Let X be an approximation of $\bar{X}$, computed via an Euler scheme: then $p := \mathbb{E}[g(X(N))]$ is an approximation of $\tilde{p}$. Furthermore,

$$\mathcal{V} := \nabla_\sigma(p) \in \mathbb{R}^{1\times q}$$

is the approximated vega. It can be calculated as the result vector

$$\mathcal{V} = v\mathcal{V}(N) \tag{4.17}$$

of an ARP with matrix recursion of the form

$$\mathcal{V}(n+1) = D(n)\mathcal{V}(n) + B(n), \quad \mathcal{V}(1) = 0\,. \tag{4.18}$$

Here, the recursing matrix $\mathcal{V}(n) \in \mathbb{R}^{m \times q}$ is defined by

$$\mathcal{V}_{ij} := \frac{\partial X_i(n)}{\partial \sigma_j} \,\forall\, 1 \leq i \leq m,\, \forall\, 1 \leq j \leq q,\, \forall\, 1 \leq n \leq N\,, \tag{4.19}$$

the translation matrix $B(n)$ is given by

$$B_{ij}(n) := \frac{\partial F_n^i}{\partial \sigma_j}(X(n), \sigma) \,\forall\, 1 \leq i \leq m,\, \forall\, 1 \leq j \leq q,\, \forall\, 1 \leq n \leq N-1\,, \tag{4.20}$$

while the factor matrix $D(n)$ and the start vector v are defined as per (4.5) and (4.7) respectively. Recalling the general form of an ARP from (3.9), this equates to:

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

The above can be used to find Vega via the forward mode. As should be familiar by now, the adjoint method can alternatively be employed. Let $V(n)$ be the sequence adjoint to the recursion and let $\bar{V}(n)$ be the total adjoint sequence. Vega can then be calculated as:

$$\mathcal{V} = v\mathcal{V}(N) = \sum_{n=1}^{N-1} V(n+1)^\top B(n) = \bar{V}(1)^\top\,. \tag{4.21}$$

The proof of (4.18) simply involves the application of the chain rule to (4.19):

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

while it is clear that $\mathcal{V}_{ij}(1) = \frac{\partial X_i(1)}{\partial \sigma_j} = 0$. Proving (4.17) involves another implementation of the chain rule:

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

while (4.21) follows directly from (3.16):

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

The form of $B(n)$ is easily determined in the case where $F(n)$ is the evolution equation as given by the general form of an Euler scheme, but is perhaps more useful to work with a particular model. As a result a return to the context of the Libor Market Model is prudent.

4.5 Pathwise Vega within the LMM

Much of what was established in Section 4.3 applies here. The key difference is that the volatility parameters affect the evolution equation: that is why the ARP for calculating Vega (4.18) includes a translation term, while the ARP for calculating Delta (4.4) does not. It is thus required to find the form of this translation term, given the evolution equation prescribed by the Libor Market Model (4.1). Recalling the definition (4.20), $B(n)$ is determined to have the form

$$B_{ij}(n) = \begin{cases} \frac{L_i(n+1)\sigma_i\delta_i L_i(n)h}{1+\delta_i L_i(n)} + \\ \left(S_i(n)h - \sigma_i h + Z(n+1)\sqrt{h}\right) L_i(n+1) & i = j \geq \mathcal{I}(nh); \\ \frac{L_i(n+1)\sigma_i\delta_j L_j(n)h}{1+\delta_j L_j(n)} & i > j \geq \mathcal{I}(nh); \\ 0, & \text{otherwise.} \end{cases} \tag{4.22}$$

The equation (4.22) follows from directly computing $\frac{\partial F_n^i}{\partial \sigma_j}(X(n),\sigma)$. Firstly consider the case $i = j \geq \mathcal{I}(nh)$:

$$\frac{\partial F_n^i}{\partial \sigma_i}(X(n),\sigma) = \frac{\partial}{\partial \sigma_i}\left[L_i(n)\exp\left([\mu_i(L(n)) - \tfrac{||\sigma_i||^2}{2}]h + \sigma_i^\top Z(n+1)\sqrt{h}\right)\right]$$

$$= \overbrace{L_i(n)\exp\left([\mu_i(L(n)) - \tfrac{||\sigma_i||^2}{2}]h + \sigma_i^\top Z(n+1)\sqrt{h}\right)}^{=L_i(n+1)}$$

$$\times \frac{\partial}{\partial \sigma_i}\left[\left(\sigma_i^\top \sum_{j=\mathcal{I}(t)}^{i} \frac{\sigma_j\delta_j L_j(n)}{1+\delta_j L_j(n)} - \frac{||\sigma_i||^2}{2}\right)h + \sigma_i^\top Z(n+1)\sqrt{h}\right]$$

$$= L_i(n+1)$$

$$\times \frac{\partial}{\partial \sigma_i}\left[\left(\sigma_i^\top \sum_{j=\mathcal{I}(t)}^{i-1} \frac{\sigma_j\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{\sigma_i^2\delta_i L_i(n)}{1+\delta_i L_i(n)} - \frac{||\sigma_i||^2}{2}\right)h + \sigma_i^\top Z(n+1)\sqrt{h}\right]$$

$$= L_i(n+1) \times \left[\left(\sum_{j=\mathcal{I}(t)}^{i-1} \frac{\sigma_j\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{2\sigma_i\delta_i L_i(n)}{1+\delta_i L_i(n)} - \sigma_i\right)h + Z(n+1)\sqrt{h}\right]$$

$$= L_i(n+1) \times \left[\left(\sum_{j=\mathcal{I}(t)}^{i} \frac{\sigma_j\delta_j L_j(n)}{1+\delta_j L_j(n)} + \frac{\sigma_i\delta_i L_i(n)}{1+\delta_i L_i(n)} - \sigma_i\right)h + Z(n+1)\sqrt{h}\right]$$

$$= \frac{L_i(n+1)\sigma_i\delta_i L_i(n)h}{1+\delta_i L_i(n)} + \left(S_i(n)h - \sigma_i h + Z(n+1)\sqrt{h}\right) L_i(n+1)\,.$$

Next consider the case $i > j \geq \mathcal{I}(nh)$:

$$\frac{\partial F_n^i}{\partial \sigma_j}(X(n),\sigma) = \frac{\partial}{\partial \sigma_j}\left[L_i(n)\exp\left([\mu_i(L(n)) - \tfrac{||\sigma_i||^2}{2}]h + \sigma_i^\top Z(n+1)\sqrt{h}\right)\right]$$

$$= Li(n+1) \times \frac{\partial}{\partial \sigma_j}\left[\left(\sigma_i^\top \sum_{j=\mathcal{I}(t)}^{i} \frac{\sigma_j\delta_j L_j(n)}{1+\delta_j L_j(n)} - \frac{||\sigma_i||^2}{2}\right)h + \sigma_i^\top Z(n+1)\sqrt{h}\right]$$

$$= \frac{L_i(n+1)\sigma_i\delta_j L_j(n)h}{1+\delta_j L_j(n)}\,.$$

For the cases $j > i$ or $j < \mathcal{I}(nh)$, σ_j would not appear in the evolution equation, and as such $B_{ij}(n)$ is 0 in these cases. This proves (4.22).

In order to implement these ideas, one can use the `MatrixDBuilder` function in Figure A.2 in Appendix A.1 to construct the $D(n)$ matrices, while the $B(n)$ matrices can be constructed using the `MatrixBBuilder` function, shown in Figure A.7 in Appendix A.2.

Furthermore, the forward and adjoint implementations must be adjusted to incorporate the translation matrix. The code for implementing the forward mode is shown in Figure A.8, while adjoint mode can be implemented using the code in Figure A.9.

4.5.1 Numerical Results

The same model specifications are used as in Section 4.3.2. Figure 4.4 shows the Vegas computed using each of these methods for the model with $N = 40$.

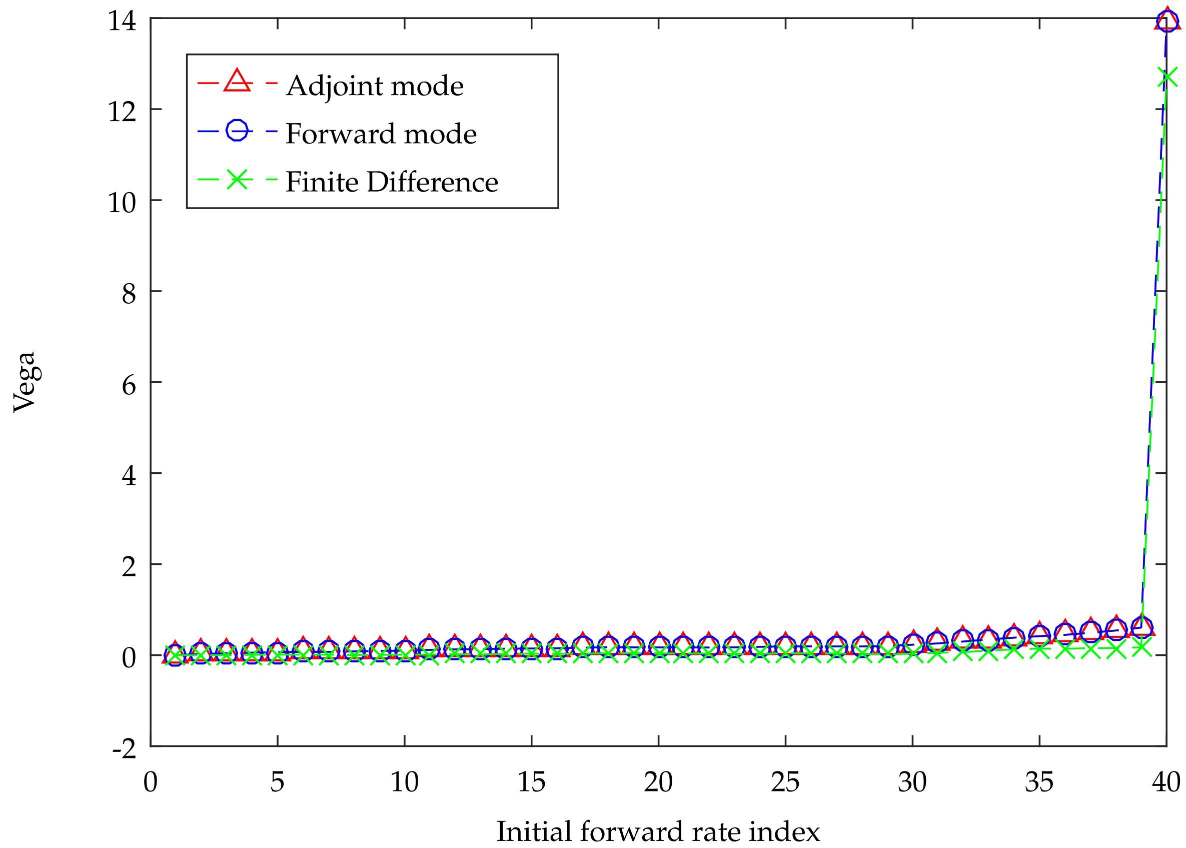


Fig. 4.4: Vegas calculated via each method for the model with $N = 40$. Figures above on the y-axis are $\times 10^{-3}$.

The relative computational cost of each approach as proxied by (4.16) is shown in Figure 4.5. As the graph shows, the forward and adjoint ARP approaches offer large computational savings in comparison to a finite difference scheme, but these are not as pronounced as in the case of calculating Delta. The average runtimes used to compute these relative efficiency ratios can be found in Table B.2, in Appendix B.2.

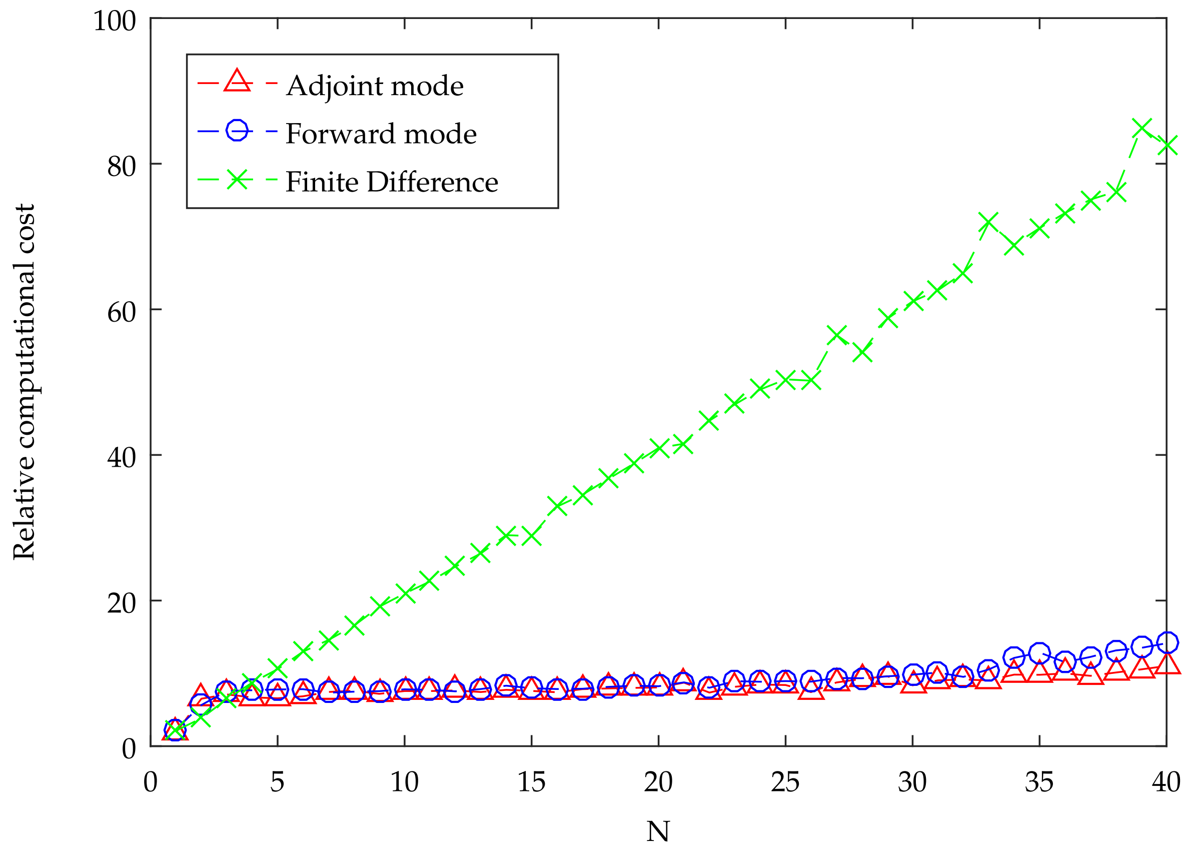


Fig. 4.5: Relative computational cost of each approach to finding Vega for increasing number of tenors N.

Removing the finite difference scheme from the comparison and zooming in on the forward and adjoint ARP approaches, Figure 4.6 shows that the adjoint mode generally performs better.

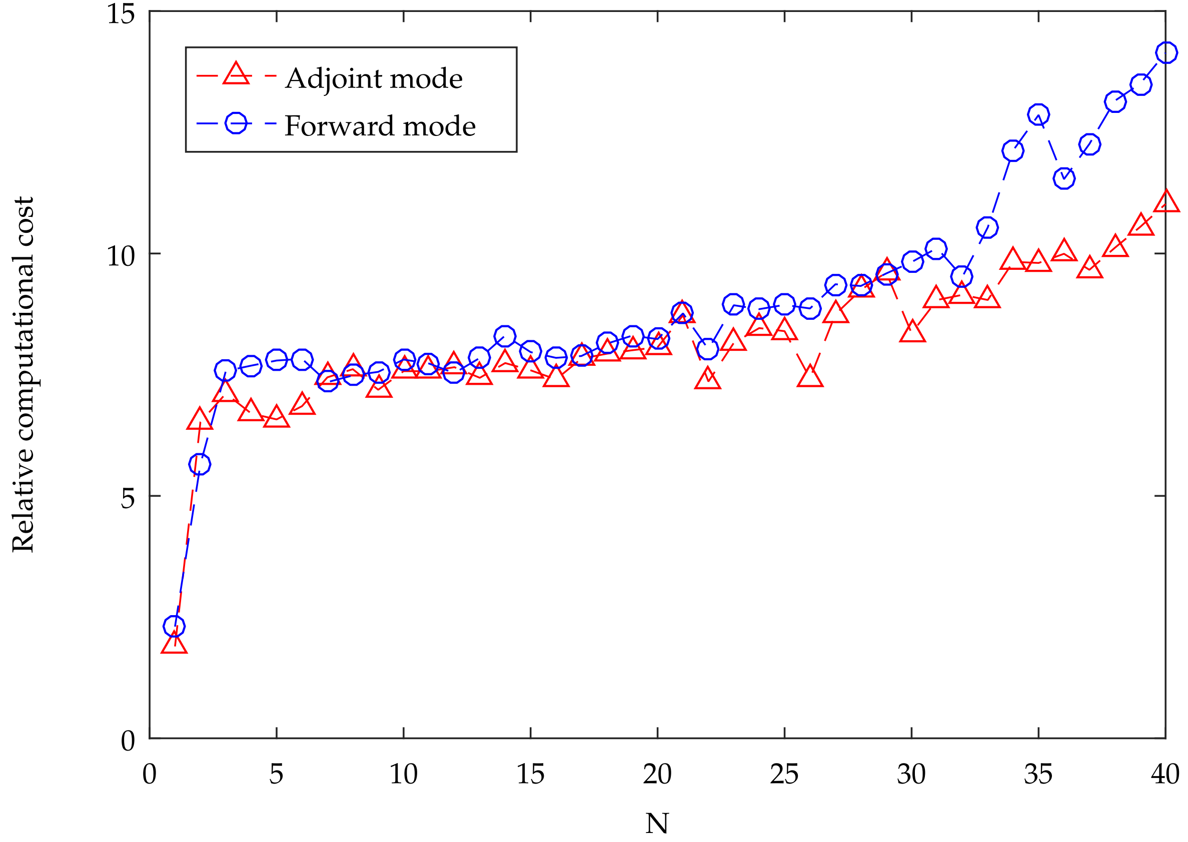


Fig. 4.6: Relative computational cost of forward and adjoint ARP approaches to finding Vega for increasing number of tenors N.

4.6 Approximating Gamma Using an ARP

The following is guided by Kienitz and Nowaczyk. Recall the Greeks discussed in Section 3.1.1, and Gamma in particular. Let X be an approximation of $\bar{X}$, computed via an Euler scheme: then, $p := \mathbb{E}[g(X(N))]$ is an approximation of $\tilde{p}$. Furthermore,

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

is the approximated Gamma. The estimation of Gamma via an ARP is somewhat more challenging than that of Delta or Vega. A number of additional structures are

required. For any $1 \leq i, j, k, \leq m$, $1 \leq n \leq N$, consider the scalar quantities

$$\Delta_{ij}(n) := \frac{\partial X_i(n)}{\partial X_j(1)}, \qquad D_{ik}(n) := \frac{\partial}{\partial X_k(n)} F_n^i(X(n), \sigma),$$

$$G_{ik}^{(j)}(n) := \frac{\partial^2 X_i(n)}{\partial X_j(1) \partial X_k(1)}, \qquad E_{jk}^{(i)}(n) := \frac{\partial}{\partial X_j(n) \partial X_k(n)} F_n^i(X(n), \sigma),$$

$$C_{ik}^{(j)}(n) := (\Delta(n)^\top E^{(i)}(n)^\top \Delta(n))_{kj}, \qquad H_{ij} := \frac{\partial}{\partial X_i(N) \partial X_j(N)} g(X(N)).$$

These are the entries of the respective matrices

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

which are all $\in \mathbb{R}^{m \times m}$.

In order to develop an expression for Γ, it must firstly be shown that for any $1 \leq j \leq m$, $G^{(j)}(n)$ satisfies the recursion

$$G^{(j)}(n+1) = D(n) G^{(j)}(n) + C^{(j)}(n), \quad \forall 1 \leq n \leq N-1, \quad G^{(j)}(1) = 0. \tag{4.23}$$

This can be shown by considering the entry $G_{ik}^{(j)}(n+1)$ and using this to infer the structure of $G^{(j)}(n+1)$, which is given by

$$\begin{aligned}
G_{ik}^{(j)}(n+1) &= \frac{\partial^2 X_i(n+1)}{\partial X_j(1) \partial X_k(1)} \\
=& \frac{\partial}{\partial X_k(1)} \left(\frac{\partial}{\partial X_j(1)} X_i(n+1) \right) \\
=& \frac{\partial}{\partial X_k(1)} (\Delta_{ij}(n+1)) \\
=& \frac{\partial}{\partial X_k(1)} ((D(n)\Delta(n))_{ij}) = \sum_{s=1}^m \frac{\partial}{\partial X_k(1)} (D_{is}(n) \Delta_{sj}(n)) \quad \text{(matrix multiplication)} \\
=& \sum_{s=1}^m \left(\frac{\partial}{\partial X_k(1)} [D_{is}(n)] \Delta_{sj}(n) + D_{is}(n) \frac{\partial}{\partial X_k(1)} [\Delta_{sj}(n)] \right) \quad \text{(by the product rule)} \\
=& \sum_{s=1}^m \frac{\partial}{\partial X_k(1)} \left[\frac{\partial}{\partial X_s(n)} F_n^i(X(n), \sigma) \right] \Delta_{sj}(n) + \sum_{s=1}^m D_{is}(n) \frac{\partial^2 X_s(n)}{\partial X_j(1) \partial X_k(1)} \\
=& \sum_{s=1}^m \frac{\partial}{\partial X_k(1)} \left[\frac{\partial}{\partial X_s(n)} F_n^i(X(n), \sigma) \right] \Delta_{sj}(n) + \sum_{s=1}^m D_{is}(n) G_{sk}^{(j)} \\
=& \sum_{s=1}^m \underbrace{\frac{\partial}{\partial X_k(1)} \left[\frac{\partial}{\partial X_s(n)} F_n^i(X(n), \sigma) \right]}_{\Psi} \Delta_{sj}(n) + (D(n) G^{(j)})_{ik}.
\end{aligned}$$

Consider now only the section marked Ψ above:

$$\begin{aligned}
&\frac{\partial}{\partial X_k(1)}\left[\frac{\partial}{\partial X_s(n)}F_n^i(X(n),\sigma)\right] \\
=&\sum_{t=1}^{m}\frac{\partial}{\partial X_t(n)}\frac{\partial X_t(n)}{\partial X_k(1)}\left(\frac{\partial F_n^i(X(n),\sigma)}{\partial X_s(n)}\right) \quad \text{(by the multivariable chain rule)} \\
=&\sum_{t=1}^{m}\frac{\partial X_t(n)}{\partial X_k(1)}\left(\frac{\partial^2 F_n^i(X(n),\sigma)}{\partial X_s(n)\partial X_t(n)}\right) \\
=&\sum_{t=1}^{m}\Delta_{tk}(n)E_{st}^{(i)}(n) \\
=&\sum_{t=1}^{m}E_{st}^{(i)}(n)\Delta_{tk}(n) \quad \text{(these are scalars)}
\end{aligned}$$

Substituting this back into the original equation gives

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

while it is clear 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 verifies (4.23). Consider now $U^{(j)}\in\mathbb{R}^{m\times 1}$, the vector sequences adjoint to the ARP (4.23), with start vector $v:=\nabla(g)(X(N))\in\mathbb{R}^{1\times m}$. Then, by the result shown in (3.16), the corresponding result vectors $w_{(j)}$ are given by

$$w_{(j)}=vG^{(j)}(N)=\sum_{n=1}^{N-1}U^{(j)}(n+1)^\top C^{(j)}(n)=\bar{U}^{(j)}(1)^\top\in\mathbb{R}^{1\times m}\,. \tag{4.24}$$

Furthermore, let $w\in\mathbb{R}^{m\times m}$ be a matrix, the rows of which are comprised by the vectors $w_{(j)}$, and let $Y:=\Delta(N)^\top H\Delta(N)\in\mathbb{R}^{m\times m}$. The desired Gamma is then given by

$$\Gamma=w+Y. \tag{4.25}$$

The equation (4.25) is verified by considering the entry Γ_{jk}, with $1 \leq j, k \leq m$, and using this to infer the structure of the Gamma matrix, as follows:

$$\begin{aligned}
\Gamma_{jk} &= \frac{\partial^2 g(X(N))}{\partial X_j(1)\partial X_k(1)} \\
&= \frac{\partial}{\partial X_j(1)}\left(\sum_{i=1}^{m}\frac{\partial g(X(N))}{\partial X_i(N)}\frac{\partial X_i(N)}{\partial X_k(1)}\right) \quad \text{(by the multivariable chain rule)} \\
&= \sum_{i=1}\underbrace{\frac{\partial}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right)}_{♀}\Delta_{ik}(N) + \sum_{i=1}^{m}\frac{\partial g(X(N))}{\partial X_i(N)}\frac{\partial^2 X_i(N)}{\partial X_j(1)\partial X_k(1)} \quad \text{(product rule)}\,.
\end{aligned}$$

Consider only the section marked ♀ above:

$$\begin{aligned}
&\frac{\partial}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right) \\
&= \sum_{l=1}^{m}\frac{\partial}{\partial X_l(N)}\frac{\partial X_l(N)}{\partial X_j(1)}\left(\frac{\partial g(X(N))}{\partial X_i(N)}\right) \quad \text{(by the multivariate chain rule)} \\
&= \sum_{l=1}^{m}\Delta_{lj}(N)\left(\frac{\partial^2 g(X(N))}{\partial X_l(N)\partial X_i(N)}\right) \\
&= \sum_{l=1}^{m}\left(\frac{\partial^2 g(X(N))}{\partial X_l(N)\partial X_i(N)}\right)\Delta_{lj}(N) \quad \text{(these are scalars)}\,.
\end{aligned}$$

Returning to the original equation:

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

This confirms (4.25).

As should be apparent, it is more challenging to implement an ARP to solve for Gamma than in the cases of Delta and Vega. Firstly, the discounted payoff function g of an option product typically fails to be twice differentiable, and as such g must be substituted with a smoothed approximation (Giles and Glasserman). Furthermore, all $\Delta(n)$ are required in the calculation of the $C^{(j)}$ and as such these must be calculated via the forward mode (Kienitz and Nowaczyk).

The preceding sections have hopefully served to elucidate the concept of the affine recursion problem, how one might solve an ARP via the forward and adjoint methods, and how these ideas can be applied to solve for sensitivities in the context of the Libor Market Model. It will now be illustrated how these ideas can be further extended into the paradigm of algorithmic differentiation.

Chapter 5

Algorithmic Differentiation

Giles and Glasserman (2005) provide a useful appendix which explains some of the fundamental aspects of algorithmic differentiation. A computer program generally takes as input a number of variables u_i, $i = 1, \ldots N_I$, which collectively form the input vector $\mathbf{u}^0$. The execution of this computer program will then generally involve a number of functions N, each acting on two values generated by the previous step in the execution. For completeness, functions with only one input can be considered binary functions with no dependence on the second parameter. If one takes $\mathbf{u}^n$ as the vector of active variables after the nth step $\mathbf{f}^n$ in the execution of the program, then it is clear that

$$\mathbf{u}^n = \mathbf{f}^n(\mathbf{u}^{n-1})$$

The computer program in entirety can then be expressed as

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

If one is interested in the sensitivity of the final output vector $\mathbf{u}^N$ to elements of the input vector $\mathbf{u}^0$ then one can solve this system using many of the ideas discussed in the preceding sections. Applying the concepts of sensitivity calculation in the context of computer programs is known as algorithmic differentiation (AD). AD can be performed in both forward and adjoint modes, much like how ARPs can be solved via both forward and adjoint approaches. What follows is an illustration of the key ideas surrounding AD.

5.1 A Simple Example of Algorithmic Differentiation

A simple example serves well to illustrate how one might implement algorithmic differentiation in both the forward and adjoint modes. This is an approach similarly undertaken by Capriotti (2011), Homescu (2011) and Neidinger (2010). Much of the notation and structure which follows is adopted from Capriotti.

Consider the function $F : \mathbb{R}^3 \to \mathbb{R}^2, (y_1, y_2)^\top = (F_1(x_1, x_2, x_3), F_2(x_1, x_2, x_3))^\top$, given by:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \cos(x_1 x_2) + \ln(x_3^2 + x_1) \\ \exp(x_3 - x_2) + \sin(x_2 x_3) \end{pmatrix}. \tag{5.1}$$

When this function is evaluated by a computer program, the output vector is calculated via a sequence of instructions. These instructions incorporate a set of scalar internal variables, $v_1, \dots v_N$, defined by:

$$v_i = x_i \qquad i = 1, \dots, n \tag{5.2}$$

$$v_i = \psi_i(\{v_j\}_{j \prec i}), \qquad i = n+1, \dots, N\,. \tag{5.3}$$

The first n internal variables store the original input variables and those following are given by a sequence of recursive operations: $\{v_j\}_{j \prec i}$ refers to the set of internal variables v_j, with $j < i$, such that v_i depends explicitly on v_j, while ψ_i represents some combination of elementary or intrinsic operations. The last m internal variables are the final output of the function: $y_{i-N+m} = v_i$, $i = N - m + 1, \dots, N$. This representation is not unique and the function can be decomposed in various ways (Capriotti). Here, the internal variables are chosen so that each ψ_i consists of only one elementary operation: this leads to a slightly more detailed exposition than is perhaps necessary, but is more representative of how a computer program would evaluate the function. For the function given in (5.1), the internal calculations can be represented as follows:

$$\left.\begin{array}{l}
v_1 = x_1, \quad v_2 = x_2, \quad v_3 = x_3, \\
\qquad\qquad \downarrow \\
v_4 = \psi_4(v_1, v_2) = v_1 \times v_2 \\
v_5 = \psi_5(v_2, v_3) = v_2 \times v_3 \\
v_6 = \psi_6(v_2, v_3) = v_3 - v_2 \\
v_7 = \psi_7(v_3) = v_3^2 \\
v_8 = \psi_8(v_5) = \sin(v_5) \\
v_9 = \psi_9(v_1, v_7) = v_1 + v_7 \\
v_{10} = \psi_{10}(v_4) = \cos(v_4) \\
v_{11} = \psi_{11}(v_9) = \ln(v_9) \\
v_{12} = \psi_{12}(v_6) = \exp(v_6) \\
\qquad\qquad \downarrow \\
y_1 = v_{13} = \psi_{13}(v_{10}, v_{11}) = v_{10} + v_{11} \\
y_2 = v_{14} = \psi_{14}(v_8, v_{12}) = v_{12} + v_8\,.
\end{array}\right\} \tag{5.4}$$

The set of instructions (5.4) can be visually represented by a computational graph, as shown in Figure 5.1. This graph consists of nodes, relating to each of

the internal variables v_i, and arcs connecting explicitly dependent variables. To each one of these arcs, say for example that connecting nodes w_i and w_j, $j < i$, it is possible to assign a corresponding arc derivative:

$$D_{i,j} = \frac{\partial \psi_i(\{v_k\}_{k \prec i})}{\partial v_j} \tag{5.5}$$

as shown in Figure 5.1. For example, $D_{5,2}$ represents the partial derivative of node w_5 with respect to node w_2. Inherent to the application of AD is the automatic manner in which these arc derivatives can be calculated by mechanically applying the rules of differentiation, node by node.

The arc derivatives are as follows:

$$\left.\begin{array}{lll}
D_{4,1} = v_2 & D_{4,2} = v_1 & D_{5,2} = v_3 \\
D_{5,3} = v_2 & D_{6,2} = -1 & D_{6,3} = 1 \\
D_{7,3} = 2v_3 & D_{8,5} = \cos(v_5) & D_{9,1} = 1 \\
D_{9,7} = 1 & D_{10,4} = -\sin(v_4) & D_{11,9} = \frac{1}{v_9} \\
D_{12,6} = \exp(v_6) & D_{13,10} = 1 & D_{13,11} = 1\,. \\
D_{14,12} = 1 & D_{14,8} = 1 &
\end{array}\right\} \tag{5.6}$$

5.1.1 Forward Mode

With the computer program implementing $F(x)$ given in terms of the set of instructions in (5.2) and (5.3), or equivalently, represented by the computational graph in 5.1, the calculation of the gradient of each of the m components of the output vector y

$$\nabla F_i(x) = \nabla y_i = (\partial_{x_1} F_i(x), \ldots, \partial_{x_n} F_i(x))^\top \tag{5.7}$$

involves no more than applying the chain rule of differentiation (Capriotti). This series of gradients can alternatively be represented by the Jacobian matrix:

$$J(x) = \begin{pmatrix}
\frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\
\frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\
\vdots & & & \vdots \\
\frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n}
\end{pmatrix} \tag{5.8}$$

with each row pertaining to one of the gradients in (5.7).

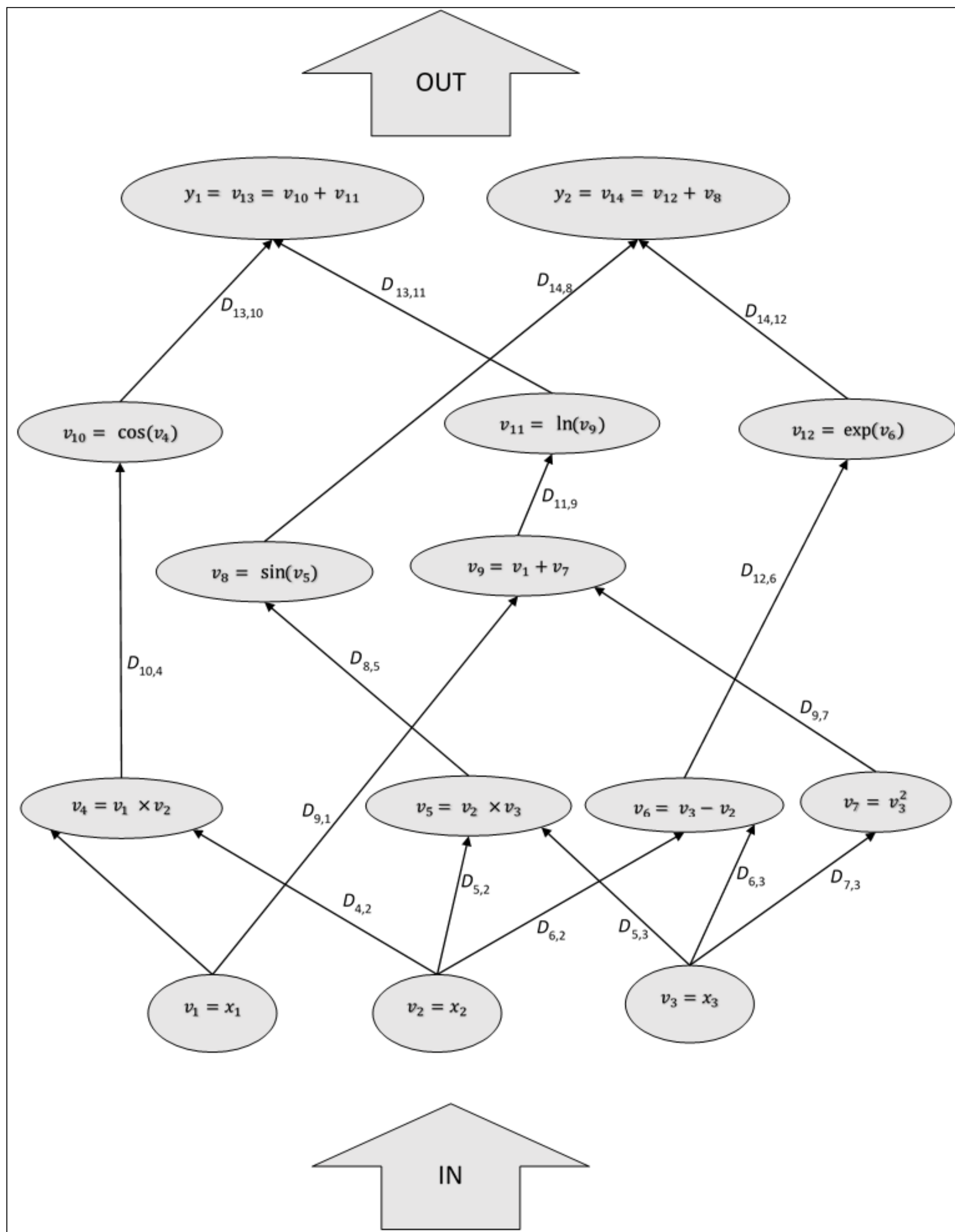


Fig. 5.1: Computational graph for function (5.1), corresponding to the instructions in (5.4).

The forward (or tangent) mode of AD is implemented in the following manner:

$$\begin{aligned} \dot{v}_i &= e_i, & i &= 1, \ldots, n \\ \dot{v}_i &= \sum_{j \prec i} D_{i,j} \dot{v}_j, & i &= n+1, \ldots, N \,. \end{aligned} \tag{5.9}$$

Here $\dot{v}_i$ is the 'tangent' of node i, and $e_i, \ldots e_n$ refer to vectors of the canonical basis in $\mathbb{R}^n$. After the preceding tangents have been propagated, we find $\dot{v}_i$, $i = N - m + 1, \ldots, N$:

$$\dot{v}_i = \dot{y}_{i-N+m} = \sum_{j=1}^{n} e_j \frac{\partial v_i}{\partial x_j} = \sum_{j=1}^{n} e_j \frac{\partial y_{i-N+m}}{\partial x_j} \tag{5.10}$$

which indicates that the final tangents are constructed as a linear combination of the columns of the Jacobian matrix. For the function given in (5.1), this leads to the following set of calculations:

$$\begin{aligned}
&\dot{v}_1 = (1,0,0)^\top, \quad \dot{v}_2 = (0,1,0)^\top, \quad \dot{v}_3 = (0,0,1)^\top \\
&\qquad\qquad\downarrow \\
&\dot{v}_4 = D_{4,1}\dot{v}_1 + D_{4,2}\dot{v}_2 \\
&\dot{v}_5 = D_{5,2}\dot{v}_2 + D_{5,3}\dot{v}_3 \\
&\dot{v}_6 = D_{6,2}\dot{v}_2 + D_{6,3}\dot{v}_3 \\
&\dot{v}_7 = D_{7,3}\dot{v}_3 \\
&\dot{v}_8 = D_{8,5}\dot{v}_5 \\
&\dot{v}_9 = D_{9,1}\dot{v}_1 + D_{9,7}\dot{v}_7 \\
&\dot{v}_{10} = D_{10,4}\dot{v}_4 \\
&\dot{v}_{11} = D_{11,9}\dot{v}_9 \\
&\dot{v}_{12} = D_{12,6}\dot{v}_6 \\
&\qquad\qquad\downarrow \\
&\dot{y}_1 = \dot{v}_{13} = D_{13,10}\dot{v}_{10} + D_{13,11}\dot{v}_{11} \\
&\dot{y}_2 = \dot{v}_{14} = D_{14,8}\dot{v}_8 + D_{14,12}\dot{v}_{12} \,.
\end{aligned}$$

Using the arc derivatives given in (5.6), which would in fact be evaluated simultaneously with the corresponding node, the final gradient vectors are given by:

$$\begin{aligned} \dot{y}_1 =& \nabla y_1 = (D_{13,10}D_{10,4}D_{4,1} + D_{13,11}D_{11,9}D_{9,1},\; D_{13,10}D_{10,4}D_{4,2},\; D_{13,11}D_{11,9}D_{9,7}D_{7,3})^\top \\ =& \left(-x_2 \sin(x_1x_2) + \frac{1}{x_1 + x_3^2},\; -x_1 \sin(x_1x_2),\; \frac{2x_3}{x_1 + x_3^2}\right)^\top \end{aligned}$$

and

$$\begin{aligned} \dot{y}_2 =& \nabla y_2 = (0,\; D_{14,8}D_{8,5}D_{5,2} + D_{14,12}D_{12,6}D_{6,2},\; D_{14,8}D_{8,5}D_{5,3} + D_{14,12}D_{12,6}D_{6,3})^\top \\ =& (0,\; x_3\cos(x_2x_3) - \exp(x_3 - x_2),\; x_2\cos(x_2x_3) + \exp(x_3 - x_2))^\top \,. \end{aligned}$$

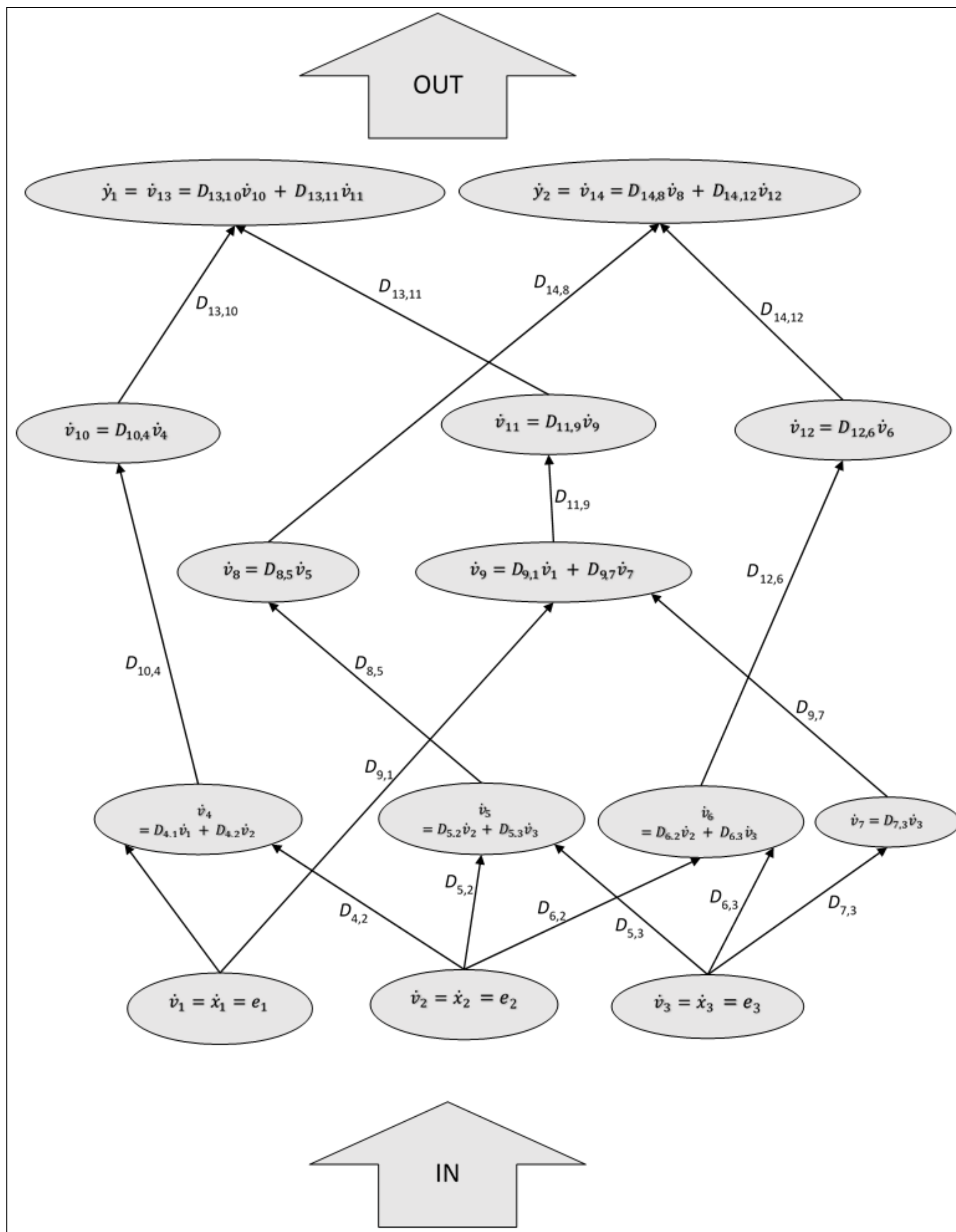


Fig. 5.2: Computational graph for the forward mode of differentiation of function (5.1).

5.1.2 Adjoint Mode

The implementation of the forward mode required one to start with the tangents of the independent variables, and then propagate forward through the computational scheme until the gradients of the output variables were attained. In contrast, the adjoint mode requires one to start with the output variables, and propagate backwards through the instructions, forming the 'adjoints' along the way. Finally, the derivatives with respect to the independent variables are formed as the last set of adjoints. The following section is strongly guided by Capriotti.

The adjoint of any intermediate variable v_i is defined as:

$$\bar{v}_i = \sum_{j=1}^{m} \lambda_j \frac{\partial y_j}{\partial v_i} \tag{5.11}$$

with λ_j a vector in $\mathbb{R}^m$. For the output variables, $\bar{y}_i = \lambda_i$, $i = 1, \ldots, m$, while for the intermediate variables:

$$\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{j \succ i} \frac{\partial y}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \succ i} D_{j,i} \bar{v}_j \tag{5.12}$$

where the indices $j \succ i$ are such that v_j depends explicitly on v_i. After the preceding adjoints have been propagated, we find the adjoints of the input variables, $\bar{v}_i$, $i = 1, \ldots, n$:

$$\bar{v}_i = \bar{x}_i = \sum_{j=1}^{M} \lambda_j \frac{\partial y_j}{\partial v_i} = \sum_{j=1}^{M} \lambda_j \frac{\partial y_j}{\partial x_i} \tag{5.13}$$

which indicates that the final adjoints are formed as a linear combination of the rows of the Jacobian matrix. For the function given in (5.1), this leads to the following set of calculations:

$$\bar{y}_1 = \bar{v}_{13} = (1, 0)^\top, \quad \bar{y}_2 = \bar{v}_{14} = (0, 1)^\top,$$

$$\downarrow$$

$$\begin{aligned}
\bar{v}_{12} &= D_{14,12}\bar{v}_{14} \\
\bar{v}_{11} &= D_{13,11}\bar{v}_{13} \\
\bar{v}_{10} &= D_{13,10}\bar{v}_{13} \\
\bar{v}_{9} &= D_{11,9}\bar{v}_{11} \\
\bar{v}_{8} &= D_{14,8}\bar{v}_{14} \\
\bar{v}_{7} &= D_{9,7}\bar{v}_{9} \\
\bar{v}_{6} &= D_{12,6}\bar{v}_{12} \\
\bar{v}_{5} &= D_{8,5}\bar{v}_{8}
\end{aligned}$$

$$\begin{aligned}
\bar{v}_4 &= D_{10,4}\bar{v}_{10} \\
\bar{v}_3 &= D_{5,3}\bar{v}_5 + D_{6,3}\bar{v}_6 + D_{7,3}\bar{v}_7 \\
\bar{v}_2 &= D_{4,2}\bar{v}_4 + D_{5,2}\bar{v}_5 + D_{6,2}\bar{v}_6 \\
\bar{v}_1 &= D_{4,1}\bar{v}_4 + D_{9,1}\bar{v}_9.
\end{aligned}$$

Using the arc derivatives in (5.6), which would be calculated by a forward sweep through the program before the adjoints are propagated in reverse, the adjoints of the input variables are finally given by:

$$\begin{aligned}
\bar{v}_1 = \bar{x}_1 &= (D_{4,1}D_{10,4}D_{13,10} + D_{9,1}D_{11,9}D_{13,11},\ 0) \\
&= \left(-x_2\sin(x_1x_2) + \frac{1}{x_1 + x_3^2},\ 0\right), \\
\bar{v}_2 = \bar{x}_2 &= (D_{4,2}D_{10,4}D_{13,10},\ D_{5,2}D_{8,5}D_{14,8} + D_{6,2}D_{12,6}D_{14,12}) \\
&= (-x_1\sin(x_1x_2),\ x_3\cos(x_2x_3) - \exp(x_3 - x_2))
\end{aligned}$$

and

$$\begin{aligned}
\bar{v}_3 = \bar{x}_3 &= (D_{7,3}D_{9,7}D_{11,9}D_{13,11},\ D_{5,3}D_{8,5}D_{14,8} + D_{6,3}D_{12,6}D_{14,12}) \\
&= \left(\frac{2x_3}{x_1 + x_3^2},\ x_2\cos(x_2x_3) + \exp(x_3 - x_2)\right).
\end{aligned}$$

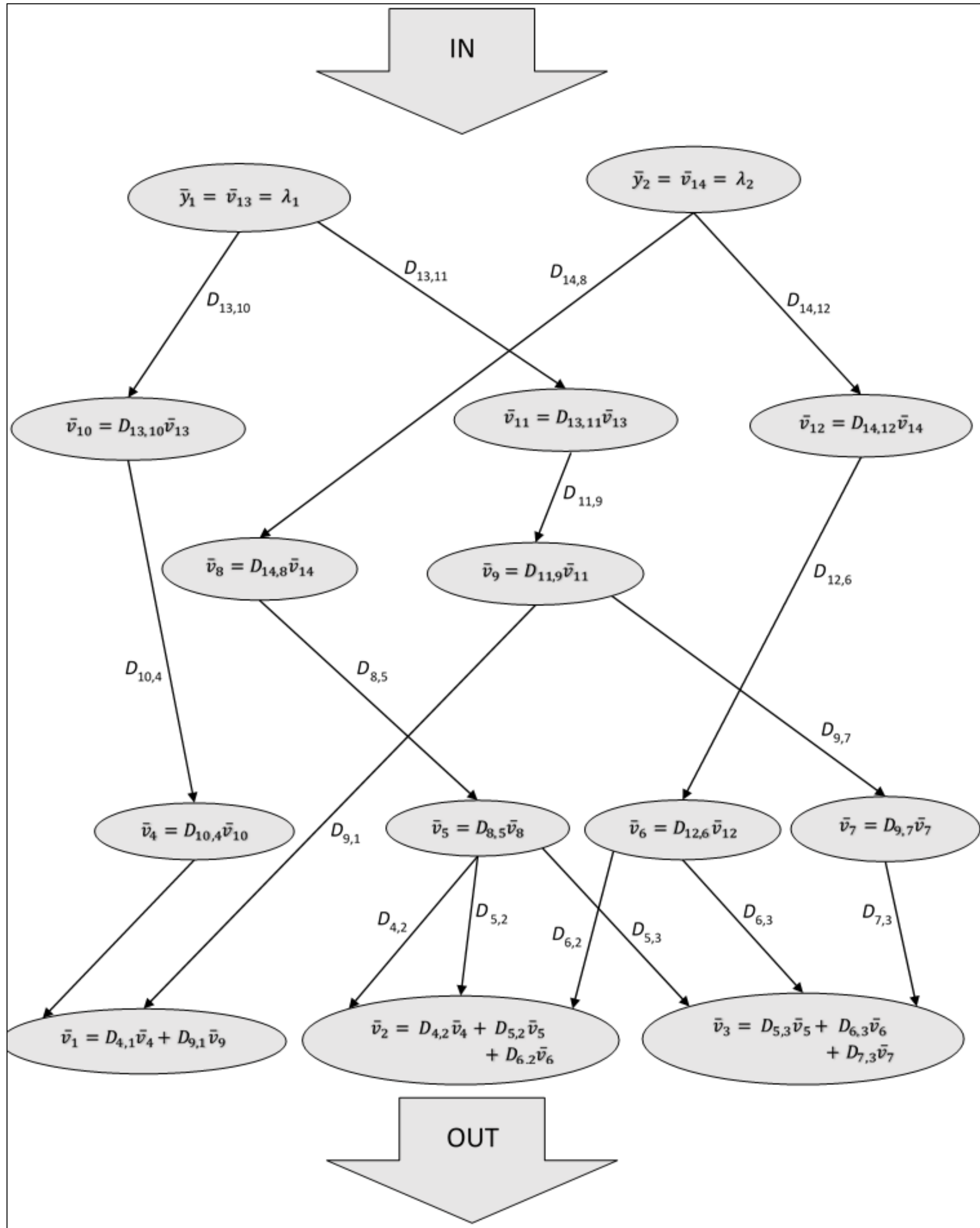


Fig. 5.3: Computational graph for the adjoint mode of differentiation of function (5.1).

Chapter 6

Implementing AD in MATLAB

The example in the previous section should serve to illustrate the mechanical nature in which the tangents or adjoints can be propagated and the gradients ultimately obtained. Because of this, this procedure can be automated, and various AD tools have been developed which enable the automatic calculation of derivatives in both the forward and adjoint modes.

There are two broad categories of tools used in the implementation of algorithmic differentiation, namely source code transformation and operator overloading. Tools falling under the former category take in the source code of the function as input and return source code implementing the chosen mode of AD. This approach is the more complex of the two. The simpler alternative, operator overloading, exploits the versatility of object-orientated programming languages to define new data types, capable of storing both value and derivative information, and 'overloading' — overwriting and replacing with user-defined functions — the programming language's internal functions to enact both value and derivative computations. Neidinger (2010) provides an insightful introduction into this area, and informs a great deal of the following section.

6.1 Implementing the Forward Mode Using Operator Overloading

The objective of the operator overloading approach to AD is to extend the input to include the desired derivatives and to enable the programming operations to produce both function values and numeric derivative values. The approach hinges on the user's ability to define new data types within an object-orientated programming (OOP) environment and overload the internal instructions of the programming language. The implementation of this approach, particularly within the MATLAB programming environment, has been described by Neidinger (2010), who defines the '`valder`' object class, Forth (2006), who uses the '`fmad`' object class, and

Verma (1999) who uses the '`deriv`' object class in his guide to the ADMAT toolbox. Here, the '`solder`' object class is defined and used in implementation in much the same way.

As an example of how the forward mode of AD might be implemented using operator overloading in a Monte Carlo setting, the most basic case is considered: that of determining the first order Greeks of a standard European call option in the Black Scholes model.

Consider a one-dimensional stock price process S_t with the following risk-neutral dynamics:

$$dS_t = S_t(rdt + \sigma dW_t)$$

with constant riskless rate r, so that:

$$S_T = S_t \exp((r - \tfrac{1}{2}\sigma^2)(T-t) + \sigma(W_T - W_t)) \,.$$

Consider also a standard European call option written on such a stock, with terminal payoff:

$$X_T = f(S_T) = \max(S_T - K, 0) = (S_T - K)^+$$

and accordingly, price given by:

$$V_t = e^{-r(T-t)}\mathbb{E}_{\mathbb{Q}}[(S_T - K)^+],$$

where $\mathbb{Q}$ is the associated risk-neutral measure. Within this model, the first order Greeks — $\Delta = \frac{\partial f}{\partial S_t}$, $\Theta = \frac{\partial f}{\partial t}$, $\nu = \frac{\partial f}{\partial \sigma}$, $\rho = \frac{\partial f}{\partial r}$, are easily derived in closed form, as follows:

$$\begin{aligned}
d_1 =& \frac{\ln(\frac{S_t}{K}) + (r - \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}}, \quad d_2 = d_1 - \sigma\sqrt{T-t} \\
\Delta_{call} =& \Phi(d_1) \\
\rho_{call} =& (T-t)Ke^{(-r(T-t))\Phi(d_2)} \\
\nu_{call} =& S_t\phi(d_1)\sqrt{T-t} \\
\Theta_{call} =& -rKe^{-r(T-t)}\Phi(d_2) - S_t\phi(d_1)\frac{\sigma}{2\sqrt{T-t}} \,.
\end{aligned}$$

We now illustrate how one might alternatively calculate these Greeks using the forward mode of AD in a Monte Carlo setting. This initially requires the construction of the `solder` object class. The methodology applied here is strongly guided by the approach of Neidinger. The full code defining the `solder` object class is given in Figure C.1 in Appendix C.1. The `solder` object class has two properties, namely `sol` - the function 'solution', or value - and `der`, its derivative or gradient vector.

After establishing these properties, the 'methods' applicable to `solder` objects are defined. These are functions which act only on `solder` objects, and functions not defined here will not run on `solder` objects. The `solder` function converts an input `(a,b)` into a `solder` object, while the `doubleconvert` function converts a `solder` object into a 1×2 array of type `double`, in order to access the `solder` object's solution and derivative values.

Subsequent functions listed under methods serve to overload MATLAB's built-in elementary operations, `plus`, `minus`, `uminus`, `exp`, `log`, `times`, `rdivide`, `power`, `sqrt`, and `max`, with functions which instead act on both of a `solder` object's `sol` and `der` properties. Note that the functions overloaded here are only those required in this simple example, and one would be required to similarly overload any other function for use in a program that called for it. For a function with inputs `(x,y)`, there is a check whether either of `x` or `y` is a scalar, and a set of `if`, `elseif` statements to accommodate accordingly. The functions are overloaded to return the original function operation on the `solder` object's `sol` property, while the standard rules of differentiation are applied to calculate the value of its `der` property. In the case of the `power` function, by rewriting x^y as $\exp(y \times \log(x))$, we are able to call the overloaded `exp`, `.*` and `log` functions in its definition.

Returning to the example at hand: consider a standard European call option written on a stock, with dynamics described above, and the following model parameters:

$$S_0 = 100, \qquad K = 100, \qquad r = 0.06, \qquad T = 1, \qquad t = 0, \qquad \sigma = 0.2\,.$$

Using a sample size of 40 000, the Monte Carlo estimates produced by this approach, and their analytical counterparts, are given in Table 6.1.

Tab. 6.1: Comparison of price and Greeks calculated with forward-mode AD and respective analytical solutions

	Forward mode AD Monte Carlo estimate	Analytical solution	Difference	Relative difference
Price	10.91039738	10.98954915	-0.07915177	-0.7255%
Δ	0.653467418	0.655421742	-0.001954324	-0.2991%
ρ	54.43634445	54.55262501	-0.11628056	-0.2136%
ν	36.38725719	36.82701403	-0.43975684	-1.2085%
Θ	-6.904906386	-6.955858904	0.050952518	-0.7379%

The code implementing this approach is given in Figure C.4 in Appendix C.1. This approach is clearly inefficient and serves simply as an introductory example

as to the implementation of forward mode in a Monte Carlo setting. Any efficient application requires one to exploit MATLAB's vector- and matrix-based functions.

6.2 Implementing Adjoint Mode AD Using ADMAT 2.0

As an illustration of how one might implement the adjoint mode in a Monte Carlo setting, and of how one might use one of the AD toolboxes built for the MATLAB environment, we consider the case of a European basket option and calculate its deltas — the sensitivity of the option price to each constituent stock in the basket — using ADMAT 2.0, an object-orientated automatic differentiation toolbox developed by Cayuga Research (Verma, 1999). This option is also used as an illustrative example by Capriotti (2011).

A basket call option has payout

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

where $X(T) = (X_1(T), \ldots, X_N(T))$ gives the value of a set of N correlated underlying assets at time T, $w_i, i = 1, \ldots, n$ are the weights associated with the composition of the basket, K is the strike price and r is the risk-free yield for the maturity considered.

The Greeks are calculated in both forward and adjoint modes, implemented using ADMAT 2.0, for $N = 2, \ldots 50$. The computational costs of each approach, given by the ratio of the average run time to compute Greeks and price, to the average run time to compute price only as per (4.16), is then compared to that of a finite difference approach. This is illustrated in Figure 6.1.

AD is implemented in this context using ADMAT 2.0 in the following manner. ADMAT acts on functions enclosed in `.m` files. These functions must necessarily take in two parameters: the parameter of interest — that is, the variable with respect to which we wish to differentiate the function — and an optional `Extra` variable — a cell structure used to feed extraneous variables to the function. The function code is included in Figure C.5 in Appendix C.2, while the code implementing the use of ADMAT to evaluate the Greeks is included in Figure C.6.

As Figure 6.1 shows, the AD approaches offer a clear computational advantage over a finite difference method. However, the adjoint mode fails to offer a distinct advantage over the forward mode; more troublingly, it fails to adhere to the theoretical cost ratio of four times. This failure is similarly described by Forth (2006), who notes that while ADMAT's operation count agrees with AD theory, its run times do not.

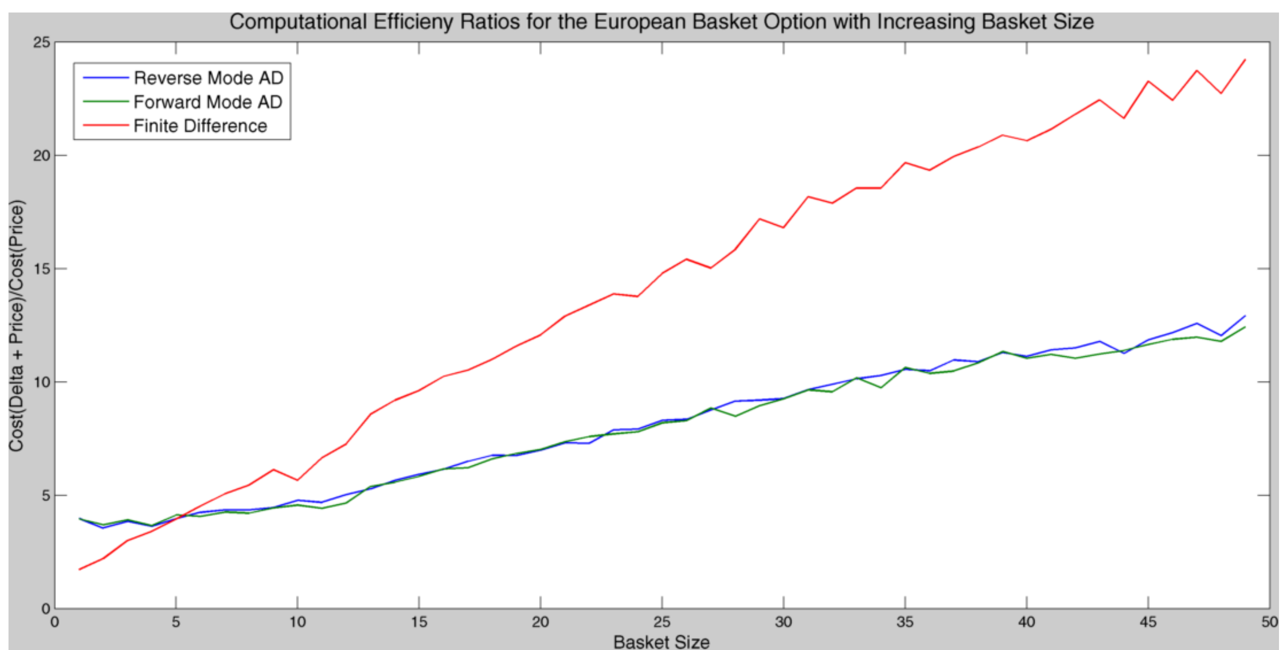


Fig. 6.1: Computational efficiency of the forward and adjoint (reverse) modes of AD in comparison to the finite-difference approach, for the European Basket Option.

Chapter 7

Conclusion

This dissertation has served to introduce and explain the adjoint approach to solving affine recursion problems as it applies to the problem of calculating sensitivities of financial instruments. Furthermore, it has been shown how aspects of this approach can be extended into the paradigm of Adjoint Algorithmic Differentiation.

Following a brief introduction and review of the pertinent literature, the concept of the Affine Recursion Problem was formalised, and the forward and adjoint approaches were introduced. It was illustrated in detail how implementing the adjoint approach instead of the more intuitive forward approach reduces the computational complexity of the algorithm from $\mathcal{O}(Nm^3)$ to $\mathcal{O}(Nm^2)$ where N is the number of steps in the recursion and $D(n) \in \mathbb{R}^{m \times m}$. This can lead to substantial computational savings.

Thereafter, it was illustrated how the adjoint approach could be implemented in the context of the Libor Market Model to solve for a selection of sensitivities — namely, the Greeks Delta and Vega. It was shown empirically that the adjoint and forward approaches offered substantial computational savings over the traditional approach of implementing a finite difference scheme, and furthermore, that the adjoint approach generally outperformed the forward approach.

It was then explained how many of the same concepts applicable in the context of affine recursion problems can be used to develop the concept of algorithmic differentiation, and specifically adjoint algorithmic differentiation, which is analogous to the adjoint approach to solving affine recursion problems. Via a simple example, it was illustrated how a computer program solves a given function, and how a program can be extended to propagate the adjoints and in that way enable the simultaneous and efficient calculation of derivatives.

Following this, it was shown via a practical example how the forward mode of algorithmic differentiation can be implemented in MATLAB using the operator overloading approach. Thereafter, it was illustrated how the adjoint mode can be implemented using one of the many toolboxes developed for using adjoint algo-

rithmic differentiation within the MATLAB programming environment — namely, ADMAT 2.0. It was shown that the forward and adjoint modes offered significant computational savings over the finite difference approach. The performance difference between the forward and adjoint modes were however not as pronounced as expected.

It is hoped that this dissertation will stand as a useful and detailed introduction into the concepts surrounding the adjoint approach to solving affine recursion problems and adjoint alogirithmic differentiation. There is much left undiscussed here — for example, the application of adjoint methods to instruments with Bermuda-style payoffs, or to the area of credit valuation adjustments — and these areas remain topics of great interest for researchers and practitioners alike. Many exciting and useful developments in the field have been documented in the last decade, and this trend will likely continue as the adoption of adjoint methods becomes increasingly ubiquitous.

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

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. B. (2010). Fast Correlation Greeks by Adjoint Algorithmic Differentiation.
URL: *http://dx.doi.org/10.2139/ssrn.1587822*

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

Chan, J. H., Joshi, M. S. and Zhu, D. (2010). First and Second Order Greeks in the Heston Model.
URL: *http://dx.doi.org/10.2139/ssrn.1718102*

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*

Forth, S. A. (2006). An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB, *ACM Transactions on Mathematical Software* **32**(2): 195–222.

Giles, M. and Glasserman, P. (2005). Smoking Adjoints: Fast Evaluation of Greeks in Monte Carlo Calculations, *RISK* .

Glasserman, P. (2003). *Monte Carlo Methods in Financial Engineering*, Springer Science & Business Media.

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.

Henrard, M. (2013). Calibration in Finance: Very Fast Greeks Through Algorithmic Differentiation and Implicit Function, *Procedia Computer Science* **18**: 1145–1154.

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

Joshi, M. and Pitt, D. (2010). Fast Sensitivity Computations for Monte Carlo Valuation of Pension Funds, *Astin Bulletin* **40**(02): 655–667.

Joshi, M. and Yang, C. (2011). Fast Delta Computations in the Swap-Rate Market Model, *Journal of Economic Dynamics and Control* **35**(5): 764–775.

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.

Neidinger, R. D. (2010). Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming, *SIAM Review* **52**(3): 545–563.

Savickas, V., Hari, N., Wood, T. and Kandhai, D. (2014). Super Fast Greeks: an Application to Counterparty Valuation Adjustments, *Wilmott* **2014**(69): 76–81.

Verma, A. (1999). ADMAT: Automatic Differentiation in MATLAB Using Object Oriented Methods, *SIAM Interdiscplinary Workshop on Object Oriented Methods for Interoperability*, pp. 174–183.

Appendix A

Code to Implement Forward and Adjoint ARP Approaches

A.1 Code to Implement Euler Approximations for Delta in the Libor Market Model

```
% LIBOR Market Model Simulation

function LIBORs = LMM_Sim(LInit,m,NumPaths,Sigma,Tau,Z)

%Z = randn(m,NumPaths);
LIBORs = zeros(m,m,NumPaths);

S = zeros(m,1);

for j = 1:NumPaths
LIBORs(:,1,j)=LInit;
end

for j = 1:NumPaths
    for n = 1:m-1
        S = Sigma.*Tau.*LIBORs(:,n,j)./(1+Tau.*LIBORs(:,n,j));
        S(1:n)=0;
        S = cumsum(S);
        S = exp( Sigma.*( (S-Sigma*0.5)*Tau(n)+ Z(n,j)*sqrt (Tau(n))));
        LIBORs(n+1:m,n+1,j)=LIBORs(n+1:m,n,j).*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 `LMM_Sim` function, used to simulate Libor Market Model realisations (Kienitz and Wetterau, 2012).

```
function D = MatrixDBuilder(LIBORs,m,Sigma,Tau,j)

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

for n = 1:m-1
    %constructing diagonal
    v = LIBORs(:,n+1,j)./LIBORs(:,n,j)+Sigma.^2.*(Tau*Tau(n)).*LIBORs(:,n+1,j)...
        ./((1+Tau.*LIBORs(:,n,j)).^2);
    v(1:n) = 1;
    D(:,:,n)=diag(v);
    %constructing subdiagonal entries
    x = Sigma.*Tau./((1+Tau.*LIBORs(:,n,j)).^2);
    y = LIBORs(:,n+1, j).*Sigma;
    A = y*x.';
    sec = n+1:m;
    D(sec,sec,n)=D(sec,sec,n)+tril(A(sec,sec),-1);
end
end
```

Fig. A.2: The `MatrixDBuilder` function, used to build the $D(n)$ matrices (Kienitz and Wetterau, 2012).

```
%building V matrix
VNmat = zeros(m,NumPaths);
DFs = 1./(1+repmat(Tau,1,NumPaths).*(reshape(LIBORs(:,m,:),m,NumPaths)));
MaxFunc = max((reshape(LIBORs(m,m,:),1,NumPaths))-K,0);
PVs = prod(DFs);
Ind = ((reshape(LIBORs(m,m,:),1,NumPaths))>K);
VNmat(end,:) = Tau(end).*PVs.*(Ind-Tau(end).*DFs(end,:).*MaxFunc);
for i = 1:m-1
    VNmat(i,:)=Tau(end).*PVs.*MaxFunc.*(-Tau(i).*DFs(i,:));
end
```

Fig. A.3: An algorithm to construct $V(N)$.

```
%Implementing forward method
for p = 1:NumPaths
    D = MatrixDBuilder(LIBORs,m,Sigma,Tau,p);
    m = size(D,1);
    N = size(D,3)+1;
    A = zeros(m,m,N);
    A(:,:,1) = Delta0;
    V = VNmat(:,p);
    for n = 1:N-1
        A(:,:,n+1) = D(:,:,n)*A(:,:,n);
    end
    w = V.'*A(:,:,N);
    Delta_for = Delta_for+w;
end
Delta_for=Delta_for/NumPaths;
Delta_for = Delta_for';
```

Fig. A.4: Implementing the forward method (Kienitz and Wetterau, 2012).

```
%Implementing adjoint method
for p = 1: NumPaths
    D = MatrixDBuilder(LIBORs,m,Sigma,Tau,p);
    N = size(D,3)+1;
    V = VNmat(:,p);
    for n = N-1:-1:1
        V = D(:,:,n).'*V;
    end
    Delta_adj = Delta_adj+V;
end
Delta_adj = Delta_adj/NumPaths;
```

Fig. A.5: Implementing the adjoint method (Kienitz and Wetterau, 2012).

```
%Implementing finite difference method

for i = 1:m
    LInitLeft=LInit;
    LInitRight=LInit;
    LInitLeft(i)=LInitLeft(i)-epsilon;
    LInitRight(i)=LInitRight(i)+epsilon;

    LIBORsLeft= LLM_Sim(LInitLeft,m,NumPaths,Sigma,Tau,Z);
    LIBORsRight=LLM_Sim(LInitRight,m,NumPaths,Sigma,Tau,Z);

    MaxFuncLeft = max((reshape(LIBORsLeft(m,m,:),1,NumPaths))-K,0);
    MaxFuncRight = max((reshape(LIBORsRight(m,m,:),1,NumPaths))-K,0);

    DFsLeft = 1./(1+repmat(Tau,1,NumPaths).*(reshape(LIBORsLeft(:,m,:),m,NumPaths)));
    DFsRight = 1./(1+repmat(Tau,1,NumPaths).*(reshape(LIBORsRight(:,m,:),m,NumPaths)));


    PVsLeft = prod(DFsLeft);
    PVsRight = prod(DFsRight);

    PriceLeft = mean(PVsLeft.*MaxFuncLeft*Tau(end));
    PriceRight = mean(PVsRight.*MaxFuncRight*Tau(end));

    Delta_fd(i) = (PriceRight-PriceLeft)/(2*epsilon);

end

Delta_fd=Delta_fd';
```

Fig. A.6: Implementing a finite difference scheme (Kienitz and Wetterau, 2012).

A.2 Code to Implement Euler Approximations for Vega in the Libor Market Model

```
function B = MatrixBBuilder(LIBORs,Z,m,Sigma,Tau,j)

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

for n = 1:m-1
    x = LIBORs(:,n+1,j).*Sigma;
    y = Tau(1).*Tau.*LIBORs(:,n,j)./(1+Tau.*LIBORs(:,n,j));
    %Tau(1) at start is h in formula
    A = x*y.';
    A = tril(A(n+1:m,n+1:m),-1);
    B(n+1:m,n+1:m,n)=A;
    Si = Tau.*LIBORs(:,n,j).*Sigma./(1+Tau.*LIBORs(:,n,j));
    Si(1:n)=0;
Si = cumsum(Si);
x = LIBORs(:,n+1,j).*(Si.*Tau -Sigma.*Tau +Z(n+1,:).*sqrt(Tau)+...
    Sigma.*Tau.*LIBORs(:,n,j)./(1+Tau.*LIBORs(:,n,j)));
    x(1:n) = 0;
    B(:,:,n) = B(:,:,n)+diag(x);
end

end
```

Fig. A.7: The `MatrixBBuilder` function, used to build the $B(n)$ matrices (Kienitz and Wetterau, 2012).

```
%Implementing forward method
        Vega0 = zeros(m);
        for p = 1:NumPaths
            D =MatrixDBuilder(LIBORs,m,Sigma,Tau,p);
            q = size(D,1);
            N = size(D,3)+1;
            A = zeros(q,q,N);
            A(:,:,1) = Vega0;
            v = VNmat(:,p);
             B = MatrixBBuilder(LIBORs,Z(:,p),m,Sigma,Tau,p);
            for n = 1:N-1
                A(:,:,n+1) = D(:,:,n)*A(:,:,n) +B(:,:,n);
            end
            w = v.'*A(:,:,N);
            Vega_for = Vega_for+w;
        end
        Vega_for=Vega_for/NumPaths;
        Vega_for = Vega_for';
```

Fig. A.8: Implementing the forward method where the ARP includes a translation term (Kienitz and Wetterau, 2012).

```
%Implementing adjoint method
        for p = 1: NumPaths
            D =MatrixDBuilder(LIBORs,m,Sigma,Tau,p);
            B = MatrixBBuilder(LIBORs,Z(:,p),m,Sigma,Tau,p);
            N = size(D,3)+1;
            V = VNmat(:,p);
            Vbar = 0;
            for n = N-1:-1:1
                Vbar = Vbar + B(:,:,n).' * V;
                V = D(:,:,n).'*V;
            end
            Vega_adj = Vega_adj+Vbar';
        end
        Vega_adj = Vega_adj/NumPaths;
```

Fig. A.9: Implementing the adjoint method where the ARP includes a translation term (Kienitz and Wetterau, 2012).

Appendix B

Average Runtimes

B.1 Average Runtimes - Delta

Tab. B.1: Average Runtimes of the Various Approaches; Delta.

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price	Price + Delta	Price	Price + Delta	Price	Price + Delta
1	0.004	0.009	0.006	0.011	0.003	0.008
2	0.009	0.033	0.010	0.028	0.007	0.029
3	0.012	0.049	0.014	0.048	0.012	0.077
4	0.017	0.069	0.022	0.069	0.017	0.146
5	0.023	0.092	0.023	0.085	0.023	0.240
6	0.028	0.111	0.029	0.112	0.028	0.356
7	0.034	0.136	0.034	0.122	0.034	0.493
8	0.040	0.158	0.040	0.143	0.040	0.655
9	0.045	0.209	0.046	0.164	0.045	0.840
10	0.051	0.207	0.051	0.186	0.051	1.049
11	0.057	0.234	0.057	0.212	0.057	1.286
12	0.063	0.254	0.062	0.232	0.063	1.591
13	0.069	0.289	0.068	0.255	0.073	1.959
14	0.078	0.325	0.081	0.286	0.078	2.330
15	0.080	0.347	0.081	0.305	0.093	2.613
16	0.087	0.364	0.087	0.335	0.088	3.088
17	0.093	0.421	0.093	0.378	0.103	3.494
18	0.100	0.457	0.104	0.412	0.103	3.811
19	0.106	0.473	0.107	0.413	0.108	4.201
20	0.112	0.496	0.112	0.438	0.110	4.508
21	0.118	0.535	0.118	0.522	0.116	5.067
22	0.124	0.593	0.124	0.558	0.123	5.504
23	0.130	0.638	0.132	0.541	0.132	6.112
24	0.147	0.685	0.138	0.568	0.137	6.636

Continued on next page

Table B.1 – continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price	Price + Delta	Price	Price + Delta	Price	Price + Delta
25	0.154	0.724	0.145	0.601	0.142	7.254
26	0.159	0.774	0.150	0.633	0.149	8.334
27	0.169	0.827	0.159	0.675	0.161	9.081
28	0.182	0.906	0.174	0.719	0.170	9.346
29	0.194	0.984	0.171	0.737	0.173	10.032
30	0.202	1.046	0.178	0.777	0.175	10.706
31	0.223	1.137	0.185	0.821	0.182	12.034
32	0.215	1.157	0.193	0.865	0.197	12.799
33	0.226	1.218	0.198	0.930	0.200	13.181
34	0.220	1.488	0.205	0.977	0.203	14.016
35	0.236	1.521	0.241	1.024	0.209	15.384
36	0.239	1.549	0.220	1.032	0.231	16.349
37	0.249	1.714	0.226	1.087	0.224	16.846
38	0.256	1.819	0.232	1.127	0.230	19.196
39	0.261	2.011	0.240	1.226	0.247	19.162
40	0.261	1.934	0.248	1.277	0.245	19.849

B.2 Average Runtimes - Vega

Tab. B.2: Average Runtimes of the Various Approaches; Vega.

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price	Price + Vega	Price	Price + Vega	Price	Price + Vega
1	0.005	0.011	0.004	0.008	0.005	0.011
2	0.010	0.058	0.008	0.052	0.008	0.031
3	0.012	0.094	0.015	0.104	0.012	0.078
4	0.017	0.133	0.022	0.145	0.017	0.151
5	0.023	0.177	0.028	0.184	0.023	0.243
6	0.029	0.226	0.032	0.220	0.028	0.369
7	0.036	0.265	0.035	0.258	0.034	0.497
8	0.040	0.297	0.041	0.316	0.040	0.663
9	0.045	0.342	0.048	0.345	0.046	0.873
10	0.050	0.393	0.053	0.401	0.051	1.077

Continued on next page

Table B.2 – continued from previous page

	Forward Method		Adjoint Method		Finite Difference Scheme	
N	Price	Price + Vega	Price	Price + Vega	Price	Price + Vega
11	0.057	0.439	0.059	0.449	0.058	1.306
12	0.063	0.474	0.067	0.514	0.064	1.567
13	0.069	0.545	0.078	0.581	0.070	1.852
14	0.075	0.619	0.081	0.628	0.075	2.166
15	0.082	0.652	0.088	0.668	0.087	2.510
16	0.087	0.683	0.096	0.713	0.087	2.865
17	0.094	0.741	0.099	0.775	0.094	3.242
18	0.099	0.803	0.104	0.828	0.099	3.645
19	0.105	0.873	0.110	0.879	0.106	4.116
20	0.112	0.920	0.118	0.953	0.112	4.604
21	0.118	1.038	0.130	1.137	0.123	5.103
22	0.137	1.096	0.166	1.221	0.126	5.627
23	0.131	1.174	0.158	1.287	0.131	6.161
24	0.139	1.228	0.150	1.272	0.138	6.777
25	0.146	1.307	0.154	1.295	0.146	7.349
26	0.163	1.441	0.194	1.444	0.160	8.047
27	0.159	1.488	0.189	1.652	0.164	9.258
28	0.165	1.535	0.189	1.744	0.175	9.437
29	0.172	1.649	0.224	2.152	0.172	10.093
30	0.178	1.747	0.265	2.212	0.178	10.867
31	0.189	1.914	0.218	1.973	0.186	11.645
32	0.209	1.994	0.224	2.048	0.192	12.510
33	0.201	2.122	0.224	2.030	0.200	14.396
34	0.209	2.540	0.212	2.086	0.219	15.061
35	0.219	2.811	0.219	2.148	0.214	15.231
36	0.248	2.863	0.232	2.321	0.222	16.266
37	0.245	3.008	0.256	2.473	0.232	17.394
38	0.243	3.191	0.265	2.678	0.238	18.125
39	0.259	3.497	0.266	2.811	0.246	20.906
40	0.254	3.586	0.274	3.019	0.271	22.360

Appendix C

Code to Implement AD in Matlab

C.1 The `solder` Object Class

```
classdef solder

    properties
        sol %function solution
        der %function derivative or gradient vector
    end
    methods
        function obj = solder(a,b)
            %solder class constructor
            if nargin == 0 % not intended for use
                obj.sol = [];
                obj.der = [];
            elseif nargin == 1 % for constant with derivative 0
                obj.sol = a;
                obj.der = 0;
            else
                obj.sol = a;
                obj.der =b;
            end
        end
        function output = doubleconvert(obj)
            output = [obj.sol, obj.der];
            %converts solder object to a vector of doubles
        end
```

Fig. C.1: Full code defining the `solder` object class (for the functions required in the examples provided).

```
function z = plus(x,y)
            %overloads addition (+) with at least one solder argument
            if ~isa(x,'solder') %if x is a scalar
                z = solder(x+y.sol,y.der);
            elseif ~isa(y, 'solder') %if y is a scalar
                z = solder(x.sol+y, x.der);
            else
                z = solder(x.sol+y.sol, x.der+y.der);
            end
        end

        function z = minus(x,y)
            %overloads subtraction (-) with at least one solder argument
            if ~isa(x, 'solder') %if x is a scalar
                z = solder(x-y.sol, -y.der);
            elseif ~isa(y, 'solder') %if y is a scalar
                z = solder(x.sol-y, x.der);
            else
                z = solder(x.sol-y.sol,x.der-y.der);
            end
        end

        function z = uminus(x)
            % overloads negation (-) with a solder argument
            z = solder(-x.sol,-x.der);
        end

        function z = exp(x)
            %overloads exp with a solder argument
            z = solder(exp(x.sol), exp(x.sol).*x.der);
        end

        function z = log(x)
            %overloads natural logarithm with a solder argument
            z = solder(log(x.sol), 1./x.sol .* x.der);
        end

        function z = times(x,y)
            %overloads element-wise multiplication (.*) with at least one
            %solder argument
            if ~isa(x,'solder') %if x is a scalar
                z = solder(x.*y.sol, x.*y.der);
            elseif ~isa(y,'solder') %if y is a scalar
                z = solder(y.*x.sol, y.*x.der);
            else
                z = solder(x.sol.*y.sol, x.der.*y.sol + x.sol.*y.der);
            end
        end
```

Fig. C.2: Full code defining the `solder` object class (continued).

```
        function z = rdivide(x,y)
            %overloads element-wise division (./) with at least one solder
            %argument
            if ~isa(x,'solder') %if x is a scalar
                z = solder(x./y.sol,(-x.* y.der)./((y.sol).^2));
            elseif ~isa(y, 'solder') %if y is a scalar
                z=solder(x.sol./y, x.der./y);
            else
                z = solder(x.sol./y.sol, (x.der.*y.sol-x.sol.*y.der)...
                    ./((y.sol).^2));
            end
        end

        function z = power(x,y)
            %overloads element-wise power (.^) with at least one solder
            %argument
            if ~isa(x,'solder') %if x is a scalar
                z = solder(x.^y.sol, x.^y.sol.*log(x).*y.der);
            elseif ~isa(y, 'solder') %if y is a scalar
                z = solder(x.sol.^y, y.*x.sol.^(y-1).*x.der);
            else
                z = exp(y.*log(x)); %calls overloaded exp, .* and log
            end
        end

        function z = sqrt(x)
            %overloads square root with a solder argument
            z = solder(sqrt(x.sol),x.der./(2*sqrt(x.sol)));
        end

        function z = max(x,y)
            %overloads max function with at least one solder argument
            if ~isa(x,'solder') %if x is a scalar
                z = solder(max(x,y.sol), y.der * (y.sol > x));
            elseif ~isa(y,'solder') %if y is a scalar
                z = solder(max(x.sol,y), x.der * (x.sol > y));
            else
                z = solder(max(x.sol,y.sol), y.der*(y.sol>x.sol)...
                    +x.der*(x.sol>y.sol));
            end
        end
    end
end
```

Fig. C.3: Full code defining the `solder` object class (continued).

```
% Model parameters
S0 = 100;
K = 100;
r = 0.06;
sigma = 0.2;
T = 2;
t = 1;
% Size of random sample
m =40000;
%convert model parameters to solder objects
S0 = solder(S0, [1 0 0 0]);
r = solder(r, [0 1 0 0]);
sigma = solder(sigma, [0 0 1 0]);
t = solder(t, [0 0 0 1]);
%initialise matrix to store vector of sample prices and Greeks
PriceGreeksMat = zeros(m,5);

%compute sample payoffs and Greeks
for i = 1:m
    Z = randn(1);
    ST = S0.*exp((r-0.5.*sigma.^2).*(T-t)+sigma.*sqrt(T-t).*Z);
    payoff = exp(-r.*(T-t)).*max(ST-K,0);
    PriceGreeksMat(i,:) = doubleconvert(payoff);
end
%take average of samples to arrive at price and Greeks estimates
Price = mean(PriceGreeksMat(:,1));
Delta = mean(PriceGreeksMat(:,2));
Rho = mean(PriceGreeksMat(:,3));
Vega = mean(PriceGreeksMat(:,4));
Theta = mean(PriceGreeksMat(:,5));
```

Fig. C.4: MATLAB code implementing forward mode AD for the European call option using the `solder` object class.

```
%Basket Option

function Price = BasketOptionADMAT(S0,Extra)
%feed Z already transformed by Cholesky decomp of correlation matrix
r=Extra.r;
sigmavec=Extra.sigmavec;
dt = Extra.dt;
Z = Extra.Z;
K=Extra.K;
m = Extra.m;
STOCKS = repmat(S0,1,m).*(exp((r-0.5.*sigmavec.^2)*dt + sigmavec.*sqrt(dt).*Z));

PAYOFFS = exp(-r*dt).*max(sum(STOCKS)-K,0);

Price = sum(PAYOFFS)/m;

end
```

Fig. C.5: The BasketOptionADMAT function, which can be evaluated using ADMAT 2.0.

C.2 MATLAB Code Implementing AD Evaluation of Basket Option, Using ADMAT 2.0

```
T = 1.5;
r = 0.1;
dt = T;
m = 10000;

STOCKNUMS = 2:50;
Extra.dt = dt;
Extra.r = r;
DeltaRuntimesBasketADMAT = zeros(1,length(STOCKNUMS));
PriceRuntimesBasketADMAT = zeros(1,length(STOCKNUMS));
runs = 20;
for w = 1:runs %repeats the process so that we get an average run time
    for i = 1:length(STOCKNUMS) % loops through basket sizes
        rng(0)
        NumStocks = STOCKNUMS(i);
        S0 = 10 + (200-10).*rand(NumStocks,1);
        % initial stock prices randomly distributed between 10 and 200
        K = sum(S0); %at the money
        sigmavec = 0.1 + (0.6-0.1).*rand(length(S0),1);
        %variances randomly distributed between 0.1 and 0.6;
        sigmavec=repmat(sigmavec,1,m);
        [E, ignore] = RandomCorr(length(S0),0.9);
        L = chol(E, 'lower');
        tic
        Z1 = L*randn(size(S0,1),m);

        Extra.dt = dt;
        Extra.r = r;
        Extra.K = K;
        Extra.Z=Z1;
        Extra.sigmavec=sigmavec;
        Extra.m=m;
        Price=BasketOptionADMAT(S0,Extra);
        PriceRuntimesBasketADMAT(i)=PriceRuntimesBasketADMAT(i)+toc;
        rng(0)
        tic
        clear Price Extra Deltas Z1
        Z1 = L*randn(size(S0,1),m);
        Extra.dt = dt;
        Extra.r = r;
        Extra.K = K;
        Extra.Z=Z1;
        Extra.sigmavec=sigmavec;
        Extra.m=m;
        myfun=ADfun('BasketOptionADMAT',1);
        [Price,Deltas] = feval(myfun,S0,Extra);
        DeltaRuntimesBasketADMAT(i)=DeltaRuntimesBasketADMAT(i)+toc;
        clear Price Extra Deltas NumStocks S0 K sigmavec E ignore L Z1
    end
end

PriceRuntimesBasketADMAT=PriceRuntimesBasketADMAT/runs;
DeltaRuntimesBasketADMAT=DeltaRuntimesBasketADMAT/runs;
DeltaPriceRatiosBasketADMAT=...
    DeltaRuntimesBasketADMAT./PriceRuntimesBasketADMAT;
```

Fig. C.6: Implementing AD to find the Greeks of the basket option, using ADMAT 2.0.