University of Birmingham

Masters Degree Thesis

Sensitivities: A Numerical Approach

Author:
Matthew Robinson

Supervisor:
Daniel J. Duffy

Student Number ID: 1346964

A thesis submitted in partial fulfilment of the requirements for the degree of MSc Mathematical Finance

in the

Department of Economics
Birmingham Business School

UNIVERSITY OF BIRMINGHAM

Abstract

MSc Mathematical Finance

Sensitivities: A Numerical Approach

by Matthew ROBINSON

Sensitivity analysis is widespread across multiple disciplines, focusing on the effect a parameter change has on some mathematical model. It is widely accepted that the analysis of sensitivities in finance is of extreme importance, it can inform investors of the behaviour of financial instruments concerning the current state of the market; indicating the sensibility of an investment in bonds or option.

This paper identifies potential methods used to approximate the Black-Scholes option Greeks and bonds that follow the Cox-Ingersoll-Ross (CIR) interest rate process. These methods include the Divided Difference method, Cubic Spline method, Forward Automatic Differentiation, Complex-Step Method, Method Of Lines scheme, Crank Nicolson method, Alternating Directional Explicit method and the Continuous Sensitivity Equation approach.

The paper expands on the mathematical background of each method and attempts to approximate such sensitivities using the C++ and MATLAB coding environments. An emphasis is placed on the Continuous Sensitivity Equation approach and the issues that present itself when applying the approach to the approximation of bond sensitivities.

An application of Fichera theory is applied to the Cox-Ingersoll-Ross bond pricing PDE to identify the conditions necessary to implement boundary conditions. The application of Fichera theory and the Continuous Sensitivity Equation approach has led to a potential area of new research concerning the uniqueness of such approximations and whether such equations are well-posed.

Finally, the importance of choosing the correct boundary and initial conditions is investigated using the Greek Rho partial differential equation. In closing, it is found that more research needs to be undertaken before the CSE approach is considered a viable alternative to the Divided Difference and Cubic Spline approach. Despite these findings, the Complex-Step method and Forward Automatic Differentiation methods produce approximation on par with the closed-form sensitivity formulae. It is concluded that the Forward Automatic Differentiation methods ease of implementation into C++ and its accuracy make it a practical alternative to large closed-form sensitivity equation such as the CIR volatility sensitivity equation.

Acknowledgements

A special thanks goes to my supervisor Daniel J. Duffy for putting up with me over these last few months; answering questions and providing me with the content required to complete this thesis. Many euros are owed.

An additional thanks goes to my friends and family for the support despite my stressed and often times agitated demeanour.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Financial Instruments and Sensitivities 2
- 2.1 Black-Scholes Model . . . 2
- 2.2 Black-Scholes Sensitivities . . . 3
- 2.3 Cox-Ingersoll-Ross (CIR) Process . . . 5
- 2.4 CIR Bond Sensitivities . . . 6

3 PDE Finite Difference Methods 9
- 3.1 Motivation . . . 9
  - 3.1.1 Domain Transformation . . . 10
  - 3.1.2 Fichera Theory . . . 10
  - 3.1.3 Finite Difference Grid (Mesh) . . . 11
- 3.2 Crank Nicolson FDM Method . . . 12
  - 3.2.1 Generalised Crank Nicolson . . . 12
  - 3.2.2 Crank Nicolson Black-Scholes Implementation . . . 13
  - 3.2.3 Crank Nicolson CIR PDE Implementation . . . 14
  - 3.2.4 Thomée Scheme Crank Nicolson . . . 16
- 3.3 Alternating Direction Explicit . . . 17
  - 3.3.1 Generalised Alternating Direction Explicit . . . 18
  - 3.3.2 B&C ADE Black-Scholes Implementation . . . 19
  - 3.3.3 B&C ADE CIR PDE Implementation . . . 20
  - 3.3.4 Thomée Scheme ADE . . . 20
- 3.4 Method of Lines . . . 20
  - 3.4.1 Generalised Method of Lines . . . 21
    - Runge Kutta Dormand-Prince ODE Method . . . 21
  - 3.4.2 MOL Black-Scholes Implementation . . . 23
  - 3.4.3 MOL CIR PDE Implementation . . . 23

4 Sensitivity Approximation Methods 24
- 4.1 Divided Difference Method . . . 24
- 4.2 Cubic Spline Interpolation . . . 25
- 4.3 Complex Step Method . . . 28
- 4.4 Forward Automatic Differentiation . . . 29
  - 4.4.1 Dual Numbers . . . 30
- 4.5 Continuous Sensitivity Equation (CSE) . . . 31

4.5.1 Black-Scholes Continuous Sensitivity Equations 32
Delta CSE 32
Gamma CSE 33
Vega CSE 34
Rho CSE 35
4.5.2 Cox-Ingersoll-Ross Continuous Sensitivity Equations . 36
Duration CSE 36
Convexity CSE 36
Speed of Adjustment CSE 37
Volatility CSE 37
4.5.3 Boundary Conditions: Further Research Opportunity? 37
Duration CSE 37
Convexity CSE 38
Speed of Adjustment CSE 39
Volatility CSE 39
Possible Issues... 40

5 Code Implementation 41
5.1 C++ Code Design 41
5.1.1 C++ Definer Code Section 43
Option Class 43
PDE Classes 44
Initial Boundary Value Problem Class 46
5.1.2 C++ Solver Code Section 46
5.2 C++ Numerical Method Implementation 48
5.2.1 C++ Crank Nicolson 50
5.2.2 C++ ADE Implementation 52
5.2.3 C++ Method Of Lines Implementation 54
5.3 C++ Implementation: Approximation Methods 56
5.3.1 Divided Difference 57
5.3.2 Cubic Spline Interpolation 58
5.3.3 Forward Automatic Differentiation 60
5.3.4 Complex Step Method 62

6 Coding Results 65
6.1 Closed Form Solutions 65
6.2 Forward AD, CSM & Closed Form Solutions 66
6.3 Black-Scholes Numerical Method Approximations 67
6.3.1 Black-Scholes Option Price 67
6.3.2 Divided Difference Sensitivity Approximation 69
6.3.3 Cubic Spline Sensitivity Approximation 70
6.3.4 Continuous Sensitivity Equations 71
6.4 Cox-Ingersoll-Ross Numerical Approximations 75

A Option Greek Derivations 80
A.1 Greek Delta 80
A.2 Greek Gamma 81
A.3 Greek Rho 82

A.4 Greek Theta 82
A.5 Greek Vega 83

B Sensitivity Components **84**
B.1 Speed of Adjustment Closed Form 84
B.2 Volatility Closed Form 85

C Continuous Sensitivity Equation Derivations **86**
C.1 Black-Scholes Continuous Sensitivity Equation Derivations . . 86
C.1.1 Greek Delta CSE 86
C.1.2 Greek Gamma CSE 86
C.1.3 Greek Vega CSE 87
C.1.4 Greek Rho CSE 87
C.2 CIR Continuous Sensitivity Equation Derivations 87
C.2.1 Duration CSE 88
Duration CSE Domain Transformation 88
C.2.2 Convexity CSE 88
Convexity CSE Domain Transformation 89
C.2.3 Speed of Adjustment Sensitivity CSE 89
Speed Of Adjustment CSE Domain Transformation . . 89
C.2.4 Volatility Sensitivity CSE 89
Volatility CSE Domain Transformation 90
C.3 Domain Transformation of the Black-Scholes Equation 90
C.3.1 Transformed Vega CSE 90
C.3.2 Transformed Rho CSE 91

D C++ Code Appendix **92**
D.1 Crank Nicolson Definitions C++ Code 92
D.2 Method Of Lines Definitions C++ Code 95
D.3 Black-Scholes Forward AD C++ Code 99

E MATLAB Code Appendix **101**
E.1 MATLAB Crank Nicolson Code 101
E.2 MATLAB Alternating Direction Explicit Code 104

List of Figures

3.1 Implementation of the Thomée Scheme into the Crank Nicolson method. 17
3.2 Buĉková's ADE upward and downward sweep figure [5, pg.311] 17
3.3 Dormand-Prince (1980) Method Coefficient Table [9, pg.23] . . 22

4.1 Automatic Differentiation Simplistic Example 29

5.1 C++ Code Structure Diagram. 42
5.2 C++ IBvp Class cpp file. 42
5.3 C++ Instantiation Flowchart. 43
5.4 C++ Option Class hpp file. 44
5.5 C++ Black-Scholes Class hpp file. 45
5.6 C++ IBvpSolver hpp file. 46
5.7 C++ Crank Nicolson Method hpp file. 47
5.8 C++ Option Pricing Example. 48
5.9 C++ IBvp Results Method. 48
5.10 ThetaResultHold Vector Process. 49
5.11 Crank Nicolson calculate method. 50
5.12 Implementation of the Thomée Scheme into the Crank Nicolson Class. 52
5.13 Alternating Direction Explicit calculate method. 53
5.14 Method Of Lines header file. 55
5.15 Method Of Lines *main.cpp* function. 56
5.16 A Simplified diagram to explain the procedure of calculating Divided Difference sensitivity approximation. 57
5.17 Divided Difference *main.cpp* function. 58
5.18 Cubic Spline *main.cpp* function. 59
5.19 C++ CIR Forward AD Header File 61
5.20 Complex Step Method *main.cpp* CIR template function. 63
5.21 Complex Step Method *main.cpp* CIR sensitivity functions. . . . 64

6.1 CIR bond price results table 75
6.2 CIR bond price approximations 76
6.3 CIR Divided Difference Duration approximations 77
6.4 CIR Divided Difference Convexity approximations 78
6.5 CIR ADE Volatility CSE approximations 79

D.1 C++ Crank Nicolson Definitions File 92
D.2 C++ Crank Nicolson Definitions File 93
D.3 C++ Crank Nicolson Definitions File 94

D.4 C++ Method Of Lines Definitions File 95
D.5 C++ Method Of Lines Definitions File 96
D.6 C++ Method Of Lines Definitions File 97
D.7 C++ Method Of Lines Definitions File 98
D.8 C++ Black-Scholes Forward AD Header File 99
D.9 C++ Black-Scholes Forward AD Header File 100

E.1 C++ CN MATLAB Definitions File 101
E.2 C++ CN MATLAB Definitions File 102
E.3 C++ CN MATLAB Definitions File 103
E.4 C++ CN MATLAB Definitions File 104
E.5 C++ CN MATLAB Definitions File 105
E.6 C++ CN MATLAB Definitions File 106

List of Tables

6.4 Black-Scholes Sensitivity FAD and CSM Results 66
6.5 CIR Bond Sensitivity FAD and CSM Results 67
6.6 Put Option Price Approximations 69
6.7 Divided Difference Sensitivity Approximations 70
6.8 Cubic Spline Sensitivity Approximations 71
6.9 Greek CSE Approximations 72
6.10 Delta CSE Plot 72
6.11 Gamma CSE Plot 73
6.12 Vega CSE Plot 73
6.13 Rho CSE Plot 74
6.14 Rho CSE Approximations with varied initial conditions 75

List of Symbols and Abbreviations

S	Underlying Security
V	Option Price
r	Short Rate
T	Expiration Time / Time to Maturity
σ	Volatility
K	Option Strike Price
$\mathcal{N}()$	Standard Normal Cumulative Distribution Function (CDF)
$n()$	Standard Normal Probability Distribution Function (PDF)
Δ	Delta
Γ	Gamma
θ	Theta
ρ	Rho
$\mathcal{V}$	Vega
κ	Speed of Adjustment
Θ	Long run mean
$\mid$	Such that
$\forall$	For all

d/c/r	Direction-Convection-Reaction
CN	Crank Nicolson
ADE	Alternating Direction Explicit
CSE	Continuous Sensitivity Equation
CSM	Complex Step Method
AD	Automatic Differentiation
IBVP	Initial Boundary Value Problem

1 Introduction

Financial instruments are tradeable assets ranging from cash, bonds and even loans. There exist two forms: cash and derivative instruments. zero-coupon Bonds and European Options will be the focus throughout this thesis where a range of methods are used to calculate sensitivities concerning a change in some model parameters such as the interest rate. In most cases, closed-form solutions do not exist, forcing the use of numerical methods to approximate such sensitivities.

A range of methods are applied to the above financial instruments where the implementation and tractability of each method are expanded on. The results obtained are calculated using C++14 on a 2018 Macbook Pro equipped with a 6-core i7 2.2GHz processor running Visual Studio via the Parallels virtualisation software and MATLAB 9.6, using the standard macOS.

Post introduction in Chapter 2, the Black-Scholes model and the Cox-Ingersoll-Ross zero-coupon bond pricing partial differential equation (PDE) is introduced alongside their respective closed-form sensitivity equations. The theory behind each of the sensitivities is covered.

Within Chapter 3, three numerical methods are covered known as the Method Of Lines (MOL) scheme; the Crank-Nicolson (CN) scheme and the Alternating Direction Explicit (ADE) scheme. In each case, these methods are applied to a generalised diffusion-convection-reaction PDE which can be easily expressed in terms of the models introduced in Chapter 2.

In Chapter 4, the following methods are introduced: Divided Difference; Cubic Spline; Forward Automatic Differentiation; the Complex-Step Method (CSM) and the Continuous Sensitivity Equation (CSE) approach. Each of which is introduced with an emphasis on their functionality in relation to calculating sensitivities.

Chapter 5 expands on how each method is implemented into C++ with the addition of code extracts.

In Chapter 6, the above numerical and approximation methods are applied using C++ and MATLAB with graphical and table representations of the results obtained using each method.

Lastly, a conclusion on the discovered issues and practicality of the implemented methods are given.

2 Financial Instruments and Sensitivities

The calculation of sensitivities comprises of two methods: the approximation of the security, and the application of a second method to calculate the sensitivity.

The first method requires either a closed-form solution to calculate the exact option/bond price or a partial differential equation (PDE) that models the security. The secondary method, once again, is calculated via a closed-form solution or the pricing PDE that is differentiated using a numerical method.

2.1 Black-Scholes Model

The Black-Scholes model was introduced in 1973 by Black and Scholes [3, pg.642] and was developed to price the value of an option given some underlying price. However, the model is fundamentally flawed by the market assumptions imposed on the model, such as:

- A constant interest rate.
- There are no dividend payouts over options lifetime.
- A constant volatility.
- Price follows a Brownian motion movement pattern.
- Future stock prices follow a log-normal distribution.

Despite the above limitations that do not hold in the market; the model is used due to the wide variety of academic literature and that no model can fully capture all aspects of the market. In addition, the Black-Scholes model uses the cumulative distribution function of the Gaussian distribution, and the expectations can offer a simple interpretation. Wilmott (2009) [27, pg.139] remarks that quantitative analysts prefer the closed-form solutions to that of numerical solutions; given the pressures of choosing a model that satisfies that requirements of being: robust, fast, accurate and easy to calibrate.

Consider the Black-Scholes model:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{2.1}$$

Where: $V = V(S, K, r, \sigma, T)$.

The Black-Scholes model is also known as a Diffusion/Convection/Reaction (d/c/r) equation. It follows the layout of models used in science and engineering to describe the movement of some physical quantity in a physical system governed by a diffusion, convection and reaction process. Concerning the Black-Scholes model, d/c/r corresponds to the convexity, drift and discounting term which is as follows:

Convexity: $\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}$.

Drift: $rS\frac{\partial V}{\partial S}$.

Discount Term: $-rV$.

In short, the convexity term measures the amount made from a change in the underlying price given a hedged (Delta) position. The discount term is self-explanatory as it is the valuation of the option in present time. Lastly, the drift consists of the underlying and the risk-free rate which measures the growth of the underlying given the risk-free interest rate. The above PDE is used to price European options; however, under the model assumptions, there exists an analytical closed-form solution for both European put and call options. Consider the European call closed-form solution:

$$C(S(t), t) = S\mathcal{N}(d_1) - Ke^{-rT}\mathcal{N}(d_2). \tag{2.2}$$

$$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} \qquad d_2 = d_1 - \sigma\sqrt{T}.$$

The above analytical solution provides an exact option price with the limitation being the machine precision. The closed-form expression will be used to identify the precise solution to calculate the absolute error of numerical methods in proceeding chapters.

2.2 Black-Scholes Sensitivities

Greeks are option sensitivities obtained by differentiating the option with respect to some parameter, either analytically or numerically. This thesis will focus on the following Greeks: Delta; Gamma; Rho; Theta and Vega. In addition, this subsection will provide the closed-forms for each Greek where their derivations are given in Appendix A.

Consider the option Greek Delta denoted (Δ) which is the sensitivity of some option V concerning changes in the underlying S. In layman's terms, it is a measure on how much the price of an option is expected to change per £1

change in the price of the underlying asset. The closed-form solution of Delta for a Call option is given by:

$$\Delta_{call} = \mathcal{N}(d_1) > 0. \tag{2.3}$$

Noticing that the closed-form contains the normal CDF, a property of Delta presents itself in that the value of Delta cannot be greater than 1 (this will find use in later chapters). The classically Delta is used in Delta Hedging and is found in the derivation of Black-Scholes PDE.

Gamma denoted (Γ) is the sensitivity that corresponds to the second derivative of the option pricing equation with respect to the underlying. Consider the closed-form :

$$\Gamma_{call/put} = \frac{1}{S\sigma\sqrt{2\pi T}} e^{\frac{-d_1^2}{2}} > 0. \tag{2.4}$$

Gamma indicates when a position in the market needs to be re-hedged, ensuring that a Delta neutral position is maintained. The limitation is that in a shifting market, continual re-hedging of a Delta hedge is required to maintain a position. Gamma has uses for those in the Quant industry as it declares how much Delta will change given changes in the underlying asset. If one considers the transactional costs of re-hedging, a smaller Gamma would indicate that little cost is associated with this action, making solutions such as dynamic hedging plausible. One can summarise by stating that as an option becomes At-The-Money the more Gamma increases given the great variation on the position.

Theta (θ) is the rate of change of the option price with time. Glendall (2014) [14] states that "An option is more valuable the longer it is valid"; this is logical as it provides the holder with a higher probability of an option landing In-The-Money. However, Theta measures a (long) options time decay such that the closer the option gets to expiry, the more the option price 'decays'. Therefore the change in the options price with respect to a one day decrease until expiry is calculated by the closed-form:

$$\theta_{call} = -\frac{Sn(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}\mathcal{N}(d_2). \tag{2.5}$$

Vega ($\mathcal{V}$) is arguably the most important Greek as it is a measure of the rate of change of an options price per percentage change in the implied volatility of a given underlying. Traders use Vega to identify the sensitivity of a portfolio to changes in implied volatility and can indicate if an option should be longed or shorted. For example, a higher Vega (assuming all parameters are constant) would indicate that an option should be sold, as the increase in volatility would subsequently increase the options price. This is partly due to an options increased probability of expiring In-The-Money. The closed-form

solution is given by:

$$\mathcal{V}_{call/put} = \frac{S}{\sqrt{2\pi}} e^{-\frac{-d_1^2}{2}} \sqrt{T}. \tag{2.6}$$

Lastly, Rho (ρ) is the sensitivity an option price with respect to changes in the risk-free interest rate and measures the change in the price of an option given a percentage change in the interest rate. Therefore indicating whether a trader would expect the option price to rise and fall should the risk-free rate change. The closed-form (call) solution is given by the following:

$$\rho_{call} = TKe^{-rT}\mathcal{N}(d_2) > 0. \tag{2.7}$$

2.3 Cox-Ingersoll-Ross (CIR) Process

Single-factor models are processes that assume that a specified factor, such as the interest rate can summarise the term structures behaviour at any point in time. The general form of a single factor model consists of a time-dependent drift term and some stochastic term that attempts to recreate the random nature of the state variable.

The interest rates term structure is the relation between the term to maturity and the interest (or bond yields). The graphical representation of the term structure is classically known as the yield curve and represents the market's attitude towards future events. Modelling the term structure theoretically provides one with the ability to anticipate the sensitivity of the yield curve to changes in variables that affect the term structure.

Cox, Ingersoll and Ross understood that modelling the term structure was of high importance and developed the CIR model in 1985 [8, pg.390-91] under the assumptions that:

- Changes are described by a single state variable.
- The development of the state variable is governed by a stochastic differential equation.
- The rate of return has mean and variances proportional to the state variable.

Giving rise to the (later known) CIR model:

$$dr(t) = \kappa(\Theta - r(t)) + \sigma\sqrt{r(t)}dw(t). \tag{2.8}$$

The stochastic differential equation adheres to Brownian motion $w(t)$ (under risk neutrality) and possesses the important quality in that there exists no negative interest rates. In addition, Θ, σ and κ remain non-negative via the Feller condition which will be expanded on in later chapters.

Subsequent chapters will focus on the application of the CIR process in the pricing of zero-coupon bonds. Where at maturity a payoff of 1 is returned which translates to the initial condition of the zero-coupon bond pricing PDE, see :

$$\frac{\partial B}{\partial t} + \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial r^2} + (\kappa(\Theta - r) - \lambda\sigma)\frac{\partial B}{\partial r} - rB = 0. \tag{2.9}$$

Where $\lambda\sigma = \Theta - r$ and the boundary condition is $B(T, T) = 1$.

For this thesis 2.9 will be expressed as the following, where $a = \Theta\kappa$ and $b = \kappa$:.

$$\frac{\partial B}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial r^2} + (a - br)\frac{\partial B}{\partial r} - rB. \tag{2.10}$$

Which follows the fundamental bond pricing equation (see Wilmott (2007) [28, pg.362-63] for the bond pricing PDE derivation) and holds similar form to general d/c/r equations. which will be discussed further in later chapters. Similarly to 2.1, the PDE has an exact solution under no-arbitrage arguments at time t given some expiry time: T (Cox et. al (1985) [8, pg.393]):

$$\begin{aligned} B(r, t, T) &= a(t, T)e^{-b(t,T)r}, \\ a(t, T) &= \left(\frac{2\gamma e^{(\gamma+\kappa)(T-t)\frac{1}{2}}}{2\gamma + (\gamma+\kappa)(e^{\gamma(T-t)} - 1)} \right)^{\frac{2\kappa\Theta}{\sigma^2}}, \\ b(t, T) &= \frac{2(e^{\gamma(T-t)} - 1)}{2\gamma + (\gamma+\kappa)(e^{\gamma(T-t)} - 1)}. \\ \gamma &= \sqrt{\kappa^2 + 2\sigma^2}. \end{aligned} \tag{2.11}$$

2.4 CIR Bond Sensitivities

When considering the bond market, investors will identify the expected performance of the bond should interest rates change; allowing them to identify the risks associated with purchasing a bond. In the case of this thesis, the following sensitivities are expanded upon Duration, Complexity, the expiry sensitivity, speed of adjustment sensitivity and the volatility sensitivity.

Duration (denoted $\mathcal{D}$) is simply a measurement of a bond's sensitivity to a change in the interest rate and is dependent on the bonds term to maturity, yield and the coupon rate. In the words of Choudhry (2005 [7, pg.32]), "The average time until receipt of a bond's cash flows, weighted according to the present values of these cash flows, measured in years, is known as Duration". In general, a bond that has a higher calculated Duration will indeed have greater risks associated with interest rate changes. The bond is said to have a higher sensitivity in this regard. Duration is known in two forms:

- Macaulay Duration.

- Modified Duration.

The prior is defined as the weighted average term to maturity of bond cash flows. The later is the inverse changes in bond prices as a result of small changes in interest rates. This thesis will focus on the modified definition of Duration.

An important note is that this thesis uses the zero-coupon bond pricing PDE. Therefore, no coupon payments over the bonds lifetime result in a Duration that is equivalent to the bond's maturity (see Webber and James (2000 [26, pg.116]) proof). Similar to the Black-Scholes Greeks, the establishment of the closed-form bond equation 2.11 gives rise to the following closed-form solution:

$$\mathcal{D} = \frac{\partial B}{\partial r} = -a(t,T)b(t,T)e^{-b(t,T)r(t)}. \tag{2.12}$$

Duration is, by definition, a linear relationship between the change in yield and the change in bond price. Therefore if the Duration is approximated at a singular point, then a flaw in Duration is encountered. If there exists any deviation from that point inaccuracies are introduced, resulting in either over or under-estimations of the sensitivity

Convexity (denoted $\mathcal{C}$) alleviates this issue by being a second-order measurement on the interest rate risk of a bond. Similarly to Greek Gamma (2.4), it identifies how quickly a bond price is likely to change given any interest rate change. It holds the advantage that the error associated with the Duration is adjusted for by measuring the curvature on the yield curve. Choudhry (2005 [7, pg.44-45]) provides an excellent summary of the bonds price regarding convexity, where a positive convexity would indicate an increased Duration and fallen yield. Therefore the bond would subsequently experience a price increase.

Convexity is, by definition, the second derivative of the bond price with respect to the interest rate. Consider the closed-form solution of convexity:

$$\mathcal{C} = \frac{\partial^2 B}{\partial r^2} = a(t,T)b(t,T)^2 e^{-b(t,T)r(t)}. \tag{2.13}$$

The remaining sensitivities have little to no current literature. This stems from the 'usefulness' of such sensitivities and the difficulty in calculating them. Beginning with the sensitivity of a bond to its maturity, the following closed-form solution is calculated by differentiating 2.11 with respect to T when $t = 0$:

$$\frac{\partial B}{\partial T} = \frac{\partial a(0,T)}{\partial T} e^{-b(0,T)r(0)} - a(0,T)\frac{\partial b(0,T)}{\partial T} r(0) e^{-b(0,T)r(0)}. \tag{2.14}$$

Where:

$$\frac{\partial a(0,T)}{\partial T} = -\frac{\kappa\,\Theta\,2^{\frac{2\kappa\Theta}{\sigma^2}}(\gamma+\kappa)(\gamma-\kappa)(e^{\gamma T}-1)\left(e^{\frac{\kappa\,\Theta(\gamma+\kappa)T}{\sigma^2}}\right)\left(\frac{\gamma}{(\gamma+\kappa)(e^{\gamma T}-1)+2\gamma}\right)^{\frac{2\kappa\Theta}{\sigma^2}}}{\sigma^2((\gamma+\kappa)e^{\gamma T}+\gamma-\kappa)}.$$

$$\frac{\partial b(0,T)}{\partial T} = \frac{4\gamma^2 e^{\gamma T}}{((\gamma+\kappa)e^{\gamma T}+\gamma-\kappa)^2}.$$

The sensitivity associated with the speed of adjustment (κ) has a closed-form representation which is as follows:

$$\frac{\partial B}{\partial \kappa} = \frac{\partial a(t,T,\kappa)}{\partial \kappa} e^{-b(t,T,\kappa)r(t)} - a(t,T,\kappa)\frac{\partial b(t,T,\kappa)}{\partial \kappa} r(t) e^{-b(t,T,\kappa)r(t)}. \quad (2.15)$$

Where $\frac{\partial a(t,T,\kappa)}{\partial \kappa}$ and $\frac{\partial b(t,T,\kappa)}{\partial \kappa}$ are given in Appendix B.

The final sensitivity is the sensitivity of the bond with respect to its volatility. Consider the following closed-form:

$$\frac{\partial B}{\partial \sigma} = \frac{\partial a(t,T,\sigma)}{\partial \sigma} e^{-b(t,T,\sigma)r(t)} - a(t,T,\sigma)\frac{\partial b(t,T,\sigma)}{\partial \sigma} r(t) e^{-b(t,T,\sigma)r(t)}. \quad (2.16)$$

Where $\frac{\partial a(t,T,\sigma)}{\partial \sigma}$ and $\frac{\partial b(t,T,\sigma)}{\partial \sigma}$ are given in Appendix B.

A possible explanation for the lack of literature lends itself to the substantial equations associated with the closed-form solutions, making them extremely impractical for coding purposes. However, for the sake of consistency, each of the above closed-forms will be used in later chapters.

3 PDE Finite Difference Methods

Given the analytical formulae of the Greeks and bond sensitivities, one may ask if these closed-form solutions can always be found? In short, no. Closed-form solutions may not exist or be impractical to implement; therefore, finite difference methods (FDMs) and methods of a similar nature are introduced to approximate such solutions. The following section applies finite difference methods to linear one-factor d/c/r equations to price financial derivatives and instruments.

3.1 Motivation

Consider equation 2.1, the Black-Scholes PDE contains one space variable (S) and one time variable (t). Given that most d/c/r equations are defined on a semi-infinite ($0 < S < \infty,\ t > 0$) or infinite domain ($-\infty < S < \infty,\ t > 0$), there exists an infinite number of possible solutions. Finite difference methods restrict this domain to ensure a unique solution exists by imposing constraints, classically known as the boundary conditions and the initial condition. Consider the following types:

- Dirichlet - Solution is identified on the boundary.
- Neumann - Directional Derivatives are given on the boundary.

The remaining types include Robin, Mixed and Cauchy: that are not covered in this thesis.

Consider the generalised linear one factor d/c/r equation, where the boundary conditions are defined on some bounded domain $[A, B]$ for $t \in (0, T)$, where $A < B$:

$$\begin{cases} -\frac{\partial V}{\partial t} + \alpha(x,t)\frac{\partial^2 V}{\partial x^2} + \beta(x,t)\frac{\partial V}{\partial x} + \gamma(x,t)V = 0, & 0 < x < \infty,\ t > 0 \\ V(x,0) = f(x), \quad 0 < x < \infty. & \text{Initial Condition.} \\ V(A,t) = \zeta_A(t),\ V(B,t) = \zeta_B(t). & \text{Boundary Condition.} \end{cases}$$

It is clear from the above that three unknowns $f(x)$, $\zeta_A(t)$ and $\zeta_B(t)$ exist. The initial condition is defined based on the PDEs function and is usually

related to a payoff function such as an option or bond payoff. The boundary conditions require a transformation to a bounded domain $[A, B]$ using the following methods: Domain Truncation, Domain Transformation (see Duffy (2009) [11, pg.5-7]), the PDE Conservative Form and Log Transformation. However, only the truncation and transformation methods are applied within this thesis.

Domain truncation is defined as transforming the domain of a PDE to that of a bounded domain (i.e. [A, B]). However, there exists much uncertainty on how this can be accomplished in the literature. Duffy (2009) [11, pg.4] highlights the issue with domain truncation and suggests an alternative method known as domain transformation.

3.1.1 Domain Transformation

Domain transformation resolves the issues associated with domain truncation by identifying the far-field boundary condition. Duffy (2009) [11] focuses on taking the semi-infinite domain and transforming it to that of a unit interval [0,1]. He additionally introduces five transformations [11, pg.5-6], however, the following transformation is used throughout this thesis.

$$y = \frac{x}{x+\phi}, \qquad \text{Where } \phi \text{ is the user defined free-scale factor.} \tag{3.1}$$

Transformation 3.1 is inserted into a one-factor PDE replacing the state variable defined on the semi-infinite domain with a new state variable (y) defined on the unit interval. The transformed PDE requires an application of Fichera theory to identify whether new boundary conditions are required in most cases.

3.1.2 Fichera Theory

Fichera Theory was introduced by G.Fichera in 1960 and was subsequently developed in 1973 by Radkevic and Olejnik [4, pg.1]. Such theory finds use in the field of partial differential equations where PDEs degenerate on the boundary of some ($\Omega \subset \mathbb{R}^n$) bounded space domain.

Following along the lines of Buckova et.al (2014 [4, pg.2]), Lu (2014 [17, pg.21]) and Duffy (2009 [11, pg.7-9]) the classical introduction is via the consideration of the second order elliptical equation:

$$LV = \sum_{i,j=1}^{n} \alpha_{i,j} \frac{\partial^2 V}{\partial x_i \partial x_j} + \sum_{i=1}^{n} \beta_i \frac{\partial V}{\partial x_i} + \gamma V = f; \qquad x \in \Omega.$$

Given the condition:

$$\left\{ \sum_{i,j=1}^{n} a_{i,j}\xi_i\xi_j \geq 0, \quad \forall\, \xi \in \mathbb{R}^n \right\} = \eta \tag{3.2}$$

Consider the bounded domain Ω, let Σ be a piece-wise smooth boundary such that $\Omega \cup \Sigma$. Σ splits into a hyperbolic and parabolic subset. The hyperbolic subset corresponds to the case in which 3.2 is equal to zero. Consider the Fichera function:

$$\mathcal{F} = \sum_{i=1}^{n} \Big(\beta_i - \sum_{k=1}^{n} \frac{\partial \alpha_{i,k}}{\partial x_k}\Big) v_i, \quad | \quad \eta \to \mathbb{R}. \tag{3.3}$$

Where v_i is the ith directional cosine component of the inner normal vector at the boundary Ω.

Considering equation 3.3, there exists three subsets of the hyperbolic boundary known as the tangent flow, outflow and inflow [4, pg.2]. Consider:

$$\begin{cases} \Sigma_0 = \{\mathcal{F} = 0 | \mathcal{F}(x \in \Sigma)\} \\ \Sigma_+ = \{\mathcal{F} > 0 | \mathcal{F}(x \in \Sigma)\}. \\ \Sigma_- = \{\mathcal{F} < 0 | \mathcal{F}(x \in \Sigma)\} \end{cases}$$

Radkevic and Olejnik (1973 [20, pg.18]) demonstrated in *lemma 1.1.1* that at the single points of the hyperbolic boundary, the sign of the Fichera function does not change given smooth non-degenerate changes in the elliptic equations state variables. Therefore, in practice, Fichera theory can be applied to some PDE for the application of appropriate boundary conditions if any are required.

3.1.3 Finite Difference Grid (Mesh)

Upon identifying the initial condition and boundary conditions of the PDE, discretisation takes place through the introduction of the finite difference grid (or mesh) with equal time steps between nodes. In the generalised linear one factor PDE case, the domain/plane of (x, t) is converted into a discretised set of nodes:

$$V_j^n = (jh, nk), \quad \text{where:} \quad k = \frac{T}{N}, \quad h = \frac{B}{J}. \tag{3.4}$$

Such that: $j = 1, 2, \cdots, J; \quad n = 1, 2, \cdots, N.$

The above forms an FDM mesh containing a space and time interval.

3.2 Crank Nicolson FDM Method

The Crank Nicolson (CN) method is prevalent in the approximation of one-factor d/c/r equations (e.g. Black-Scholes), reasons stem from its stability as a second-order scheme and the relative ease of implementing the method into code.

The advantage of using the CN method becomes apparent by observing the classic Implicit and Explicit FDMs. The Explicit method has a fundamental flaw as it must satisfy a stability condition (see Wilmott et.al (1995) [29, pg.140-42]), otherwise, it does not converge to the exact solution (conditionally stable), yet the method requires no tridiagonal matrix solver to approximate derivatives. The fully implicit scheme does not suffer from the instability issues of the explicit method (unconditionally stable) but requires the solution of a tridiagonal matrix. Both methods have a truncation error of order one, making them only first-order accurate.

The CN method is a variation on the fully implicit scheme in which the average between nodes in the space domain is taken, the method is implicit and requires the solving of a tridiagonal system, yet is an unconditionally stable second-order scheme.

3.2.1 Generalised Crank Nicolson

Consider the following generalised derivation of the Crank Nicolson method for the following linear one factor PDE:

$$\frac{\partial V}{\partial t} = \alpha(x,t)\frac{\partial^2 V}{\partial x^2} + \beta(x,t)\frac{\partial V}{\partial x} + \gamma(x,t)V. \tag{3.5}$$

Now respectively apply the implicit and explicit methods to 3.5 in order to obtain the following discretised PDEs (see Wilmott et.al (1995) [29, pg.139-47] for an explanation on the explicit/implicit methods):

$$\frac{V_j^{n+1}-V_j^n}{k} = \alpha(x,t)\left(\frac{V_{j+1}^n-2V_j^n+V_{j-1}^n}{h^2}\right) + \beta(x,t)\left(\frac{V_{j+1}^n-V_{j-1}^n}{2h}\right) + \gamma(x,t)V_j^n.$$

$$\frac{V_j^{n+1}-V_j^n}{k} = \alpha(x,t)\left(\frac{V_{j+1}^{n+1}-2V_j^{n+1}+V_{j-1}^{n+1}}{h^2}\right) + \beta(x,t)\left(\frac{V_{j+1}^{n+1}-V_{j-1}^{n+1}}{2h}\right) + \gamma(x,t)V_j^{n+1}.$$

The derivation of Crank Nicolson is given by the addition and subsequent averaging of the above discretisations:

$$V_j^{n+\frac{1}{2}} = \frac{1}{2}(V_j^n + V_j^{n+1}).$$

Therefore:

$$\frac{1}{2}\left(2\left(\frac{V_j^{n+1}-V_j^n}{k}\right)\right) = \frac{1}{2}\left(\alpha(x,t)\left(\frac{V_{j+1}^n-2V_j^n+V_{j-1}^n+V_{j+1}^{n+1}-2V_j^{n+1}+V_{j-1}^{n+1}}{h^2}\right)\right)$$
$$+ \frac{1}{2}\left(\beta(x,t)\left(\frac{V_{j+1}^n-V_{j-1}^n+V_{j+1}^{n+1}-V_{j-1}^{n+1}}{2h}\right)\right)$$
$$+ \left(\gamma(x,t)\left(\frac{V_j^n+V_j^{n+1}}{2}\right)\right).$$

$$\left(\frac{V_j^{n+1}-V_j^n}{k}\right) = \alpha(x,t)\left(\frac{V_{j+1}^n-2V_j^n+V_{j-1}^n+V_{j+1}^{n+1}-2V_j^{n+1}+V_{j-1}^{n+1}}{2h^2}\right)$$
$$+ \beta(x,t)\left(\frac{V_{j+1}^n-V_{j-1}^n+V_{j+1}^{n+1}-V_{j-1}^{n+1}}{4h}\right)+\gamma(x,t)\left(\frac{V_j^n+V_j^{n+1}}{2}\right).$$

Let $\alpha = \alpha(x,t)$; $\beta = \beta(x,t)$ and $\gamma = \gamma(x,t)$. Moving the known terms on the right-hand side and multiplying both sides by h^2 and k yields:

$$V_{j-1}^{n+1}\left(-\frac{\alpha k}{2}+\frac{\beta kh}{4}\right)+V_j^{n+1}\left(h^2+\alpha k-\frac{\gamma h^2 k}{2}\right)+V_{j+1}^{n+1}\left(-\frac{\alpha k}{2}-\frac{\beta kh}{4}\right)=$$
$$V_{j-1}^{n}\left(\frac{\alpha k}{2}-\frac{\beta kh}{4}\right)+V_j^{n}\left(h^2-\alpha k+\frac{\gamma h^2 k}{2}\right)+V_{j+1}^{n}\left(\frac{\alpha k}{2}+\frac{\beta kh}{4}\right).$$

The derivation is concluded with the generalised Crank Nicolson method:

$$\begin{cases} aV_{j-1}^{n+1}+bV_j^{n+1}+cV_{j+1}^{n+1}=dV_{j-1}^{n+1}+eV_j^{n+1}+fV_{j+1}^{n+1}, \\ a=-\frac{\alpha k}{2}+\frac{\beta kh}{4}, \qquad d=\frac{\alpha k}{2}-\frac{\beta kh}{4}, \\ b=h^2+\alpha k-\frac{\gamma h^2 k}{2}, \qquad e=h^2-\alpha k+\frac{\gamma h^2 k}{2}, \\ c=-\frac{\alpha k}{2}-\frac{\beta kh}{4}, \qquad f=\frac{\alpha k}{2}+\frac{\beta kh}{4}. \end{cases} \tag{3.6}$$

3.2.2 Crank Nicolson Black-Scholes Implementation

Considering the single factor Black-Scholes model 2.1 and the generalised implementation of the Crank Nicolson method 3.6. The Crank Nicolson method can be applied by considering the diffusion, convection and reaction components of the Black-Scholes model and inserting them into α, β and γ which correspond to:

- Diffusion Term (α): $\frac{1}{2}\sigma^2 S^2$.

- Convection Term (β): rS.
- Reaction Term (γ): $-r$.

Substituting the above terms into 3.6 yields the following representation:

$$\begin{cases} aV_{j-1}^{n+1} + bV_j^{n+1} + cV_{j+1}^{n+1} = dV_{j-1}^{n+1} + eV_j^{n+1} + fV_{j+1}^{n+1}, \\ a = -\frac{\sigma^2 S^2 k}{4} + \frac{rSkh}{4}, \qquad d = -\frac{\sigma^2 S^2 k}{2} - \frac{rSkh}{4}, \\ b = h^2 + \frac{\sigma^2 S^2 k}{2} - \frac{rh^2 k}{2}, \qquad e = h^2 - \frac{\sigma^2 S^2 k}{2} - \frac{rh^2 k}{2}, \\ c = -\frac{\sigma^2 S^2 k}{4} - \frac{rSkh}{4}, \qquad f = \frac{\sigma^2 S^2 k}{4} + \frac{rSkh}{4}. \end{cases} \tag{3.7}$$

Which hold under domain truncation for the following:

- Initial Conditions:
  $V_{call}(S,T) = \max(S-K,0)$; $V_{put}(S,T) = \max(K-S,0)$.
- Boundary Conditions:
  Call Option: $V(0,t) = 0$; $V(S,t) = 3K$.
  Put Option: $V(0,t) = Ke^{-rt}$; $V(S,t) = 0$.

Where the call options far-field boundary condition is a multiple of the options strike price K (see Duffy (2018) [10, pg.647]).

3.2.3 Crank Nicolson CIR PDE Implementation

Consider the following domain transformation using 3.1 as outlined by Duffy (2009) where the free scale factor ϕ is set equal to 1 (see: [17, pg.17-18] for a full derivation):

$$y = \frac{r}{r+1}; \quad r = \frac{y}{1-y}; \quad \frac{\partial y}{\partial r} = (1-y)^2; \quad \frac{\partial^2 y}{\partial r^2} = -2(y-1)^3. \tag{3.8}$$

Applying the chain rule to obtain the bond price for the first and second derivatives with respect to the short rate and substituting them into 2.10 yields:

$$\begin{aligned}
\frac{\partial B}{\partial t} &= \frac{1}{2}\sigma^2 r\frac{\partial^2 B}{\partial r^2} + (a-br)\frac{\partial B}{\partial r} - rB, \\
&= \frac{1}{2}\sigma^2\Big(\frac{y}{1-y}\Big)\Big(-2(1-y)^3\frac{\partial B}{\partial y} + (1-y)^4\frac{\partial^2 B}{\partial y^2}\Big) \\
&+ \Big(a - b\Big(\frac{y}{1-y}\Big)\Big)(1-y)^2\frac{\partial B}{\partial y} - \Big(\frac{y}{1-y}\Big)B, \\
&= \frac{1}{2}\sigma^2 y(1-y)^3\frac{\partial^2 B}{\partial y^2} + (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)\frac{\partial B}{\partial y} - \Big(\frac{y}{1-y}\Big)B.
\end{aligned} \tag{3.9}$$

Given the domain transformation, the interval in which the mesh is defined alters ([17, pg.19]) bringing to attention the existence of the denominator $(1-y)$ which in consideration of the interval $(0,1)$ causes issues as 3.9 is undefined at this point. A proposition is made to restrict the interval to $[0, \frac{J}{J+1}]$ which alleviates the issue. In addition to the change of interval, Fichera theory must be applied to 3.9 to obtain the required boundary conditions. Consider the Fichera function in 3.3 and apply it to the transformed PDE:

$$\mathcal{F} = \Big(\beta(y,t) - \frac{1}{2}\sigma^2(1-y)^3 + \frac{3}{2}\sigma^2 y(1-y)^2\Big)\Big)v(y) \tag{3.10}$$

Where $\beta(y,t) = (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)$.

Note that the direction cosine has two cases which need to be identified: $v(y=0) = 1$ and $v(y=1) = -1$.

$$\begin{aligned}
&\mathcal{F} = (0)(-1) = 0; \quad \text{No Boundary condition is required at } y = 1. \\
&\mathcal{F} = (a - \frac{1}{2}\sigma^2)(1).
\end{aligned}$$

Therefore when $y = 0$, the hyperbolic boundary depends on the following tangent flow, outflow and inflow subsets:

$$\begin{cases}
\mathcal{F} = 0 & | \quad \sigma^2 = 2a. \quad \text{No Boundary Condition Required.} \\
\mathcal{F} > 0 & | \quad \sigma^2 > 2a. \quad \text{No Boundary Condition Required.} \\
\mathcal{F} < 0 & | \quad \sigma^2 < 2a. \quad \text{Boundary Condition Required.}
\end{cases}$$

The third case requires a boundary condition. Therefore by letting $y = 0$ in 3.9 the following first order hyperbolic equation is obtained:

$$\frac{\partial B}{\partial t} = a\frac{\partial B}{\partial y}. \tag{3.11}$$

Lu (2014) [17, pg.23] implements the Thomée scheme to approximate the near-field boundary condition; the same method is applied to both the Crank Nicolson and Alternating Direction Explicit methods. The initial condition and far-field conditions of the transformed CIR PDE are the following:

- Initial Conditions:
  $B(y, T) = 1$.
- Far-field Boundary Condition:
  $B(\frac{J}{J+1}, t) = 0$.

Applying the diffusion, convection and reaction terms in 3.9 to the generalised Crank Nicolson derivation (3.6) yields the implementation where y is denoted by the steps j $(= 1, 2, \cdots, J)$ in the space domain:

$$\begin{cases} aB_{j-1}^{n+1} + bB_j^{n+1} + cB_{j+1}^{n+1} = dB_{j-1}^n + eB_j^n + fB_{j+1}^n, \\ a = -\frac{\sigma^2 y(1-y)^3 k}{4} + \frac{(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)kh}{4}, \\ b = h^2 + \frac{1}{2}\sigma^2 y(1-y)^3 k + \frac{yh^2 k}{2(1-y)}, \\ c = -\frac{\sigma^2 y(1-y)^3 k}{4} - \frac{(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)kh}{4}, \\ d = \frac{\sigma^2 y(1-y)^3 k}{4} - \frac{(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)kh}{4}, \\ e = h^2 - \frac{1}{2}\sigma^2 y(1-y)^3 k - \frac{yh^2 k}{2(1-y)}, \\ f = \frac{\sigma^2 y(1-y)^3 k}{4} + \frac{(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)kh}{4}. \end{cases} \tag{3.12}$$

3.2.4 Thomée Scheme Crank Nicolson

Consider the following application of the Thomée scheme, paralleling with Lu (2014, [17]):

$$\frac{B_{j+0.5}^{n+1} - B_{j+0.5}^n}{k} = a\frac{B_{j+1}^{n+0.5} - B_j^{n+0.5}}{h}.$$

Setting $\lambda = \frac{ak}{h}$.

$$B_{j+0.5}^{n+1} - B_{j+0.5}^n - \frac{ak}{h}\left(B_{j+1}^{n+0.5} - B_j^{n+0.5}\right) = 0.$$

$$B_{j+1}^{n+1} + B_j^{n+1} - B_{j+1}^n - B_j^n = \lambda\left(B_{j+1}^{n+1} + B_{j+1}^n - B_j^{n+1} - B_j^n\right).$$

Rearranging such that the known values are on the right-hand side gives:

$$B_j^{n+1}(1+\lambda) + B_{j+1}^{n+1}(1-\lambda) = B_j^n(1-\lambda) + B_{j+1}^n(1+\lambda). \tag{3.13}$$

The application into the Crank Nicolson method can be represented by the following where one additional unknown is included in the system of equations.

$$
\begin{bmatrix}
1+\lambda & 1-\lambda & & & & \\
A_1 & B_1 & C_1 & & & \\
 & A_2 & B_2 & C_2 & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & A_{J-1} & B_{J-1} & C_{J-1} \\
 & & & & & 1
\end{bmatrix}
\begin{bmatrix}
B_0^{n+1} \\ B_1^{n+1} \\ B_2^{n+1} \\ \vdots \\ B_{J-1}^{n+1} \\ B_J^{n+1}
\end{bmatrix}
=
\begin{bmatrix}
(1-\lambda)B_0^n + (1+\lambda)B_1^n \\ F_1 \\ F_2 \\ \vdots \\ F_{J-1} \\ F_J
\end{bmatrix}
$$

FIGURE 3.1: Implementation of the Thomée Scheme into the Crank Nicolson method.

3.3 Alternating Direction Explicit

Alternating Direction Explicit (ADE) is a relatively unknown numerical method first applied to financial computation for non-linear and linear PDEs in 2009 by D. Duffy [11, pg.9-15]. The literature over the past decade has grown, leading to many variations of the ADE scheme. Variations are based on a modification of the difference quotient used to approximate the convection term in d/c/r PDEs.

The method consists of an upward and downward sweep (explicit in nature) that individually compute the solution at a given node inside a computational grid. The solution is subsequently averaged out resulting in an ADE approximation of the node. Buĉková et.al (2010) [13, pg.311] provides an excellent visual aid to the function of the upward and downward sweep:

Upward sweep.

Downward sweep.

FIGURE 3.2: Buĉková's ADE upward and downward sweep figure [5, pg.311]

Concerning linear one factor PDEs, it is proven by Ehrhardt et. al (2015) in [5, pg.314] that the ADE scheme was unconditionally stable. Additionally, this

thesis focuses on the Barakat and Clark ADE variant given by Duffy (2018) [10, pg.672] which simultaneously performs a backward and forward sweep. The advantage lies with the truncation error of $O[k^2, h^2]$ which comes about from the sweeping cancellation effect (see Buckova et.al (2015) [5, pg.323]) making the ADE variant a competitor to that of the Crank Nicolson method.

3.3.1 Generalised Alternating Direction Explicit

The proceeding difference quotients give the time derivative, diffusion term and reaction terms of 3.5:

Upwind Difference Quotients ($j = 1, \cdots, J-1$):

$$\frac{\partial U}{\partial t} = \frac{U_j^{n+1} - U_j^n}{k}; \qquad \frac{\partial^2 U}{\partial x^2} = \frac{U_{j+1}^n - U_j^n - U_j^{n+1} + U_{j-1}^{n+1}}{h^2}; \qquad U = U_j^{n+1}.$$

Downwind Difference Quotients ($j = J-1, \cdots, 1$):

$$\frac{\partial D}{\partial t} = \frac{D_j^{n+1} - D_j^n}{k}; \qquad \frac{\partial^2 D}{\partial x^2} = \frac{D_{j+1}^{n+1} - D_j^{n+1} - D_j^n + D_{j-1}^n}{h^2}; \qquad D = D_j^{n+1}.$$

A range of modified convection difference quotients can be applied within the derivation, however, for simplicity the classical Towler and Yang (TY) difference quotients (Towler and Yang, 1978) [24, pg.46] are implemented (for other difference quotients the reader is referred to: Buckova (2015) [5, pg.312-313] and Lu (2014) [17, pg.12-13]):

Upwind TY Quotients,

$$\frac{\partial U}{\partial x} = \frac{U_{j+1}^n - U_{j-1}^{n+1}}{2h};$$

Downwind TY Quotients

$$\frac{\partial D}{\partial x} = \frac{D_{j+1}^{n+1} - D_{j-1}^n}{2h}.$$

The upwind difference quotients are substituted into 3.5 to yield:

$$\frac{U_j^{n+1} - U_j^n}{k} = \alpha(x,t)\frac{U_{j+1}^n - U_j^n - U_j^{n+1} + U_{j-1}^{n+1}}{h^2} + \beta(x,t)\frac{U_{j+1}^n - U_{j-1}^{n+1}}{2h} + \gamma(x,t)U_j^{n+1}.$$

In the downwind difference case:

$$\frac{D_j^{n+1} - D_j^n}{k} = \alpha(x,t)\frac{D_{j+1}^n - D_j^n - D_j^{n+1} + D_{j-1}^{n+1}}{h^2} + \beta(x,t)\frac{D_{j+1}^n - D_{j-1}^{n+1}}{2h} + \gamma(x,t)D_j^{n+1}.$$

Taking both cases (dropping notation), multiplying by the time step k and rearranging such that the known values are on the right-hand side gives:

$$U_{j-1}^{n+1}\Big(-\frac{\alpha K}{h^2}+\frac{\beta K}{2h}\Big)+U_j^{n+1}\Big(1+\frac{\alpha K}{h^2}-\gamma K\Big)=U_j^n\Big(1-\frac{\alpha k}{h^2}\Big)+U_{j+1}^n\Big(\frac{\alpha K}{h^2}+\frac{\beta K}{2h}\Big),$$

$$D_{j+1}^{n+1}\Big(-\frac{\alpha K}{h^2}-\frac{\beta K}{2h}\Big)+D_j^{n+1}\Big(1+\frac{\alpha K}{h^2}-\gamma K\Big)=D_j^n\Big(1-\frac{\alpha k}{h^2}\Big)+D_{j-1}^n\Big(\frac{\alpha K}{h^2}-\frac{\beta K}{2h}\Big).$$

Rearranging the above terms:

$$U_j^{n+1}\Big(1+\frac{\alpha K}{h^2}-\gamma K\Big)=U_j^n\Big(1-\frac{\alpha k}{h^2}\Big)+U_{j+1}^n\Big(\frac{\alpha K}{h^2}+\frac{\beta K}{2h}\Big)+U_{j-1}^{n+1}\Big(\frac{\alpha K}{h^2}-\frac{\beta K}{2h}\Big),$$

$$D_j^{n+1}\Big(1+\frac{\alpha K}{h^2}-\gamma K\Big)=D_j^n\Big(1-\frac{\alpha k}{h^2}\Big)+D_{j-1}^n\Big(\frac{\alpha K}{h^2}-\frac{\beta K}{2h}\Big)+D_{j+1}^{n+1}\Big(\frac{\alpha K}{h^2}+\frac{\beta K}{2h}\Big).$$

The ADE approximation is thus obtained by averaging out the upwind and downwind equations:

$$V_j^{n+1}=\frac{1}{2}\Big(U_j^{n+1}+D_j^{n+1}\Big).$$

3.3.2 B&C ADE Black-Scholes Implementation

The implementation of ADE into Black-Scholes PDE (2.1) follows the same process as the Crank Nicolson implementation. The diffusion, convection and reaction terms are substituted into the upwind and downwind equations, then subsequently averaged resulting in the final solution. For clarity the upwind and downwind equations containing the Black-Scholes d/c/r terms are as follows:

$$U_j^{n+1}=\frac{1}{1+\frac{\sigma^2S^2k}{2h^2}+rK}\left(U_{j-1}^{n+1}\Big(\frac{\sigma^2S^2k}{2h^2}-\frac{rSk}{2h}\Big)+U_j^n\Big(1-\frac{\sigma^2S^2k}{2h^2}\Big)+U_{j+1}^n\Big(\frac{\sigma^2S^2k}{2h^2}+\frac{rSk}{2h}\Big)\right).$$

$$D_j^{n+1}=\frac{1}{1+\frac{\sigma^2S^2k}{2h^2}+rK}\left(D_{j+1}^{n+1}\Big(\frac{\sigma^2S^2k}{2h^2}+\frac{rSk}{2h}\Big)+D_j^n\Big(1-\frac{\sigma^2S^2k}{2h^2}\Big)+D_{j-1}^n\Big(\frac{\sigma^2S^2k}{2h^2}-\frac{rSk}{2h}\Big)\right).$$

3.3.3 B&C ADE CIR PDE Implementation

Inputting the terms in 3.9 from the transformed PDE into the ADE generalisation results in the following upwind and downwind equations:

$$\begin{aligned}U_j^{n+1} &= \frac{1}{1+\frac{\sigma^2Ky(1-y)^3}{2h^2}+\frac{yK}{1-y}}\left(U_{j-1}^{n+1}\left(\frac{\sigma^2Ky(1-y)^3}{2h^2}-\frac{(a(1-y)^2-\sigma^2y(1-y)^2-by(1-y))K}{2h}\right)\right.\\ &+ \left.U_j^n\left(1-\frac{\sigma^2Ky(1-y)^3}{2h^2}\right)+U_{j+1}^n\left(\frac{\sigma^2Ky(1-y)^3}{2h^2}+\frac{(a(1-y)^2-\sigma^2y(1-y)^2-by(1-y))K}{2h}\right)\right).\end{aligned}$$

$$\begin{aligned}D_j^{n+1} &= \frac{1}{1+\frac{\sigma^2Ky(1-y)^3}{2h^2}+\frac{yK}{1-y}}\left(D_{j+1}^{n+1}\left(\frac{\sigma^2S^2k}{2h^2}+\frac{(a(1-y)^2-\sigma^2y(1-y)^2-by(1-y))K}{2h}\right)\right.\\ &+ \left.D_j^n\left(1-\frac{\sigma^2Ky(1-y)^3}{2h^2}\right)+D_{j-1}^n\left(\frac{\sigma^2Ky(1-y)^3}{2h^2}-\frac{(a(1-y)^2-\sigma^2y(1-y)^2-by(1-y))K}{2h}\right)\right).\end{aligned}$$

3.3.4 Thomée Scheme ADE

Using 3.13, the Thomée scheme is implemented into the ADE method by applying the ADE derivation to the generalised PDE in 3.5 for a bond price B. The result is the following where the known terms have been moved to the right-hand side:

$$B_0^{n+1}\left(\frac{\alpha}{h^2}-\frac{\beta}{2h}\right)+B_1^{n+1}\left(-\frac{1}{h^2}-\frac{\alpha}{k}+\gamma\right)=B_1^n\left(\frac{\alpha}{h^2}-\frac{1}{k}\right)-B_2^n\left(\frac{\beta}{2h}+\frac{C}{2h}\right).$$

The above form alongside 3.13 form a 2×2 system which is solved to obtain the bond prices at $j=0$ and $j=1$. The bond price B_0^{n+1} is then used as an approximation to the near-field boundary condition during the ADE iterative process.

3.4 Method of Lines

The Method Of Lines (MOL) scheme is a numerical method that involves the algebraic approximation of an initial value boundary problem. The generalised d/c/r PDE (3.5) consists of two independent variables x and t, and the method takes the space variable and converts the PDE to that of a system of ordinary differential equations (ODEs), in which there exists only one independent variable. Schiesser et al. (2009) [21, pg.6] defines the method as a replacement of the spatial derivatives with algebraic approximations such that PDEs independent variable remains.

3.4.1 Generalised Method of Lines

The MOL scheme, when applied to 3.5 results in a general framework in which the d/c/r equations throughout this thesis, can be implemented with ease. The standard procedure begins by replacing the derivatives in the spatial domain with the following central-difference quotients:

$$\frac{\partial V}{\partial x} = \frac{V(x+h) - V(x-h)}{2h} + O(h^2) = \frac{V_{n+1} - V_{n-1}}{2h}, \quad n = 1, \cdots, N.$$

$$\frac{\partial^2 V}{\partial x^2} = \frac{V(x+h) - 2V(x) + V(x-h)}{h^2} + O(h^2) = \frac{V_{n+1} - 2V_n + V_{n-1}}{h^2}.$$

Note that there is only one independent variable such that the central difference quotient variables no longer contain a spatial subscript (j). Moreover, step size h is now defined as the space between each ODE line discretisation.

Substituting the above into the generalised PDE yields the following:

$$\frac{\partial V_n}{\partial t} = \alpha(x,t)\Big(\frac{V_{n+1} - 2V_n + V_{n-1}}{h^2}\Big) + \beta(x,t)\Big(\frac{V_{n+1} - V_{n-1}}{2h}\Big) + \gamma(x,t)V_n.$$

Rearranging yields:

$$\frac{\partial V_n}{\partial t} = V_{n-1}\Big(\frac{\alpha(x,t)}{h^2} - \frac{\beta(x,t)}{2h}\Big) + V_n\Big(\frac{-2\alpha(x,t)}{h^2} + \gamma(x,t)\Big) + V_{n+1}\Big(\frac{\alpha(x,t)}{h^2} + \frac{\beta(x,t)}{2h}\Big). \tag{3.14}$$

Equation 3.14 corresponds to a system of ODEs that require solving via the use of an integration scheme. The benefit of this method is the existence of many open-source ODE integration solvers that make coding the Method Of Lines scheme easier than other methods such as ADE. The above system can be solved using explicit numerical integration, such as the Explicit Euler method. However, this suffers from instability caused by the CFL-number rising above some critical value (see Schiesser and Griffiths (2009) [21, pg.9] for more details) and has a local truncation error of $O(h^2)$. This is often known as the simplest Runge-Kutta method and is not used within this thesis. Instead, the widely known ODE solver, known as the Runge-Kutta Dormand-Prince (RKDP) method is implemented.

Runge Kutta Dormand-Prince ODE Method

Unlike the explicit Euler method, the RKDP method is a Runge-Kutta solver and was developed in 1980 by Dormand and Prince [9]. The method comprises of seven stages and uses six-stage functions to generate a fifth and fourth-order solution (Dormand and Prince (1980) [9, pg.23]). It is known to have a truncation error of order five, which in comparison to lower-order Runge-Kutta schemes produce a significant improvement in approximating ODE solution. This was confirmed in Kimura's 2009 paper [16, pg.4,6].

Briefly consider the standard fourth order Runge-Kutta method (see Stoer and Bulirsch (1993) [23, pg.438]):

$$V_{n+1}(t;h) = V_n(t;h) + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \qquad h > 0.$$

Where:

$$\begin{aligned} k_1 &= f(t, V), \\ k_2 &= f(t + 0.5h, V + 0.5hk_2), \\ k_3 &= f(t + 0.5h, V + 0.5hk_2), \\ k_4 &= f(t + h, V + hk_3). \end{aligned}$$

The above was generalised to higher orders [9, pg.19] and gave rise to the following generalisation:

$$V_{n+1} = V_n + h\sum_{i=1}^{s} b_i k_i.$$

Where: $k_s = f(t_n + c_s h, V_n + h(a_{s,1}k_1 + a_{s,2}k_2 + \cdots + a_{s,s-1}k_{s-1})),$

Where: a_{ik} is the Runge-Kutta Matrix.
s is the number of stages (k).
$\{b_i\}_{i=1}^{s}$, $\{c_i\}_{i=2}^{s}$ are the stage weight coefficients.

Such that: $j \in [1, s], \quad j < i \leq s.$

Dormand and Prince (1980) presented their method via the coefficient table: Using the Runge-Kutta generalisation and figure 3.3, the difference between

c_i	a_{ij}						$\hat{b}_i$	b_i
0							$\frac{35}{384}$	$\frac{5179}{57600}$
$\frac{1}{5}$	$\frac{1}{5}$						0	0
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					$\frac{500}{1113}$	$\frac{7571}{16695}$
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				$\frac{125}{192}$	$\frac{393}{640}$
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			$-\frac{2187}{6784}$	$-\frac{92097}{339200}$
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		$\frac{11}{84}$	$\frac{187}{2100}$
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	$\frac{1}{40}$

FIGURE 3.3: Dormand-Prince (1980) Method Coefficient Table
[9, pg.23]

the fourth and fifth order Runge-Kutta equations are taken. The result is then used to obtain an optimised step h which is subsequently used in the next stage calculation (see Kimura (2009) [16, pg.1-2] for an in-depth example).

3.4.2 MOL Black-Scholes Implementation

Substituting the Black-Scholes d/c/r terms into equation 3.14 yields the following:

$$\frac{\partial V_n}{\partial t} = V_{n-1}\left(\frac{\sigma^2 S^2}{2h^2} - \frac{rS}{2h}\right) + V_n\left(\frac{-\sigma^2 S^2}{h^2} - r\right) + V_{n+1}\left(\frac{\sigma^2 S^2}{2h^2} + \frac{rS}{2h}\right).$$

3.4.3 MOL CIR PDE Implementation

Consider the diffusion, convection and reaction terms of the domain transformed CIR zero-coupon bond price PDE (3.9) and substitute into the generalised Method Of Lines equation.

$$\begin{aligned}\frac{\partial V_n}{\partial t} = {} & V_{n-1}\left(\frac{\sigma^2 y(1-y)^3}{2h^2} - \frac{a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2}{2h}\right) + \\ & V_n\left(\frac{-\sigma^2 y(1-y)^3}{h^2} - \frac{y}{1-y}\right) + V_{n+1}\left(\frac{\sigma^2 y(1-y)^3}{2h^2} + \frac{a(1-y)^2 - by(1-y)\sigma^2 y(1-y)^2}{2h}\right).\end{aligned}$$

4 Sensitivity Approximation Methods

The introduction of the numerical methods in chapter 3 has the primary use of approximating bond and option prices. However, this thesis is concerned with the approximation of sensitivities using the results obtained from the prior numerical methods. Within this section, each sensitivity approximation method is introduced and expanded on concerning the Black-Scholes PDE (2.1) and the CIR zero-coupon pricing PDE (3.9).

4.1 Divided Difference Method

This thesis has partially covered Divided Differences; however, in this case, the central, forward and backward difference schemes will be implemented into sensitivity calculations. The foundation of Divided Difference lies with Taylor series expansion and formation of difference quotients used to approximate derivatives. The Taylor series gives rise to the truncation errors that reflect the finite components of the series used in the calculation.

First consider a second-order continuous function denoted f defined on an interval with step size $h > 0$. The forward and backward differences are given by:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h); \qquad f'(x) = \frac{f(x) - f(x-h)}{h} + O(h).$$

These are first order approximations of the derivative (see big'O'notation) and only require two points for approximation at the cost of a higher discrepancy between the exact and approximated value. The centralised difference is constructed from the above quotients and is of the form:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

Which is a second order approximation and will be used as the primary sensitivity approximation. The final difference quotient is a second order approximation of the second derivative given by:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2).$$

The application of the Divided Difference method to the approximation of sensitivities requires a vector V containing J prices of a financial derivative (or instrument). The difference quotients can then be represented on the following interval $(j = 1, \cdots, J-1)$ in which the vector is defined. However this interval shifts when considering the first-order difference such as the forward difference ($j = 0, \cdots, J-1$) or backward difference ($j = 1, \cdots, J$).

The inherent disadvantage of such method reveals that it suffers from a cancellation effect. If the step size ($h > 0$) is set too low the difference between $f(x+h)$ and $f(x-h)$ becomes small enough that it reaches machine precision resulting in a cancellation effect. This issue is common with difference quotients, and this leads to the question of what step size is required for accurate computation.

Regarding the approximation of option price sensitivities. Each of the first-order sensitivities denoted V' are approximated using the central difference quotient at time step n along the jth interval:

$$V' = \frac{V_{j+1}^n - V_{j-1}^n}{2h}. \tag{4.1}$$

The exception is Theta (2.5) (or time related sensitivities) which utilises the backward difference as time is continuous in a single direction and will not change:

$$\theta = \frac{V_j^n - V_{j-1}^n}{k}, \quad \text{Where } k \text{ is the time step.} \tag{4.2}$$

Finally, Gamma uses the second derivative approximation given by the following:

$$\Gamma = \frac{V_{j+1}^n - 2V_j^n + V_{j-1}^n}{h^2}, \qquad j = 1 \leq j \leq J-1. \tag{4.3}$$

Additionally, the sensitivities associated with the price of a zero-coupon bond are identical to that of the option price sensitivities. Therefore, the implementation of the Divided Difference into the sensitivity approximations is trivial.

4.2 Cubic Spline Interpolation

Cubic Spline is an interpolatory method used to fit a set of numerical points with a series of unique cubic polynomials, known as cubic splines, between each data point. The splines depend on a set of coefficients named moments (M) that adjust the curve of the spline near its endpoints to produce a smooth continuous curve that fits the numerical data. The advantage of this method lies with its avoidance of over-fitting and its ability to create a smooth solution, unlike standard piece-wise interpolation. Therefore the severe flaw of interpolatory methods known as Runge's Phenomenon (as a result of over-fitting) is avoided.

A spline is a composite function defined on some interval $[A, B]$ where: $A = x_0 < x_1 < \cdots < x_n = B$, made of n low order polynomials (in this case 3rd order) for :

$$S(x) = \begin{cases} S_0(x), & x_0 \leq x \leq x_1, \\ \vdots \\ S_{i-1}(x), & x_{i-1} \leq x \leq x_i \\ \vdots \\ S_{n-1}(x), & x_{n-1} \leq x \leq x_n. \end{cases} \tag{4.4}$$

Subject to the following continuity constraint in that each consecutive polynomial must join at each interval t_i:

$$S_{i-1}(x) = S_i(x), \quad \text{for:} \quad i = 1, 2, \cdots, n-1.$$

In addition to the smoothness constraint in which for some degree k the derivative of each low order polynomial must be the same at each interval t_i:

$$S^k_{i-1}(x_i) = S^k_i(x_i), \quad \text{for:} \quad i = 1, 2, \cdots, n-1.$$

Applying a cubic spline to the above (4.4) yields the following:

$$S(x) = \begin{cases} S_0(x) = a_0x^3 + b_0x^2 + c_0x + d_0, & x_0 \leq x \leq x_1, \\ \vdots \\ S_{i-1}(x) = a_{i-1}x^3 + b_{i-1}x^2 + c_{i-1}x + d_{i-1}, & x_{i-1} \leq x \leq x_i \\ \vdots \\ S_{n-1}(x), = a_{n-1}x^3 + b_{n-1}x^2 + c_{n-1}x + d_{n-1} & x_{n-1} \leq x \leq x_n. \end{cases} \tag{4.5}$$

Given the following constraints where the spline is a second order continuous function:

$$S(x) = \begin{cases} S_{i-1}(x) = S_i(x), \\ S'_{i-1}(x) = S'_i(x) \\ S''_{i-1}(x) = S''_i(x). \end{cases}$$

There exists a set of cubic spline variations, however, for the purpose of this thesis, the natural cubic spline will be used in which the following holds: $S''(x_0) = S''(x_n) = 0$.

Proofs from Stoer and Bulirsch (1993) [23, pg.99] can be observed that result in the coefficient expressions of a, b, c and d leading to the following cubic spline equation, where $y_i \in \mathbb{R}$ for $i = 0, 1, \cdots, n-1$:

$$\begin{aligned} S_i(y, x) &= \Big(\frac{h_i}{h_{i+1} + h_i}\Big)M_{i-1} + 2M_i + \Big(\frac{h_{i+1}}{h_{i+1} + h_i}\Big)M_{i+1} \\ &= \frac{6}{h_{i+1} + h_i}\Big(\frac{y_{i+1} - y_i}{h_{i+1}} - \frac{y_i - y_{i-1}}{h_i}\Big). \end{aligned}$$

Let $b_i = \frac{1}{h_i}\Big(y_{i+1} - y_i\Big)$ and $b_i = \frac{1}{h_i}\Big(y_i - y_{i-1}\Big)$ to obtain:

$$S_i(y,x) = \Big(\frac{h_i}{h_{i+1} - h_i}\Big)M_{i-1} + 2M_i + \Big(\frac{h_{i+1}}{h_{i+1} - h_i}\Big)M_{i+1} = \frac{6\big(b_{i+1} - b_i\big)}{h_{i+1} + h_i} = d_i. \tag{4.6}$$

Let $\frac{h_i}{h_{i+1}-h_i} = \nu$, $\frac{h_{i+1}}{h_{i+1}-h_i} = \xi$ and apply the Natural Cubic spline to obtain the following $n \times n$ matrix, giving rise to the following:

$$\begin{bmatrix} 2 & 0 & & & \\ \nu_1 & 2 & \xi_1 & & \\ & \nu_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \xi_{n-1} \\ & & & 0 & 2 \end{bmatrix} \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ \vdots \\ M_n \end{bmatrix} = 6 \begin{bmatrix} 0 \\ d_1 \\ \vdots \\ d_{n-1} \\ 0 \end{bmatrix}$$

The above system is solved using a tridiagonal matrix solver to obtain the moments (M) used to adjust the spline. Upon using moment solving methods such as the Thomas algorithm, the calculated moments are then implemented into the following spline functions (a derivation of the formulae can be found in Stoer and Bulirsch's (1993) paper [23, pp. 97-100]):

$$S(y,x) = M_j \frac{(x_{i+1}-x)^3}{6h_{i+1}} + M_{i+1}\frac{(x-x_i)^3}{6h_{i+1}} + A_j(x - x_i) + B_i.$$

Where: $A_j = \frac{y_{i+1}-y_i}{h_{j+1}} + \frac{h_{j+1}}{6}(M_{j+1} - M_j)$,

$$B_j = y_j - M_j \frac{h_{j+1}^2}{6}.$$

Where the first and second derivatives used to calculate sensitivities is given by:

$$S'(y,x) = -M_j\Big(\frac{(x_{j+1}-x)^2}{2h_{j+1}}\Big) + M_{j+1}\Big(\frac{(x-x_j)^2}{2h_{j+1}}\Big) + \frac{y_{j+1}-y_j}{h_{j+1}} - \frac{h_{j+1}(M_{j+1}-M_j)}{6}.$$

$$S''(y,x) = M_j\Big(\frac{x_{j+1}-x}{h_{j+1}}\Big) + M_{j+1}\Big(\frac{x-x_j}{h_{j+1}}\Big).$$

The above cubic spline derivatives are used for calculating the option and bond sensitivities by supplying a vector of prices and defining the interval in which the cubic spline algorithm can be applied.

4.3 Complex Step Method

The Complex Step Method (CSM) is a relatively unknown way of approximating sensitivities of analytical functions. In recent years, the benefits of such a method have become known and implemented in fields such as Biology and Geophysical analysis. In chapter 4.1, the Divided Difference was commented on in regards to its main disadvantage: the cancellation effect, where a function evaluated at two consecutive points along an interval is subtracted at an extremely short time step h. The resulting differences become smaller than machine precision resulting in a subsequent cancellation. Squire and Trapp (1998) [22, pg.110-111] discussed this effect and showed that when a function is approximated and is analytical, the approximation can be carried out using a complex number where the imaginary component is evaluated to produce an approximation. The result is an approximation with no subtraction operator, in turn removing any cancellation effect for smaller step sizes. This was further supported through the Taylor series expansion about a point α_0 of some arbitrary function f:

$$f(\alpha_0 + ih) = f(\alpha_0) + ihf'(\alpha_0) - \frac{h^2}{2!}f''(\alpha_0) - \frac{ih^3}{3!}f'''(\alpha_0) + \cdots$$

Taking the imaginary part or the real part then rearranging leads to the respective derivative approximations:

$$f(\alpha_0) = \frac{Im(f(\alpha_0 + ih)}{h} + O(h^2); \quad f''(\alpha_0) = \frac{2(f(\alpha_0) - Re(f(\alpha_0 + ih))}{h} + O(h^2).$$

The second derivative provides a limitation of the method in that it is real, so no imaginary parts are included. However, Abreu (2013) in [1] generalised the CSM and introduced a complex step where $ih = h + iv$. Assuming that if $v = \sqrt{3}h$, the derivative approximation becomes a 4th order approximation and contains only imaginary components:

$$\begin{aligned} f'(x) &= \frac{Im(f(\alpha+h+iv))+Im(f(\alpha+iv))}{2v} + O(h, v^2) \\ f''(x) &= \frac{Im(f(\alpha+h+iv))-Im(f(\alpha+iv))}{hv} + O(h, v^2) \end{aligned}$$

The above method can be applied to the analytical Black-Scholes and CIR zero-coupon bond equations to calculate sensitivities as an alternative to calculating the closed form solution. Due to the simplicity of the Black-Scholes closed form this may not be a better alternative; yet the large CIR closed form sensitivity equations in Appendix B would make the CSM a viable alternative.

4.4 Forward Automatic Differentiation

Forward Automatic Differentiation (AD) is another method applied to this thesis. Similar to Complex-Step Method, AD is relatively new in comparison to finite difference methods, with the ability to calculate high order derivatives and gradients without a large computational overhead. Unlike finite difference methods, AD does not suffer from subtractive cancellation and instability issues when dealing with differencing (similar to the complex step method).

However, unlike FDM, the method suffers significantly from its difficulty to implement into coding languages such as C++ or MATLAB. In reference to C++, AD was only recently implemented through the use of 'Template Metaprogramming' which in short compiles source code via templates at compile time (see Meyers (2005) [18, pg.233]). The issue is that template metaprogramming is complicated and hard to implement, resulting in relatively low adoption of the method to that of the classical methods. Despite the present issues of the method, there exists a range of C++ Automatic Differentiation libraries that reduce the difficulty of code implementation.

Byrne and Greenwell (2017) [6] give an intuitive explanation of how AD differs from finite-difference and symbolic differentiation. They explain that "AD uses the same programmatic logic dictated by code except it propagates along the gradient information via standard properties such as the chain rule". Template meta-programming is applied explicitly for this action of propagating gradients.

Forward Automatic Differentiation relates to the forward propagation of derivative information, while Backwards Automatic Differentiation is concerning the backwards propagation of derivatives. The concept is intuitive when considering a simple multi-variable function that requires differentiation. The following function will be separated into its elementary/primitive functions and then into its subsequent input variables. Consider the following:

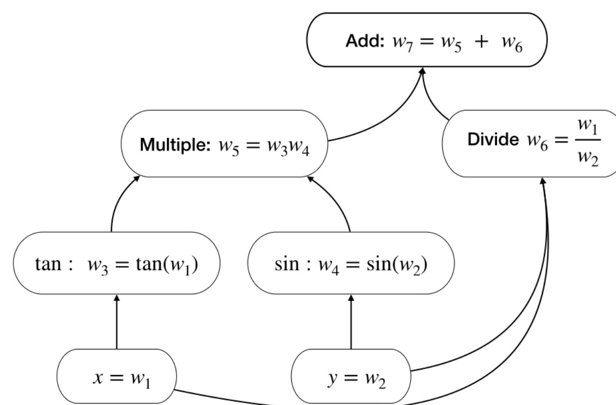

FIGURE 4.1: Automatic Differentiation Simplistic Example

$$f(x,y) = tan(x)sin(y) + \frac{x}{y}.$$

Figure 4.1 is a visual representation of simple Automatic Differentiation applied to the above function. The derivative of $f(x,y)$ with respect to x is calculated using the following procedure:

Let $Dw = \frac{dw}{dx}$,

$$\begin{aligned}
Dw_1 &= Dx = 1, \\
Dw_2 &= Dy = 0, \\
Dw_3 &= D\tan(w_1) = \sec^2(w_1)Dw_1 = \sec^2(w_1), \\
Dw_4 &= D\sin(w_2) = cos(w_2)Dw_2 = 0, \\
Dw_5 &= D(w_3 w_4) = w_3 Dw_4 + w_4 Dw_3 = w_4 \sec^2(w_1), \\
Dw_6 &= D\left(\tfrac{w_1}{w_2}\right) = \tfrac{w_1 Dw_2 - w_2 Dw_1}{w_2^2} = \tfrac{1}{w_2}, \\
Dw_7 &= D(w_5 + w_6) = Dw_5 + Dw_6 = w_4 \sec^2(w_1) + \tfrac{1}{w_2}.
\end{aligned}$$

Substituting w_4, w_2 and w_1 into Dw_7 yields the derivative:

$$\frac{\partial}{\partial x} f(x,y) = \sin(y)\sec^2(x) + \frac{1}{y}.$$

The advantage of using AD is that for a function $f(x)$ one may wish to calculate the value of the function and its derivative at a specific point. AD starts to share similarities with the Complex-Step Method but with dual numbers instead of complex numbers. By using dual numbers, AD calculates both the value of some function $f()$ and $f'()$ within the same calculation (hence Automatic Differentiation).

4.4.1 Dual Numbers

As a brief introduction, first consider a complex number where i is an imaginary number and satisfies the following property: $i^2 = -1$. Dual numbers are extremely similar to complex numbers with the caveat that the letter epsilon is used (ϵ) and satisfies the following property: $\epsilon^2 = 0$. In both dual and complex space (note: dual numbers are an extension of the real number space), elementary operations will result in a real and dual number component. Consider the second example:

$$f(x,y) = xy + sin(x), \quad \text{Find } f(2,2) \text{ and } f'(2,2):$$

Introduce the following dual numbers: $x = (2 + \epsilon_1)$ and $y = (2 + \epsilon_2)$.

$$\begin{aligned}
f(2+\epsilon_1, 2+\epsilon_2) &= (2+\epsilon_1)(2+\epsilon_2) + \sin(2+\epsilon_1), \\
&= 4 + 2\epsilon_1 + 2\epsilon_2 + 0 + \sin(2+\epsilon_1), \\
&= 4 + 2\epsilon_1 + 2\epsilon_2 + \sin(2) + \epsilon_1\cos(2),
\end{aligned}$$

$$= \quad (4+\sin(2)) + \epsilon_1(2+\cos(2)) + 2\epsilon_2.$$

The above equation corresponds to $f(2,2)$, $f'_x(2,2)$ and $f'_y(2,2)$:

$$f(2,2) = 4+\sin(2), \quad f'_x(2,2) = 2 + cos(2), \quad f'_y(2,2) = 2.$$

4.5 Continuous Sensitivity Equation (CSE)

This thesis hopes to introduce and test a method of calculating sensitivities via the continuous sensitivity equation using the literature from previous sections. This section will introduce a variety of sensitivity equations (alongside their derivations) for both the Black-Scholes PDE and the CIR zero-coupon bond PDE.

Before any derivations take place, the CSE method varies to that of the previous methods as it works in a 'backward' sense. Previous methods first require a bond pricing numerical method such as the Crank Nicolson method (see Chapter 3.2.1); from there a secondary method approximates the sensitivity. The CSE is a PDE of an option's sensitivity and only requires one method such as those described in chapter 3 to approximate the respective sensitivity.

The main disadvantage is that the resulting PDEs can be challenging to code regarding its boundary conditions and initial conditions. In addition, inhomogeneous terms (source terms) may exist which require the computation of the financial instrument prices that must be calculated first before approximating the CSE. This could potentially increase computational time, reduce the accuracy of the sensitivity approximation and introduce multiple solutions. The remaining section will cover the CSEs associated with the CIR zero-coupon bond pricing PDE and the Black-Scholes PDE.

Consider the general d/c/r partial differential equation:

$$\frac{\partial V(x,t)}{\partial t} = \alpha(x,t)\frac{\partial^2 V(x,t)}{\partial x^2} + \beta(x,t)\frac{\partial V(x,t)}{\partial x} - \gamma(x,t)V(x,t).$$

First take the derivative with respect to some sensitivity x of the above PDE:

$$\frac{\partial}{\partial x}\Big(\frac{\partial V(x,t)}{\partial t}\Big) = \frac{\partial}{\partial x}\Big(\alpha(x,t)\frac{\partial^2 V(x,t)}{\partial x^2} + \beta(x,t)\frac{\partial V(x,t)}{\partial x} + \gamma(x,t)V(x,t)\Big).$$

Dropping the variable notation from the d/c/r components and applying the chain rule:

$$\frac{\partial^2 V}{\partial t \partial x} = \frac{\partial \alpha}{\partial x}\frac{\partial^2 V}{\partial x^2} + \alpha\frac{\partial^3 V}{\partial x^3} + \frac{\partial \beta}{\partial x}\frac{\partial V}{\partial x} + \beta\frac{\partial^2 V}{\partial x^2} + \frac{\partial \gamma}{\partial x}V + \gamma\frac{\partial V}{\partial x}.$$

Introduce the following substitution $\frac{\partial V}{\partial x} = X$.

$$\frac{\partial X}{\partial t} = \frac{\partial \alpha}{\partial x}\frac{\partial X}{\partial x} + \alpha \frac{\partial^2 X}{\partial x^2} + \frac{\partial \beta}{\partial x} X + \beta \frac{\partial X}{\partial x} + \frac{\partial \gamma}{\partial x} V + \gamma X.$$

Rearranging results in the CSE for some sensitivity x of the general d/c/r equation.

$$\frac{\partial X}{\partial t} = \alpha \frac{\partial^2 X}{\partial x^2} + \left(\beta + \frac{\partial \alpha}{\partial x}\right)\frac{\partial X}{\partial x} + \left(\gamma + \frac{\partial \beta}{\partial x}\right) X + \frac{\partial \gamma}{\partial x} V. \tag{4.7}$$

It can be immediately observed that differentiating the generalised PDE by x leads to a CSE containing a source term. Despite this the equation remains in the form of the d/c/r equation when the source term is taken to the right hand-side of the equality.

The following set of CSE equations will contain all derivations within Appendix C. Additionally, the CSEs associated with Greek Theta and time until expiry will be missing from the following derivations due to its abstract nature and difficulty in coding since time is, by definition, an independent variable. However, this could be a topic for further research.

4.5.1 Black-Scholes Continuous Sensitivity Equations

The famous Black-Scholes PDE is given again:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV, \qquad \text{Where: } V = V(S,t). \tag{4.8}$$

Delta CSE

Following the order of chapter 2.2 the Delta sensitivity is obtained by differentiating 4.8 with respect to the underlying S to obtain:

$$\frac{\partial \Delta}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \Delta}{\partial S^2} + (\sigma^2 + r) S \frac{\partial \Delta}{\partial S}. \tag{4.9}$$

The CSE requires its own set of boundary and initial conditions. By definition the Delta of a call option has a maximum value of 100% or 1.0 and a minimum value of 0. Put options are the opposite and range from -1.0 to 0. With this information the boundary conditions are set:

$$\Delta_{call}(0,t) = 0, \qquad \Delta_{call}(S_{\max},t) = 1.$$

Concerning the initial condition, the option payoff of the Black-Scholes PDE is differentiated with respect to S, resulting in the following Heaviside step

function. Consider the call option case:

$$V_{call}(S,T) = \begin{cases} S-K, & \text{if } S > K. \\ 0, & \text{if } S < K. \end{cases}$$

$$\Delta_{call}(S,T) = \begin{cases} 1, & \text{if } S > K. \\ 0, & \text{if } S < K. \end{cases}$$

Where K is the strike price of the option.

Since the CSE 4.9 contains no source terms and has simple boundary conditions, it is relatively straight forward to obtain and requires little mathematical derivation.

Gamma CSE

As previously stated, Gamma is the derivative of Delta. Therefore, the CSE is derived from once again differentiating Delta by the underlying yielding the following CSE (see Appendix C):

$$\frac{\partial \Gamma}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \Gamma}{\partial S^2} + \left(2\sigma^2 S + rS\right)\frac{\partial \Gamma}{\partial S} + (\sigma^2 + r)\Gamma. \tag{4.10}$$

The boundary conditions of Gamma are the same for both (longed) call and put options in that Gamma will remain positive. It describes the rate of change of Delta, which is commonly given in percentages. Based on this, the boundary conditions are as follows:

$$\Delta_{call}(0,t) = 0, \qquad \Delta_{call}(S_{\max},t) = 1.$$

The initial condition is obtained by taking the derivative of the Heaviside function with respect to the underlying. The resulting derivative is the well known Dirac delta function which requires approximation (see Walden (1999) [25]).

However, for this thesis, a somewhat crude analytical approximation of the function is introduced beginning with the following logistic function: a smooth approximation of the Heaviside function:

$$\Delta(S,T) = \mathcal{H}(S-K) \approx \frac{1}{1+e^{-\lambda(S-K)}}.$$

Where λ is the steepness of the logistical curve and K is the options strike price. Differentiating the above equation with respect to $(S-K)$ yields:

$$\Gamma(S,T) = \frac{\partial \mathcal{H}(S-K)}{\partial (S-K)} = \delta(S-K) \approx \frac{2\lambda e^{-2\lambda(S-K)}}{(e^{-2\lambda(S-K)}+1)^2}.$$

The potential downside of the above initial condition is that it is a derivative of an approximation. This may reduce the computational accuracy of the results obtained when approximating sensitivities and will be commented on within the code implementation and results in chapters 5-6.

Vega CSE

Vega is obtained by differentiating the Black-Scholes PDE with respect to its volatility parameter (σ):

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \mathcal{V}}{\partial S^2} + rS\frac{\partial \mathcal{V}}{\partial S} - r\mathcal{V} + \sigma S^2 \frac{\partial^2 V}{\partial S^2}. \tag{4.11}$$

The issues associated with the Vega CSE stem from its initial condition and boundary conditions. Unlike the previous CSEs, volatility is not a parameter used within the payoff function of the Black-Scholes PDE. Therefore differentiating it with respect to σ results in:

$$\mathcal{V}(S,T) = 0.$$

Moreover, Vega is always positive for put and call options making the near-field boundary condition equal to zero. Moreover, the far-field boundary condition can be deduced either through a truncated or transformed domain. In a non-general sense, a similar option would be analysed, and the highest recorded Vega would be taken and multiplied by some user-defined constant to obtain the far-field boundary. However, domain transformation would be a suitable solution for this issue.

Appendix C.3 contains the derivation for both the domain transformed Black-Scholes PDE and subsequent domain transformations of the Vega and Rho CSE. Consider the transformed Vega CSE:

$$\begin{aligned}\frac{\partial \mathcal{V}}{\partial t} &= \frac{1}{2}\sigma^2 y^2 (1-y)^2 \frac{\partial^2 \mathcal{V}}{\partial y^2} + \Big(ry(1-y) - \sigma^2 y^2 (1-y)\Big)\frac{\partial \mathcal{V}}{\partial y} - r\mathcal{V} \\ &\quad + \sigma y^2 (1-y)^2 \frac{\partial^2 V}{\partial y^2} - 2\sigma y^2 (1-y)\frac{\partial V}{\partial y}.\end{aligned} \tag{4.12}$$

The transformation of the Vega CSE has bounded the spatial interval to being that of a unit interval, removing the issue of choosing a far-field boundary via domain truncation. Setting $y=0$ or 1 in the transformed Vega CSE yields the

following ordinary differential equation:

$$\frac{\partial \mathcal{V}}{\partial t} + r\mathcal{V} = 0 \rightarrow \mathcal{V} = Ce^{rt}.$$

Given that Vega decreases when the option approaches expiry at time T the issue consists of Vega not becoming zero at expiry, but under the generalised case it may be considered logical as Vega is always positive. The initial condition is therefore set to 0 at expiry, resulting in the following boundary conditions:

$$\mathcal{V}(0,t) = 0, \qquad \mathcal{V}(1,t) = 0.$$

In addition it is worth noting that even under domain transformation the initial condition of the Vega CSE remains the same.

Rho CSE

Option Greek Rho follows the same case as the Vega CSE. The standard Black-Scholes PDE is differentiated with respect to its interest rate (r) resulting in the following CSE equation:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \rho}{\partial S^2} + rS\frac{\partial \rho}{\partial S} - r\rho + S\frac{\partial V}{\partial S} - V.$$

Rho, however, is not always positive. For European put options, Rho is negative, and for European call options, Rho is always positive. For this reason, the boundary conditions will change depending on the type of option; yet suffer from the question of, what are the boundary conditions of the CSE. In the case of the call option (vice-versa), Rho will steadily decrease as the option approaches expiry. Therefore the near-field boundary can be set to 0, as for the call option case Rho must be positive. The far-field boundary can be obtained by intuition in a non-general case, but for the generalised case, the spatial domain of the CSE will be transformed into that of a unit interval.

Consider the transformed Rho CSE:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2 y^2(1-y)^2\frac{\partial^2 \rho}{\partial y^2} + \Big(ry(1-y) - \sigma^2 y^2(1-y)\Big)\frac{\partial \rho}{\partial y} - r\rho - V + y(1-y)\frac{\partial V}{\partial y}.$$

Setting the transformed variable y to zero or one yields:

$$\frac{\partial \rho}{\partial t} + r\rho = -V, \quad \rightarrow \quad \rho = \frac{C - Vt}{1 + rt}. \tag{4.13}$$

Assuming ρ is 0 at expiry yields the following near/far field boundary conditions under domain transformation:

$$\rho_{call/put}(0,t) = \rho_{call/put}(1,t) = \frac{Vt}{1+rt}, \qquad \text{Where V is the bond price.}$$

The initial condition is thus given by: $\rho(y, T) = 0$. The issue of what initial condition is required is a possible topic of research. Although 0 has been used in both the cases of the Rho and Vega CSEs, the value of Rho and Vega may never reach zero as maturity approaches. Chapter 6 uses the Rho CSE to investigate the effect of different Rho values at expiry under the assumption that Rho is decreasing.

4.5.2 Cox-Ingersoll-Ross Continuous Sensitivity Equations

As expanded in previous chapters, the CIR zero-coupon bond pricing PDE is given under a transformed domain. However, for consistency, the standard CIR PDE is given by:

$$\frac{\partial B}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial r^2} + (a - br)\frac{\partial B}{\partial r} - rB.$$

Each CSE will be presented using the above CIR PDE and then subsequently undergo domain transformation (full derivations of the CSEs are given in Appendix C.2). Lastly, a section is dedicated to the issues around the CSE boundary conditions and initial conditions.

Duration CSE

Similar to the Greek Delta CSE, the Duration CSE is obtained by differentiating the CIR PDE with its dependent variable, i.e. the interest rate. The following CSE is obtained:

$$\frac{\partial \mathcal{D}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{D}}{\partial r^2} + \left(\frac{1}{2}\sigma^2 + a - br\right)\frac{\partial \mathcal{D}}{\partial r} - (r + b)\mathcal{D} - B. \tag{4.14}$$

Applying domain transformation to the CSE yields:

$$\begin{aligned}\frac{\partial \mathcal{D}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{D}}{\partial y^2} + \left(\left(\frac{1}{2} - y\right)\sigma^2 + a - \frac{by}{1-y}\right)(1-y)^2\frac{\partial \mathcal{D}}{\partial y} \\ - \left(\frac{y}{1-y} + b\right)\mathcal{D} - B.\end{aligned} \tag{4.15}$$

Convexity CSE

Differentiating C.2 by the interest rate results in the following CSE for the convexity sensitivity:

$$\frac{\partial \mathcal{C}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{C}}{\partial r^2} + (\sigma^2 + a - br)\frac{\partial \mathcal{C}}{\partial r} - (2b + r)\mathcal{C} - 2\frac{\partial B}{\partial r}.$$

Consider the domain transformation identities:

$$\frac{\partial \mathcal{C}}{\partial r} = (1-y)^2 \frac{\partial \mathcal{C}}{\partial y}, \quad \frac{\partial^2 \mathcal{C}}{\partial r^2} = -2(1-y)^3 \frac{\partial \mathcal{C}}{\partial y} + (1-y)^4 \frac{\partial^2 \mathcal{C}}{\partial y^2}.$$

Applying the transformations to the Convexity CSE yields:

$$\frac{\partial \mathcal{C}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{C}}{\partial y^2} + (\sigma^2(1-y)^3 + a(1-y)^2 - by(1-y))\frac{\partial \mathcal{C}}{\partial y} - \left(2b + \frac{y}{1-y}\right)\mathcal{C} - 2(1-y)^2 \frac{\partial B}{\partial y}. \tag{4.16}$$

Speed of Adjustment CSE

The CSE associated to the Speed of Adjustment sensitivity (b/κ) is given by the following:

$$\frac{\partial \mathcal{K}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{K}}{\partial y^2} + (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)\frac{\partial \mathcal{K}}{\partial y} - \left(\frac{y}{1-y}\right)\mathcal{K} - y(1-y)\frac{\partial B}{\partial y}. \tag{4.17}$$

Volatility CSE

Differentiating the CIR zero-coupon bond pricing PDE by its volatility and applying domain transformation yields the following CSE:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{V}}{\partial y^2} + (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)\frac{\partial \mathcal{V}}{\partial y} - \left(\frac{y}{1-y}\right)\mathcal{V} - 2y\sigma(1-y)^2 \frac{\partial B}{\partial y} + \sigma y(1-y)^3 \frac{\partial^2 B}{\partial y^2}. \tag{4.18}$$

4.5.3 Boundary Conditions: Further Research Opportunity?

In this section, Fichera theory is applied to the CSEs introduced in chapter 4.5.2 with a discussion on possible implementations into code and solutions to the questions posed in this section.

Duration CSE

For reference, the Fichera function is:

$$\mathcal{F} = \sum_{i=1}^{n} \left(\beta_i - \sum_{k=1}^{n} \frac{\partial \alpha_{i,k}}{\partial x_k}\right) v_i, \quad | \quad \eta \to \mathbb{R}.$$

The Fichera function requires the diffusion and convection terms in 4.15 to yield:

$$\begin{aligned}\mathcal{F}(y) &= \left(\left(\frac{1}{2}-y\right)\sigma^2(1-y)^2 + \left(a-\left(\frac{by}{1-y}\right)\right)(1-y)^2 - \frac{1}{2}\sigma^2(1-y)^3 + \frac{3}{2}\sigma^2 y(1-y)^2\right)v(y),\\ &= \left(\frac{1}{2}\sigma^2(1-y)^2 + \frac{1}{2}\sigma^2 y(1-y)^2 + a(1-y)^2 - by(1-y) - \frac{1}{2}\sigma^2(1-y)^3\right)v(y),\\ &= \left(a-\left(\frac{by}{1-y}\right)\right)(1-y)^2 v(y).\end{aligned}$$

Consider the case at the near-field boundary where $v(0) = 1$.

$$\mathcal{F}(0) = a.$$

Therefore:

$$\begin{array}{lll} \mathcal{F}(0) \geq 0, & a \geq 0. & \text{No boundary conditions required.}\\ \mathcal{F}(0) < 0, & a < 0. & \text{Boundary conditions required.}\end{array}$$

Setting $y = 0$ in 4.15 yields the near-field condition:

$$\frac{\partial \mathcal{D}}{\partial t} = \left(\frac{1}{2}\sigma^2 + a\right)\frac{\partial \mathcal{D}}{\partial y} - b\mathcal{D} - B.$$

Convexity CSE

Substituting the diffusion and convection terms of 4.16 into the Fichera function yields:

$$\begin{aligned}\mathcal{F}(0) &= \left(\sigma^2(1-y)^3 + a(1-y)^2 - by(1-y) - \frac{1}{2}\sigma^2(1-y)^3 + \frac{3}{2}\sigma^2 y(1-y)^2\right)v(y),\\ &= \left(\frac{1}{2}\sigma^2(1-y)^3 + \left(a+\frac{3}{2}y\right)(1-y)^2 - by(1-y)\right)v(y).\end{aligned}$$

Consider the near-field boundary case where: $v(0) = 1$.

$$\mathcal{F}(0) = \left(\frac{1}{2}\sigma^2 + a\right).$$

The following cases are obtained:

$$\begin{array}{lll} \mathcal{F}(0) \geq 0, & \sigma^2 \geq -2a. & \text{no boundary conditions required.}\\ \mathcal{F}(0) < 0, & \sigma^2 < -2a. & \text{boundary conditions required.}\end{array}$$

Concerning the far-field boundary i.e. when $v(1) = -1$, it is found that $\mathcal{F}(1) = 0$. Therefore no boundary conditions are required, despite the transformed domain. Now setting $y = 0$, gives the following near-field boundary condition:

$$\frac{\partial \mathcal{C}}{\partial t} = (\sigma^2 + a)\frac{\partial \mathcal{C}}{\partial y} - 2\frac{\partial B}{\partial y} - 2b\mathcal{C}.$$

Speed of Adjustment CSE

Following the same procedure, the Fichera function is:

$$\mathcal{F}(y) = \Big(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2 - \frac{1}{2}\sigma^2(1-y)^3 + \frac{3}{2}\sigma^2 y(1-y)^2\Big)v(y).$$

Focusing on the near-field boundary such that $v(0) = 1$, yields:

$$\mathcal{F}(0) = \Big(a - \frac{1}{2}\sigma^2\Big).$$

Resulting in the following cases that are identical to the CIR PDE boundary conditions:

$\mathcal{F}(0) \geq 0,$	$\sigma \leq \sqrt{2a}.$	No boundary conditions required.
$\mathcal{F}(0) < 0,$	$\sigma > \sqrt{2a}.$	Boundary conditions required.

Additionally, the far-field boundary case where $v(1) = -1$ shows that no boundary conditions are required. Concerning the near-field condition, when setting $y = 0$, the boundary condition is:

$$\frac{\partial \mathcal{K}}{\partial t} = a\frac{\partial \mathcal{K}}{\partial y} \longrightarrow \frac{\partial \mathcal{K}}{\partial t} - a\frac{\partial \mathcal{K}}{\partial y} = 0.$$

which is identical to the CIR boundary condition which can thus be solved using the Thomée scheme.

Volatility CSE

A application of Fichera theory is to the Volatility CSE (4.18):

$$\mathcal{F}(y) = \Big(a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2 - \frac{1}{2}\sigma^2(1-y)^3 + \frac{3}{2}\sigma^2 y(1-y)^2\Big)v(y). \tag{4.19}$$

Consider the near-field boundary case with the following condition $v(0) = 1$:

$$\mathcal{F}(0) = \Big(a - \frac{1}{2}\sigma^2\Big).$$

Which yields the same boundary condition requirement as the CIR PDE and SoA CSE. Therefore:

$$\mathcal{F}(0) \geq 0, \qquad \sigma \leq \sqrt{2a}. \qquad \text{no boundary conditions required.}$$
$$\mathcal{F}(0) < 0, \qquad \sigma > \sqrt{2a}. \qquad \text{boundary conditions required.}$$

Setting $y = 0$ in 4.18 leads to the following boundary conditions:

$$\frac{\partial \mathcal{V}}{\partial t} = a\frac{\partial \mathcal{V}}{\partial y} \longrightarrow \frac{\partial \mathcal{V}}{\partial t} - a\frac{\partial \mathcal{V}}{\partial y} = 0.$$

Possible Issues...

Within this section multiple near-field boundary conditions have been obtained alongside their need for implementation when the Fichera condition is not satisfied ($\mathcal{F}(0) < 0$). The issue arises when analysing the PDEs and noticing that for the sensitivity PDEs the inhomogeneous term is that of a Bond price B. Therefore if the Fichera condition is satisfied ($\mathcal{F}(0) \geq 0$), the presence of the inhomogeneous term means that it does not guarantee uniqueness of the solution. The requirement of two sets of boundary conditions to solve for the inhomogeneous term and the sensitivity will result in more than one solution as the choice of boundary condition will significantly affect the approximation (see results table 6.14).

A further area of research presents itself where before applying any numerical method, it needs to be proven that the sensitivity equations are well-posed, and thus possess a unique solution in the cases of it satisfying and not satisfying the Fichera condition. The ramification of these equations not being well-posed include the introduction of errors of a large magnitude when applying standard numerical techniques. Therefore any approximation made without taking into account the possibility of multiple solutions will result in sub-par results effectively making the CSE approach unsuitable.

5 Code Implementation

Each of the methods covered in chapters 3 and 4 will be expanded on in this section regarding their implementation into code. It is worth noting that the implementation of such methods requires additional packages in the case of C++, which will be referenced throughout this section.

The following section will focus on the design of the code with much of the inspiration taken from [10], which shall be referenced accordingly. The later section will focus on the implementation of numerical methods covered in chapter 3 and the final section on the individual sensitivity calculation methods.

5.1 C++ Code Design

C++ is considered a low-level language and is essentially an iteration of the C coding language with the implementation of object orientation and low-level memory manipulation. The advantage of this language ranges from its versatility and applicability in many applications as it has no dependencies, unlike Java which uses interpreters.

Such a low-level language requires more attention in the areas of coding design, memory management and re-usability, which can be an inherent issue for novice coders. Gamma et al. (1995) [12] introduce a set of design pattern which aims to make code reusable, resulting in code with low coupling and higher cohesion. Within this thesis, the code implemented within C++ makes use of a bridge pattern that has the following definition (Gamma et al, 1995))[12, pg.19]:

The decoupling of an abstraction from its implementation so that the code can independently vary.

The bridge pattern is implemented to allow a separation between the definition of the hierarchies containing the information relating to the initial boundary value problem (IBVP) and the methods used to solve the IBVP and approximate the respective sensitivities. Duffy (2018) incorporated the same implementation of the bridge design alongside the factory and template method [10, pg.673-74]. The same implementation is followed, and 'abused' to incorporate the methods covered in previous chapters. Consider the following simplified coding diagram:

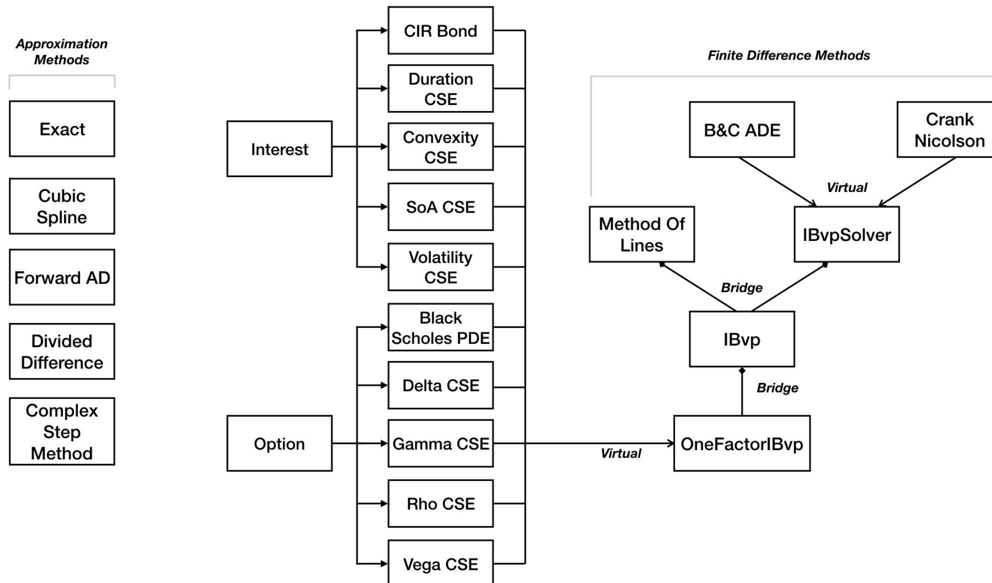

FIGURE 5.1: C++ Code Structure Diagram.

The above structure provides a simplistic representation of the classes which form the code. Three bridge pattern implementations exist within this code. Consider the *OneFactorIBvp* abstract base class, that contains a set of pure virtual functions that are overridden by the methods incorporated in the PDE classes; this is directly passed into the *IBvp* class via a pointer. *IBvp* obtains access to the methods in an instantiated PDE class through the *OneFactorIBvp* pointer. In addition, the *IBvp* class constructor takes in spatial and time axis intervals via a Range template class (Not shown in 5.4). This section of code is considered as the definer portion of the code:

```
11  class IBvp
12  {
13  public:
14
15      Range<val_type> xaxis;   // Space Interval
16      Range<val_type> taxis;   // Time Interval
17
18      OFI* imp;
19
20      IBvp() = delete;
21
22  public:
23      IBvp(OFI& executor, const Range<val_type>& xrange, const Range<val_type>& trange);
24
25      // Range Class
26      // -----------
27      const Range<val_type>& xrange() const;
28      const Range<val_type>& trange() const;
29
30      // Boundary Conditions for BS PDE
31      // ------------------------------
32      val_type LftBnd(val_type t) const;
33      val_type RhtBnd(val_type t) const;
```

FIGURE 5.2: C++ IBvp Class cpp file.

The second bridge implementation is between *IBvp* and the *IBvpSolver* base class which is responsible for the creation of the finite difference mesh and the subsequent calculation of (in this instance) option prices. *IBvpSolver* takes

in a pointer to the *IBvp* class providing the base class with access to the FDM axis information and the d/c/r equation methods.

The third bridge implementation is between *IBvp* and the *MethodOfLines* class. Such a design is required for the Boost library *integrate* function that utilises the RKDP ODE method to integrate a system of ordinary differential equations (see 3.4.1). The function requires a set of inputs that make it incompatible with the *IBvpSolver* base class forcing many of the inputs such as, mesh creation and calculation inside an overloaded operator. Therefore, to incorporate this method into the code requires a secondary bridge proceeding from the *IBvp* class. This section of the code is regarded as the solver portion.

The definer portion of the code will now be expanded on.

5.1.1 C++ Definer Code Section

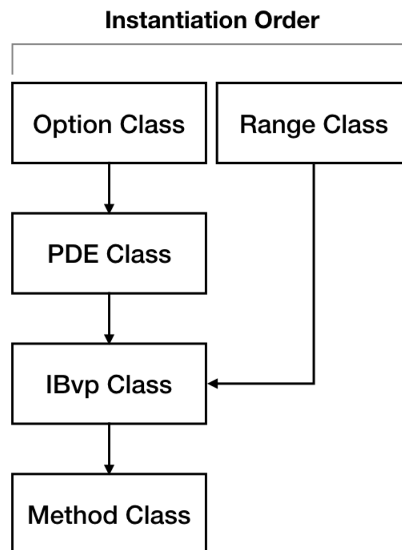

FIGURE 5.3: C++ Instantiation Flowchart.

Figure 5.3 provides the order in which classes are instantiated within the code written for this thesis. Only extracts of such code will be taken and expanded upon to provide a general idea on how the code functions. The Black-Scholes PDE will be used to expand on the classes given in 5.3, starting with the option class.

Option Class

The class *Option* is responsible for collecting user input and contain the necessary parameters to price either an option, in this case, or a bond in another instance.

```
1  #ifndef OPTION_HPP
2  #define OPTION_HPP
3
4  #include <iostream>
5
6  using val_type = double;
7
8  class Option
9  {
10 public:
11     // Define Option Values
12     val_type K;
13     val_type T;
14     val_type r;
15     val_type sig;
16     val_type b;
17     val_type SMax;
18
19     // Target Value;
20     val_type Target;
21
22     char type;
23 };
24
25 #endif
```

FIGURE 5.4: C++ Option Class hpp file.

Figure 5.4 is a simple class that contains the parameters of an option with the inclusion of a Target and type value. These are specifically implemented to identify specific values that have been approximated and whether the option is a European Put or Call option. An instance of the *Option* class is created with the subsequent user input of parameter values.

PDE Classes

Proceeding the option class instantiation comes the instantiation of a PDE object. Using the Black-Scholes PDE as an example in figure 5.5, *Option* is passed by reference to the *PdeBlackScholes* giving the class access to required option parameters. Each of the methods within *PdeBlackScholes* is inherited from the *OneFactorIBvp* class denoted *'OFI'* and remains constant such that no modification of the option class parameters can be made.

Classes that hold information on a specific PDE/CSE contain four sets of boundary conditions and two sets of initial conditions. As discussed in 4.5, many of the CSEs contain source (inhomogeneous) terms that require a set of either option or bond prices to calculate the sensitivity. Therefore, during the calculation of sensitivities, the associated bond/option price needs to be calculated in parallel, which requires its own boundary conditions and initial condition.

The set of classes contain diffusion, convection and reaction methods which define the PDE. Using the *Diffusion* method (line 43) as an example, the method will return the following diffusion term of the Black-Scholes equation:

$$\frac{1}{2}\sigma^2 S^2.$$

As each of the PDEs/CSEs covered within this thesis are d/c/r equations, the implementation of the *OneFactorIBvp* abstract base class becomes useful as multiple PDE class inclusion files with small changes do not need to be included in the *IBvp* class despite having minor changes to each d/c/r method.

```
11 class PdeBlackScholes : public OFI
12 {
13     // As other classes will be using the diffusion, convection etc. terms
14     // we need to ensure these are public.
15
16 public:
17     // Define following option class location. `
18     Option opt;
19
20     // Constructor (takes in option for access to parameters).
21     PdeBlackScholes(const Option& option);
22
23     // Destructor (default)
24     ~PdeBlackScholes();
25
26 public:
27
28     //------------------------------------------------------------------
29     //  Standard European Option
30     //------------------------------------------------------------------
31
32     // Boundary Conditions for standard option
33     // --------------------------------------
34     val_type LftBnd(val_type t) const;
35     val_type RhtBnd(val_type t) const;
36
37     // Initial Condition for standard option
38     // ------------------------------------
39     val_type IntCnd(val_type x) const;
40
41     // Diffusion, Convection etc Methods
42     // --------------------------------
43     val_type Diffusion(val_type x, val_type t) const;
44     val_type Convection(val_type x, val_type t) const;
45     val_type Reaction(val_type x, val_type t) const;
46     val_type Inhomog(val_type x, val_type t) const; // Forcing Term (RHS)
47
48     // Source Term Initial and Boundary Conditions
49     // -----------------------------------------
50     val_type inLftBnd(val_type t) const;
51     val_type inRhtBnd(val_type t) const;
52     val_type inIntCnd(val_type x) const;
53
54     // Source Term D/C/R Method
55     // -----------------------
56     val_type inDiffusion(val_type x, val_type t) const;
57     val_type inConvection(val_type x, val_type t) const;
58     val_type inReaction(val_type x, val_type t) const;
59
60     // PDE Signature (Allows ibvpSolver to identify which PDE is being used).
61     int Signature() const;
62     val_type Underlying() const;
63     val_type DiffSig(val_type x, val_type t) const;
64 };
```

FIGURE 5.5: C++ Black-Scholes Class hpp file.

Before the *IBvp* class object is instantiated, two instances of the '*Range*' template class needs to be defined for both the spatial and time interval. As the *Range* class contains the interval of the truncated (or transformed) domain, passing them into the *IBvp* class provides it with the necessary information

of the PDE and the bounded domain in which it is constrained too.

Initial Boundary Value Problem Class

Figure 5.2 has been expanded upon concerning the bridge design of the code. However, as a class, it is a cumulation of information that contains the parameters required to construct a finite-difference mesh and call any PDE methods. This is accomplished by passing the *OneFactorIBvp* pointer and the range template classes (by reference) into the class constructor. Then, via the bridge design pattern if the *IBvp* class is passed to another class, so are all the definer section methods from the PDE and Range classes.

5.1.2 C++ Solver Code Section

The solver section of the C++ code consists of passing the *IBvp* class and final set of mesh information (the number of spatial/time steps) into a numerical method and thus calculate the solution of the PDE in question as shown in 5.3. Notably, this section of code follows from an implementation in Duffy's (2018) book [10, pg.680-85].

Firstly, consider the *IBvpSolver* class header file extract:

```
14 class IBvpSolver
15 {
16 protected:
17     void initMesh(long NSteps, long JSteps);
18     void initIC();
19
20 public:
21     IBvpSolver& operator = (const IBvpSolver& source) = delete;
22
23     IBvpSolver();
24     IBvpSolver(IBvp& source, long NSteps, long JSteps);
25     virtual ~IBvpSolver();
26
27     // Virtual Function used to approximate
28     virtual vec& result();
29
30     // Iterates along the x-axis inside the result t-axis loop
31     virtual void calculate() = 0;
32
33     // Array of x values
34     const vec& XValues() const;
35     // Array of t values
36     const vec& TValues() const;
37
38 protected:
39     IBvp* ibvp;
```

FIGURE 5.6: C++ IBvpSolver hpp file.

The extract contains all the methods used within the *IBvpSolver* class file and demonstrates the bridge pattern design through the inclusion of the *IBvp* class pointer that has been passed into the class. Methods such as *initMesh* (line 17) and *initIC* (line 18) are called upon initialisation via the constructor, and this creates the finite difference mesh and vector required to compute the

solution. *IBvpSolver* contains the virtual function *calculate* which is overridden by the numerical methods through inheritance. The numerical method classes that inherit from *IBvpSolver* gain access to protected/public methods and variables within the class, providing access to methods from the definer section of the code.

Consider the Crank Nicolson header file extract in figure 5.7. Line 31 contains the constructor that takes in a pointer to the *IBvp* class and the number of steps for both the spatial and time domains. The constructor's main function (within the cpp file) is the instantiation of the *IBvpSolver* class in which the *IBvp* class is passed alongside the number of steps associated with each domain.

```
9   class CNIBVP : public IBvpSolver
10  {
11  private:
12
13      // Notice that we store the data that 'belongs' to
14      // this class. It is private and will not pollute the
15      // other classes.
16      vec A, B, C;        // Lower, diagonal, upper
17      vec F;              // Right-hand side of matrix
18
19      // Additional Vectors to calculate source term components.
20      vec inA, inB, inC;
21      vec inF;
22
23      // Thomee Box Scheme Components
24      val_type t1, t2, t3, ptvLam, ntvLam, Lam;
25
26      // 2X2 Matrix Diagonal Components
27      vec Ldiag, Mdiag, Udiag;
28
29  public:
30      CNIBVP();
31      CNIBVP(IBvp& source, long NSteps, long JSteps);
32      ~CNIBVP();
33
34      void calculate();
35
36  private:
37      val_type InhomogTerm(val_type x, val_type t, int i) const;
38      void CrankInhomog();
39
40      val_type NearField();
41      void ThomeeScheme();
42
43  private:
44      // PDE type check
45      std::vector<int> typecheck = { 0, 1, 2, 3, 4 };
46
47
48  };
```

FIGURE 5.7: C++ Crank Nicolson Method hpp file.

Through the creation of a Crank Nicolson class object, an instance of the *IBvpSolver* class is created via its constructor (fig D.1, line 15) resulting in a full FDM mesh without any further user defined code. Finally the virtual method *result(.)* is called by the user to obtain the approximation to the PDE. Consider the following option pricing example:

```
66 int main()
67 {
68
69     Option myOption;
70     myOption.K = 65.0;
71     myOption.sig = 0.3;
72     myOption.T = 0.25;
73     myOption.r = 0.08;
74     myOption.b = 0.08;
75
76     myOption.SMax = 100.0;
77     myOption.type = 'P';
78
79     myOption.Target = 60;
80
81     PdeBlackScholes PBS(myOption);
82     Range<double> rangeX(0.0, myOption.SMax);        // Space Interval
83     Range<double> rangeT(0.0, myOption.T);           // Time Interval
84     IBvp currentImpBS(PBS, rangeX, rangeT);          // IBvp Class Instance
85
86     long J = static_cast<long> (rangeX.spread());
87     long N = 1000;
88
89     CNIBVP bsCN(currentImpBS, N, J);                 // CNIBVP Instance
90     vec bCN = bsCN.result('C');                      // Results Vector
```

FIGURE 5.8: C++ Option Pricing Example.

5.2 C++ Numerical Method Implementation

The following section will provide details on the function of each numerical method using the classic Black-Scholes PDE to explain such procedures. Firstly, the calculation section of the *IBvpSolver* class will be expanded on.

```
86  vec& IBvpSolver::result()
87  {
88
89      for (std::size_t n = 0; n < tarr.size(); ++n)
90      {
91          tnow = tarr[n];
92
93          calculate();
94
95          tprev = tnow;
96
97          // Vector captures each time step along the T mesh collecting all the
98          // option prices from t = 0 to t = T. Used to calculate Greek Theta.
99          ThetaResultHold.push_back(vecOld[UnderlyingTarget]);
100
101         for (std::size_t j = 0; j < vecNew.size(); ++j)
102         {
103             vecOld[j] = vecNew[j];
104             inhomogOld[j] = inhomogNew[j];
105         }
106     }
107     return vecNew;
108 }
```

FIGURE 5.9: C++ IBvp Results Method.

Upon calling the *results* method in *IBvpSolver* (via inheritance), it utilises a set of solution vectors and mesh parameters created upon initialisation. The method effectively iterates along the user-defined time interval at a specified

step (k) given by *tarr.size()*. The loop iterates until it reaches its expiry. Within the loop, *calculate()* is activated, calling the calculate methods in either the Crank Nicolson or ADE class. *calculate* then iterates along the spatial interval for each time step and approximates the solution of an option/bond or sensitivity (via CSE).

Upon approximating the solution at a given time step, a vector denoted *vecNew* is assigned the values associated with the specific time step approximation. This vector is copied via the iterative method which is based on Duffy's implementation of FDM schemes in C++ (see Duffy (2019) [10, pg.682]) however, has two modifications allowing the *IBvpSolver* class to deal with the calculation of the Theta sensitivity (line 99) and PDEs containing inhomogeneous terms (line 104).

Line 99 contains *ThetaResultHold* and alleviates one of the issues associated with calculating time-related sensitivities via Divided Difference and the Cubic Spline method. As time is forward moving and an independent variable, the approximations of a specific target (i.e. Theta for an underlying price of 60) needs to be captured instead of all possible spatial interval values (figure 5.10 provides a visual representation). The resulting vector can then be analysed to obtain an approximation of the theta sensitivity without the cost of storing a large vector full of unused values.

Line 104 lastly contains the inhomogeneous source term solution vectors *inhomogOld* and *inhomogNew*. These vectors are initialised when an instance of the *IBvp* class is created, and the boundary conditions / initial condition are applied to the vector. They operate in the same manner as *vecOld* and *vecNew* because, after each spatial iteration, the approximation values are stored and passed to the 'old' vector for use in the next iteration. After all, time iterations are complete *vecNew* is returned containing the option, bond or sensitivity approximation values.

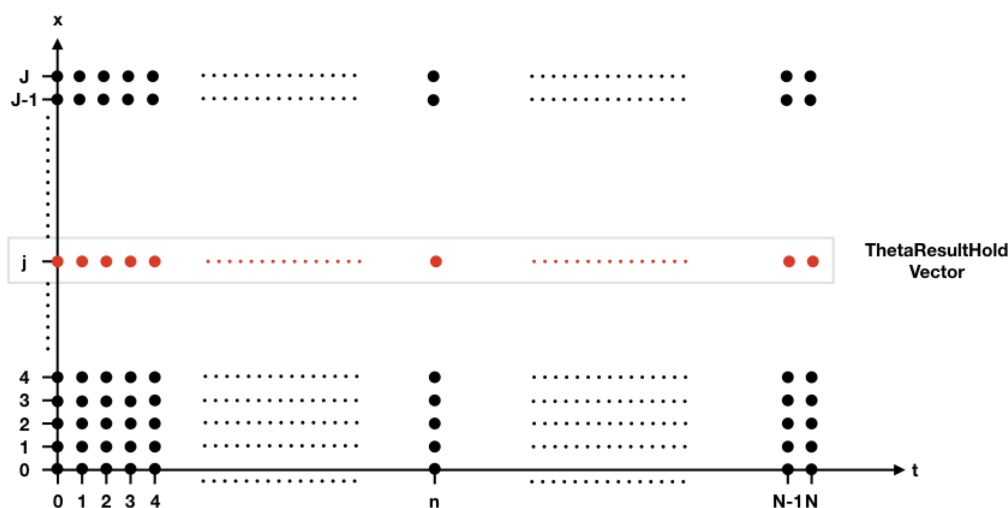

FIGURE 5.10: ThetaResultHold Vector Process.

5.2.1 C++ Crank Nicolson

The Crank Nicolson extract in figure 5.7 will now be expanded upon. Apart from its function, the layout of the method is identical to the ADE class, consider the following code extract (the full code is given in Appendix D.1):

```
123 void CNIBVP::calculate()
124 { // Tells how to calculate sol. at n+1
125
126     // If there exists a source term in the given PDE which is option price based.
127     // CN will be applied to the standard BlackScholes equation to produce a vector
128     // of option prices to calculate the RHS.
129     val_type RHS = ibvp->Inhomog(ibvp->xrange().high(), ibvp->trange().high());
130     if (RHS != 0)
131     {
132         CrankInhomog();
133     }
134
135     double t1, t2, t3, Low, Mid, Upp;
136
137     for (std::size_t i = 1; i < F.size() - 1; ++i)
138     {
139         t1 = (0.5 * k * ibvp->Diffusion(xarr[i], tnow));
140         t2 = 0.25 * k * h * ibvp->Convection(xarr[i], tnow);
141         t3 = 0.5 * k * h2 * ibvp->Reaction(xarr[i], tnow);
142
143         // Coefficients of the U terms
144         A[i] = t1 - t2;                 // Lower Diagonal
145         B[i] = -h2 - 2.0 * t1 + t3;     // Domin Diagonal
146         C[i] = t1 + t2;                 // Upper Diagonal
147
148         // Coefficients of the U terms
149         double t1A = 0.5 * k * ibvp->Diffusion(xarr[i], tprev);
150         double t2A = 0.25 * k * h * ibvp->Convection(xarr[i], tprev);
151         double t3A = 0.5 * k * h2 * ibvp->Reaction(xarr[i], tprev);
152
153         Low = -t1A + t2A;
154         Mid = -h2 + 2.0 * t1A - t3A;
155         Upp = -t1A - t2A;
156
157         // Approximate Option Price wrt the source term.
158         F[i] = Low * vecOld[i - 1] + Mid * vecOld[i] + Upp * vecOld[i + 1]
159             + 0.5 * k * h2 * InhomogTerm(xarr[i], tnow, i);
160
161     }
162
163     // Define boundary conditions
164     val_type BCL;
165     val_type BCR = ibvp->RhtBnd(tnow);
166
167     // If statement to calculate boundary condition if CIR process is in use.
168     int Typ = ibvp->Signature();
169     if (Typ == 8)
170     {   BCL = NearField(); }
171     else
172     {   BCL = ibvp->LftBnd(tnow); }
173
174     // Create option template for the double sweep tridiagonal matrix solver
175     DoubleSweep<double> mySolver(A, B, C, F, BCL, BCR);
176
177     vecNew = mySolver.solve();
178 }
```

FIGURE 5.11: Crank Nicolson calculate method.

The above extract of the *calculate* method immediately draws similarities to 3.7 and 3.12 as each implementation was made under the intent of keeping

the class applicable for any d/c/r equation. Calling *calculate()* results in the following actions:

- The identification of any possible source (inhomogeneous) terms (line 129).
- The application of the Crank Nicolson Method (3.2.2).
- Definition, calculation and implementation of the boundary conditions.
- The application of the Double Sweep tridiagonal solver (see Duffy (2018) [10, pg.396]).

Beginning with the first action, the implementation of an IF statement is to prevent unnecessary calculations. If the *Inhomog* term located in the PDE class of a d/c/r equation is not equal to zero, then a secondary spatial loop will begin that iterates on an FDM mesh using the boundary conditions/initial condition of the source term. In the case of a Black-Scholes CSE, the loop would approximate the option price in parallel with the approximation of the sensitivity.

The second action involves standard Crank Nicolson iteration that implements the algorithm from chapter 3.2 to approximate the bond price with the addition of the source term that was calculated in parallel (line 159). The third action involves the determination of the near-field boundary condition that depends on the PDE signature. This specifies whether the Thomée scheme is required in the algorithm to approximate the boundary condition PDE as described in 3.2.3.

The section of code involving the Thomée method implements itself by merely implementing the box quotient terms $(1+\lambda)$ and $(1-\lambda)$ into the Crank Nicolson matrix resulting in one additional unknown term in the solution vector of the equation system. However, in order to maintain compatibility with the double sweep tridiagonal solver, a 2x2 matrix is first solved to calculate the boundary conditions which is implemented into the tridiagonal solver in the final action of the method (see figure 5.12).

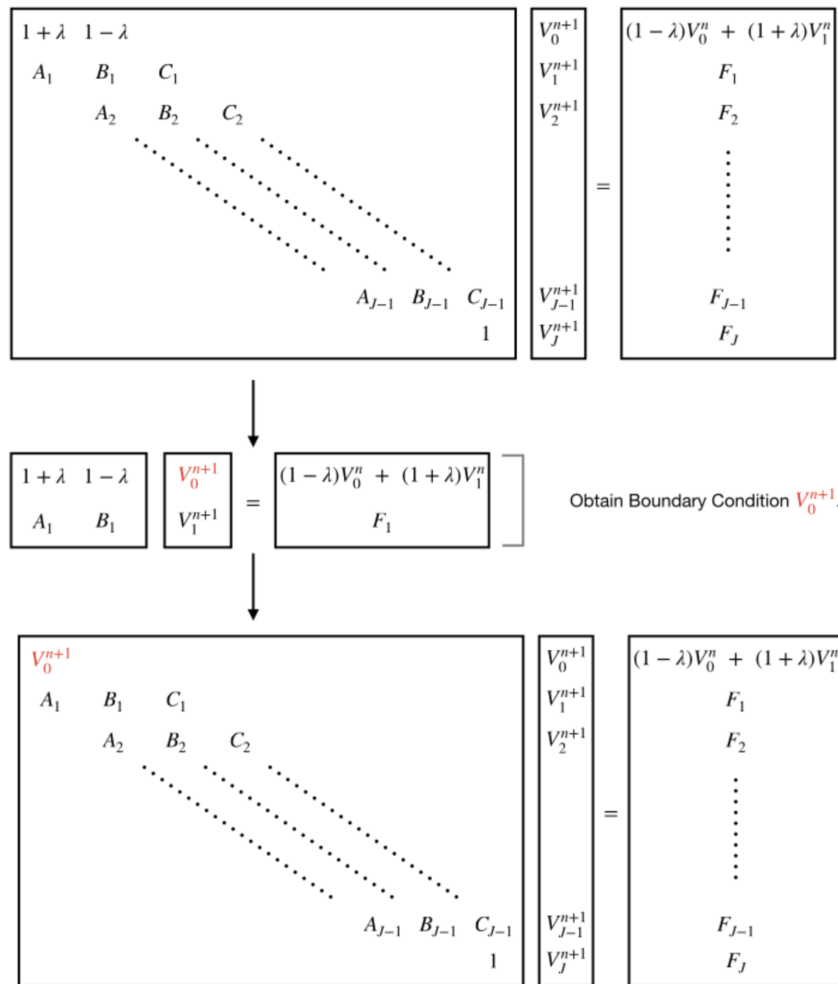

FIGURE 5.12: Implementation of the Thomée Scheme into the Crank Nicolson Class.

Note that the above method is implemented into C++ for compatibility purposes. The application of the Thomée method is implemented into MATLAB and only requires the built-in inverse 'backslash' command which although is basic, provides a small enough computation time to make it a much more feasible solution than the above. See Appendix E.1 for the full Crank Nicolson MATLAB code that is specific to solving the CIR zero-coupon bond pricing PDE.

5.2.2 C++ ADE Implementation

Following a close layout with figure 5.11, figure 5.13 uses the same code to identify the requirement of a source term and thus calculate the option/bond price in parallel with the CSE approximation. The difference within this method stems from the requirement of two loops to calculate the upwind and downwind sweeps. These work in parallel in opposite directions along the spatial axis. Lastly, on line 222, a loop is used to average the upward and downward vectors where the approximation is stored in *vecNew*. Aside

from *calculate()*, the ADE class bears a close resemblance to the previously expanded Crank Nicolson class. In addition to the ADE C++ implementation, the full code has been developed within MATLAB with an alternative CN implementation ([17]) and is given in E.2.

```
160 // Main method used to calculate upward and downward sweep for the B&C ADE method.
161 // Where the average is then taken producing a vector of approximations.
162
163 void ADE_BC::calculate()
164 {
165     // Implementation of possible inhomogeneous term.
166     val_type RHS = ibvp->Inhomog(ibvp->xrange().high(), ibvp->trange().high());
167     if (RHS != 0)
168     {
169         BCInhomog();
170     }
171
172     // Implementation of boundary value solver if the PDE is a CIR Zero Coupon PDE.
173     int Typ = ibvp->Signature();
174     if (Typ == 8) // CIR PDE
175     {
176         // Solve CIR PDE Boundary Condition
177         val_type NearBnd = cirNearField();     // Using Thomee/Box Method
178         U[0] = NearBnd;
179         V[0] = NearBnd;
180     }
181     else // Case in which no PDE needs to be solved to obtain boundary conditions.
182     {
183         U[0] = ibvp->LftBnd(tnow);
184         V[0] = ibvp->LftBnd(tnow);
185     }
186
187     // Set up boundary conditions and initial conditions of the given PDE.
188     U[U.size() - 1] = ibvp->RhtBnd(tnow);
189     V[V.size() - 1] = ibvp->RhtBnd(tnow);
190
191     for (std::size_t j = 1; j < U.size() - 1; ++j)
192     {
193         // Populate vectors
194         UOld[j] = VOld[j] = vecOld[j];
195     }
196     double t1, t2, t3, t4;
197
198     // Compute Upward Sweep (formulae in book) from j = 1, ... , J-1
199     for (std::size_t j = 1; j < U.size() - 1; ++j)
200     {
201         auto xval = xarr[j];
202         t1 = (k / h2) * ibvp->Diffusion(xval, tnow);
203         t2 = 0.5 * (k / h) * ibvp->Convection(xval, tnow);
204         t3 = (1.0 + t1 - k * ibvp->Reaction(xval, tnow));
205         t4 = -k * InhomogTerm(xval, tnow, j);
206         U[j] = ((t1 - t2) * U[j - 1] + (1.0 - t1) * UOld[j] +
207             (t1 + t2) * UOld[j + 1] + t4) / t3;
208     }
209
210     // Compute Downward Sweep from j = J-1, ... , 1 (J-2 as BCs are applied).
211     for (std::size_t j = V.size() - 2; j >= 1; --j)
212     {
213         auto xval = xarr[j];
214         t1 = (k / h2) * ibvp->Diffusion(xval, tnow);
215         t2 = 0.5 * (k / h) * ibvp->Convection(xval, tnow);
216         t3 = (1.0 + t1 - k * ibvp->Reaction(xval, tnow));
217         t4 = -k * InhomogTerm(xval, tnow, j);
218         V[j] = ((t1 - t2) * VOld[j - 1] + (1.0 - t1) * VOld[j] +
219             (t1 + t2) * V[j + 1] + t4) / t3;
220     }
221
222     for (std::size_t j = 0; j < vecNew.size(); ++j)
223     {
224         vecNew[j] = 0.5 * (U[j] + V[j]);
225     }
226 }
```

FIGURE 5.13: Alternating Direction Explicit calculate method.

5.2.3 C++ Method Of Lines Implementation

Section 5.1 alluded to a separation of the Method Of Lines scheme that does not inherit from the *IBvpSolver* class but requires a secondary 'bridge' implementation from the *IBvp* class. The reasons for such a design will be expanded upon within this subsection.

Boost is an open source library that adds additional functionality to C++ whilst working in conjunction with the STL library (see Ahnert and Mulansky (2015) [2]). The Method Of Lines scheme uses Odeint integrate functions [2] where the RKDP method is utilised (3.4.1). Consider the syntax of the boost function:

integrate(stepper , system , x0 , t0 , t1 , dt , observer)

The above parameters of the function carry out the following:

Stepper :	Defines how the integration process is to be carried out.
System :	Implements the system of ODEs and calculates the time derivative of each ODE.
x0 :	Initial Condition.
t0, t1 :	Initial and Final times for the integration process.
dt :	Time Step.
Observer :	Called at every time step and prints the current approximation.

The above boost function takes in a system variable that contains the necessary steps to discretise a PDE into a system of ODEs as explained in 3.4.1. The stepper can take in simple functions that return ODEs such as:

$$\frac{\partial x}{\partial t} = -2x.$$

However, to solve d/c/r equations, functors are required (i.e classes/structs that 'act' like functions). Consider the Method Of Lines C++ declaration extract in figure 5.14 (full code is located in Appendix D.2).

```
23 #include <iostream>
24 #include <boost/numeric/odeint.hpp>
25 #include <boost/numeric/ublas/vector.hpp>
26 #include <boost/numeric/ublas/matrix.hpp>
27 #include <boost/numeric/ublas/io.hpp>
28
29 #include "ibvp.hpp"
30
31 using val_type = double;
32 typedef std::vector<val_type> state_vec;
33 typedef boost::numeric::ublas::matrix<val_type> state_mat;
34
35 enum bnd { Lft, Rht, Inner };
36
37 class MethodOfLines
38 {
39 public:
40     // Default Constructor
41     MethodOfLines(IBvp& source, long NSteps, long JSteps);
42
43     ~MethodOfLines();
44
45     // Vector System
46     void operator()(const state_vec& U, state_vec& dUdt, const val_type t);
47     // Matrix System
48     void operator()(const state_mat& U, state_mat& dUdt, const val_type t);
49
50     inline val_type InhomogTerm(val_type x, val_type t, std::size_t i,
51         bnd bound, state_mat U) const;
52
53     int PDEtype() const;
54
55     // Initial Condition Methods (vector and matrix)
56     state_vec IntCnd();
57     state_mat MatIntCnd();
58
59 protected:
60     IBvp* ibvp;
61     long N;
62     long J;
63
64     state_vec xarr;
65     state_vec tarr;
66     state_vec inhomog;
67
68 public:
69
70     val_type h, h2, hm1;
71     val_type k;
72
73     val_type T;
74     val_type T0;
75 };
```

FIGURE 5.14: Method Of Lines header file.

As the Method Of Lines scheme reduces the PDE to a system of ODEs, a finite-difference mesh no longer needs to be constructed. Instead, the number of steps in the spatial and time directions need to be defined that are used in the integration process. As with the previous methods, the code is designed to calculate d/c/r equations that contain inhomogeneous source terms. For that reason, two operator overloads are implemented in lines 46 and 48. Each overload varies by its input as a standard application of the scheme requires only a vector to approximate a solution. However, the inclusion of a source

term implies that two approximations need to be calculated in parallel leading to the requirement of the uBLAS matrix (another feature of the boost library) that is compatible with the *integrate* function. Despite being independent of *IBvpSolver*, the class shares a near-identical *InhomogTerm* method used to supply correct inhomogeneous components to the discretised PDE depending the signature of the PDE.

The limitation of using the boost library is that it is difficult to implement into a specific code design without sufficient knowledge of the boost library and careful planning. This is clear since the *MethodOfLines* class is only a component of the final calculation (i.e the system). The additional components are defined in the *main.cpp* section of code within the following function that performs the approximation:

```
546 vec MethodOfLinesResult(MethodOfLines source)
547 {
548     int type = source.PDEtype();
549     if (type == 0)
550     {
551         state_vec U = source.IntCnd();
552         std::size_t steps = Bode::integrate(source, U, source.T0, source.T, source.k);
553         return U;
554     }
555     else
556     {
557         state_mat U = source.MatIntCnd();
558         std::size_t steps = Bode::integrate(source, U, source.T0, source.T, source.k);
559
560         // Store results inside vector and return.
561         vec sensitivity(U.size1());
562         for (std::size_t j = 0; j < sensitivity.size(); ++j)
563         { sensitivity[j] = U(j, 1); }
564         return sensitivity;
565     }
566 }
```

FIGURE 5.15: Method Of Lines *main.cpp* function.

Upon inspection, the function requires an input of the *MethodOfLines* class passed through where the PDE is identified as having either a source term or not. Depending on the type of PDE, the vector or matrix overload is passed into *integrate* with an initial vector constructed in the *MethodOfLines* class, and its time integration parameters. An additional limitation reveals the difficulty of implementing methods such as the Thomée scheme without interfering with the strict nature of the Odeint integration functions.

5.3 C++ Implementation: Approximation Methods

Before proceeding with the C++ implementation of the approximation methods used to calculate sensitivities, it is worth noting that the previous methods from 5.2 are solely used to approximate the sensitivities of the CSE equations (4.5). Therefore, the approximation of sensitivities using CSEs will not be covered further in this chapter.

5.3.1 Divided Difference

The implementation of the Divided difference method into C++ is a relatively straight forward process. 5.16 presents the procedure of using the Divided Difference.

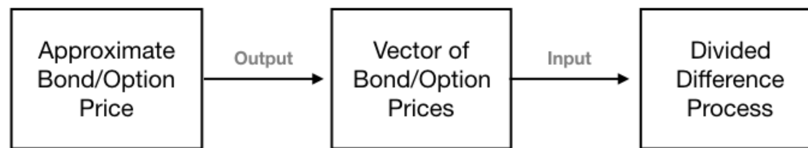

FIGURE 5.16: A Simplified diagram to explain the procedure of calculating Divided Difference sensitivity approximation.

Due to the simplistic nature of Divided Difference and the minimal code required to compute such approximation. The function code implemented into *main.cpp* is located in figure 5.17. Line 427 is the function definition and requires the inputs of a target (i.e the sensitivity of an option with an underlying of £60); a vector *xtarr*, the mesh created in *IBvpSolver* or *MethodOfLines* and lastly, the solution vector returned from the approximation methods discussed in 5.2.

The function code is designed to calculate both first and second-order approximations of sensitivities where a Cubic Spline object is instantiated to obtain specific approximations (via interpolation) based upon the user-defined target. As alluded too in figure 5.10, the calculation of time-related sensitivities, such as Theta requires special treatment given the awkward nature of approximating the sensitivity relating to the independent variable. Line 482-493 accomplishes such an approximation using the backwards-difference quotient as the direction of time is known. Upon performing the Divided Difference calculation, the Cubic Spline interpolation is used to return an approximation of the sensitivity.

```
437 void DividedDifference(double target, vec xtarr, vec solution,
438     Type sensitivity_order)
439 {
440     // Obtain new interval.
441     vec zarr(xtarr.size() - 2);
442 
443     // Populate new interval container.
444     for (std::size_t j = 0; j < zarr.size(); ++j) { zarr[j] = xtarr[j + 1]; }
445 
446     // Identify Step Size.
447     double h = zarr[1] - zarr[0];
448 
449     // Create Instance of Spline Class.
450     tk::spline DDspline;
451 
452     if (sensitivity_order == first)
453     {
454         // Create Target Vector Location
455         int target_step = static_cast<int>(target / h);
456 
457         // Create solution vector
458         vec DivDif(zarr.size());
459         for (std::size_t j = 0; j < zarr.size(); ++j)
460         {
461             // Perform Divided Difference
462             DivDif[j] = (solution[j + 1] - solution[j - 1]) / (2 * h);
463         }
464         DDspline.set_points(xtarr, DivDif);
465         printf("First Order Divided Difference at %f is %f\n", target,
466             DDspline(target));
467     }
468     else if (sensitivity_order == second)
469     {
470         // Create Target Vector Location
471         int target_step = static_cast<int>(target / h);
472         vec DivDif(zarr.size());
473         for (std::size_t j = 0; j < zarr.size(); ++j)
474         {
475             DivDif[j] = (solution[j + 1] - 2 * solution[j] + solution[j - 1])
476                 / (h * h);
477         }
478         DDspline.set_points(xtarr, DivDif);
479         printf("Second Order Divided Difference at %f is %f\n", target,
480             DDspline(target));
481     }
482     else if (sensitivity_order == theta)
483     {
484         vec ForDif(solution.size());
485         double k = xtarr[1] - xtarr[0];
486 
487         for (std::size_t j = 1; j < std::size(ForDif); ++j)
488         {
489             ForDif[j] = -(solution[j] - solution[j - 1]) / k;
490         }
491         std::cout << "S = " << static_cast<int>(target) << std::setprecision(12)
492             << ", forward sensitivity:     " << ForDif[xtarr.size()-2] << std::endl;
493     }
494 }
```

FIGURE 5.17: Divided Difference *main.cpp* function.

5.3.2 Cubic Spline Interpolation

The implementation of the Cubic Spline interpolation method follows a close resemblance to the implementation of the Divided Difference:

```
496 void CubicSpline(int target, vec xtarr, vec solution, Type derivative_order)
497 {
498     // Define new interval (Cubic, therefore N-2)
499     vec zarr(xtarr.size() - 2);
500     // Populate new vector
501     for (std::size_t j = 0; j < zarr.size(); ++j) { zarr[j] = xtarr[j + 1]; }
502
503     if (derivative_order == first)
504     {
505         // Integer identifies the target value to display the solution for.
506         int target_step = target / (zarr[1] - zarr[0]);
507
508         // Create solution vector
509         vec spline(zarr.size());
510
511         CubicSplineInterpolator CSI(xtarr, solution, SecondDeriv);
512         for (std::size_t j = 0; j < zarr.size(); ++j)
513         {
514             spline[j] = CSI.Derivative(zarr[j]);
515         }
516         std::cout << std::setprecision(12) << std::fixed << "S = " << target
517             << ", Cubic Spline: " << spline[target_step-1] << std::endl;
518     }
519     else if (derivative_order == second)
520     {
521         int target_step = target / (zarr[1] - zarr[0]);
522
523         // Create solution vector
524         vec spline(zarr.size());
525
526         CubicSplineInterpolator CSI(xtarr, solution, SecondDeriv);
527         for (std::size_t j = 0; j < zarr.size(); ++j)
528         {
529             spline[j] = CSI.SecondDerivative(zarr[j]);
530         }
531         std::cout << std::setprecision(12) << std::fixed << "S = " << target
532             << ", Cubic Spline: " << spline[target_step-1] << std::endl;
533     }
534     else if (derivative_order == theta)
535     {
536         // Identifies target step (theta range is multiplied by 1.2 to avoid oscillations).
537         int target_step = (solution.size() - 1) / 1.2;
538
539         // Create solution vector
540         vec spline(xtarr.size());
541
542         // Relates to the time decay, therefore is defined on the time interval.
543         CubicSplineInterpolator CSI(xtarr, solution, SecondDeriv);
544         for (std::size_t j = 0; j < xtarr.size(); ++j)
545         {
546             spline[j] = CSI.Derivative(xtarr[j]);
547         }
548         std::cout << std::setprecision(12) << std::fixed << "S = " << target
549             << ", Cubic Spline: " << -spline[target_step] << std::endl;
550     }
551 }
```

FIGURE 5.18: Cubic Spline *main.cpp* function.

Figure 5.18 contains a function used to calculate the sensitivity of a d/c/r PDE via the use of the *CubicSplineInterpolation* class. This applies the formulae presented in chapter 4.2 to obtain approximations of the first and second derivative (the reader is referred to Duffy (2018) [10, pg.418-20]). The above

function contains three cases of cubic spline interpolation, the first and second derivative use the standard reduced spatial axis interval to obtain an interpolated function. However, the final case approximates Theta (time sensitivities) via the uses of an extended time domain which was created to avoid the oscillations associated with the 'endpoints' of the interpolated function.

The range adjustment brought about a limitation of calculating time-related sensitivities as it requires a secondary adjusted range PDE object. As a result, several issues are apparent, such as the identification of the target sensitivity (e.g. Theta for an underlying price of £60) given the change in the domain of the PDE. The solution is to multiply the range by a factor of 1.2 and adjust the parameters accordingly. Finally, as a matter of preference, the time-sensitivity approximation is represented as a negative number to indicate time decay.

5.3.3 Forward Automatic Differentiation

Forward Automatic Differentiation is implemented through the use of Pulvers (2019) *autodiff* library [19], which takes advantage of the Boost open source library. Within this thesis, two classes are created to implement the Forward AD method, one for the Black-Scholes option price and the second for the CIR bond price which will be expanded on (full Black-Scholes code is given in Appendix D.3).

Figure 5.19 contains two main components: the template boost method and the calculation method. The calculation method begins by creating a tuple of four maximum derivative variables equal to their sensitivity values (Line 19). As this thesis is interested in calculating Convexity, the second derivative of the interest rate, the associated variable is given the value of two.

Lines 21-24 assign the tuple values to their associated variable names producing a compatible form for Forward AD. Lines 47-60 contain the template method which performs the Forward AD calculation (using the *autodiff.hpp* file). Similarly to the *make_ftuple* method, *promote* requires the tuple of sensitivity variables and the closed-form solution of the bond price. Upon creating the template the *.derivative* function performs Forward AD depending on the tuple value given. Consider line 39, Convexity is calculated by inputting a tuple value of 2, informing the derivative method that the second derivative of the bond pricing formula concerning the interest rate needs to be calculated.

The advantage of the method comes from its ease of implementation. It alludes too in 4.4 that the low adoption of the method is due to the difficulty in coding. However, the implementation of FAD, for thesis shows a relatively straight forward code given the use of external AD libraries.

```cpp
#include <boost/math/differentiation/autodiff.hpp>
using namespace boost::math::differentiation;

class CirForwardAD
{
public:
    CirForwardAD(double a, const double rate, const double sigma, const double tau,
        const double soa) : variable(a), target(rate)
    {
        Calculate(a, rate, sigma, tau, soa);
    }


    void Calculate(double a, const double rate, const double sigma, const double tau, const double soa)
    {
        auto const variables = make_ftuple<double, 2, 1, 1, 1>(rate, sigma, tau, soa);

        auto const& r = std::get<0>(variables);
        auto const& sig = std::get<1>(variables);
        auto const& T = std::get<2>(variables);
        auto const& SoA = std::get<3>(variables);

        auto output = [](auto target, auto sens1, auto sens2, auto sens3,
            auto sens4, auto sens5)
        {
            std::cout << std::setprecision(std::numeric_limits<double>::digits10)
                << "r = " << target << ", Duration:   " << sens1 << "\n"
                << "r = " << target << ", Convexity:  " << sens2 << "\n"
                << "r = " << target << ", Expiry:     " << sens3 << "\n"
                << "r = " << target << ", Speed Adj:  " << sens4 << "\n"
                << "r = " << target << ", Volatility: " << sens5 << "\n";
        };

        auto const bond_price = CirBondPrice(a, r, sig, T, SoA);
        double const Duration = bond_price.derivative(1, 0, 0, 0);
        double const Convexity = bond_price.derivative(2, 0, 0, 0);
        double const Expiry = bond_price.derivative(0, 0, 1, 0);
        double const SpeedAdj = bond_price.derivative(0, 0, 0, 1);
        double const Volatility = bond_price.derivative(0, 1, 0, 0);
        output(rate, Duration, Convexity, Expiry, SpeedAdj, Volatility);
    }

private:
    template<typename Rate, typename Sigma, typename Tau, typename SoA>
    promote<Rate, Sigma, Tau, SoA> CirBondPrice(double a, Rate const& r, Sigma const& sigma,
        Tau const& tau, SoA const& b)
    {
        auto const sqr = sqrt(b * b + 2.0 * sigma * sigma);
        auto const num1 = 2.0 * sqr * exp(0.5 * tau * (sqr + b));
        auto const num2 = 2.0 * (exp(tau * sqr) - 1.0);
        auto const denom = 2.0 * sqr + (sqr + b) * (exp(tau * sqr) - 1.0);

        auto const A = pow((num1 / denom), ((2.0 * a) / (sigma * sigma)));
        auto const B = num2 / denom;

        return A * exp(-r * B);
    }

private:
    double variable;
    double target;
};
```

FIGURE 5.19: C++ CIR Forward AD Header File

5.3.4 Complex Step Method

The Complex Step Methods implementation in the C++ code is considered awkward due to the requirement of templates that have many variations of the same equation. Consider the case of pricing a CIR zero-coupon bond; the C++ CIR template is given in figure 5.20 and holds a case for each of the four sensitivities covered within this thesis. Depending on the order of sensitivity required, the bond pricing function is altered to allow for the insertion of a complex number into the area that requires analysis. For example, if Duration is to be calculated, a complex number is inserted into all locations where there exists the interest rate variable. Therefore when the imaginary part of the difference is taken, it will be with respect to the interest rate variable, yielding an approximation.

A visible limitation is the bulk of code required to account for each sensitivity. After the creation of the CIR bond pricing template function, the sensitivities are calculated via the set of function given in figure 5.21. In each sensitivity approximation, the template is called, taking in a complex number containing a step and initial value alongside the sensitivities associated enum. Despite the clunky nature of the code, the amount required to obtain a sensitivity approximation is little compared to other implementations. A future area of work would be to implement the process more straightforwardly, possibly through the use of template programming.

Lastly, attention needs to be drawn to the issue of applying the method to the Black-Scholes closed-form solution. The existence of a Gauss error function makes coding the CSM difficult as the standard library *erf()* is not extended to the complex plane. Therefore, the Faddeva open-source C++ wrapper is required as it provides the necessary complex functions to calculate the complex error function.

```
109 template <typename Argu> Argu CirZeroBond(const Argu& z, Bond type)
110 {
111     std::vector<Argu> sol;
112
113     if (type == short_rate)
114     {
115         Argu gamma = std::sqrt(kappa*kappa + 2.0 * volatility * volatility);
116         Argu denom = 2.0 * gamma + (gamma + kappa) * (std::exp(gamma * Exp) - 1.0);
117         Argu b = (2.0 * (std::exp(gamma * Exp) - 1.0)) / denom;
118         Argu a = std::pow(((2.0 * gamma * std::exp(0.5 * (gamma + kappa) * Exp)) / denom),
119             ((2.0 * kappa * theta) / (volatility * volatility)));
120         Argu B = a * std::exp(-b * z);
121         return B;
122     }
123     else if (type == maturity)
124     {
125         Argu gamma = std::sqrt(kappa * kappa + 2.0 * volatility * volatility);
126         Argu denom = 2.0 * gamma + (gamma + kappa) * (std::exp(gamma * z) - 1.0);
127         Argu b = (2.0 * (std::exp(gamma * z) - 1.0)) / denom;
128         Argu a = std::pow(((2.0 * gamma * std::exp(0.5 * (gamma + kappa) * z)) / denom),
129             ((2.0 * kappa * theta) / (volatility * volatility)));
130         Argu B = a * std::exp(-b * s_rate);
131         return B;
132     }
133     else if (type == vol)
134     {
135         Argu gamma = sqrt(kappa * kappa + 2.0 * z * z);
136         Argu denom = 2.0 * gamma + (gamma + kappa) * (std::exp(gamma * Exp) - 1.0);
137         Argu b = (2.0 * (std::exp(gamma * Exp) - 1.0)) / denom;
138         Argu a = std::pow(((2.0 * gamma * std::exp(0.5 * (gamma + kappa) * Exp)) / denom),
139             ((2.0 * kappa * theta) / (z * z)));
140         Argu B = a * std::exp(-b * s_rate);
141         return B;
142     }
143     else // Speed of Adjustment
144     {
145         Argu gamma = sqrt(z * z + 2.0 * volatility * volatility);
146         Argu denom = 2.0 * gamma + (gamma + z) * (std::exp(gamma * Exp) - 1.0);
147         Argu b = (2.0 * (std::exp(gamma * Exp) - 1.0)) / denom;
148         Argu a = std::pow(((2.0 * gamma * std::exp(0.5 * (gamma + z) * Exp)) / denom),
149             ((2.0 * kappa * theta) / (volatility *volatility)));
150         Argu B = a * std::exp(-b * s_rate);
151         return B;
152     }
153 }
```

FIGURE 5.20: Complex Step Method *main.cpp* CIR template function.

```
228 //==============================================================================
229 // Bond Greeks
230 //==============================================================================
231 
232 double Duration(double x0, double h)
233 {
234     cmplx z(x0, h);
235     auto deriv = std::imag(CirZeroBond<cmplx>(z, short_rate));
236 
237     return deriv / h;
238 }
239 
240 double Convexity(double x0, double h)
241 {
242     // Using the complex step method, the second derivative uses the real part
243     // of the function.
244 
245     cmplx z1(x0, h); // Create an imaginary number using real number and step.
246     cmplx BS_cmplx = CirZeroBond<cmplx>(z1, short_rate);
247     double BS_real = CirZeroBond<double>(x0, short_rate);
248 
249     auto numerator = 2 * (BS_real - BS_cmplx.real());
250     return numerator / (h * h);
251 }
252 
253 double ExpirySen(double x0, double h)
254 {
255     cmplx z(x0, h);
256     auto deriv = std::imag(CirZeroBond<cmplx>(z, maturity));
257 
258     return deriv / h;
259 }
260 
261 double SoaSen(double x0, double h)
262 {
263     cmplx z(x0, h);
264     auto deriv = std::imag(CirZeroBond<cmplx>(z, speed_adjust));
265     return deriv / h;
266 }
267 
268 double VolSen(double x0, double h)
269 {
270     cmplx z(x0, h);
271     auto deriv = std::imag(CirZeroBond<cmplx>(z, vol));
272     return deriv / h;
273 }
```

FIGURE 5.21: Complex Step Method *main.cpp* CIR sensitivity functions.

6 Coding Results

This chapter focuses on the results obtained from the methods covered within this thesis. First consider the parameter values used for each approximation:

Option Parameter Values:

Underlying:	$S = 60$ (Target).
Interest Rate:	$r = 0.08$.
Time to Expiry:	$T = 0.25$.
Strike Price:	$K = 65$.
Volatility:	$\sigma = 0.3$.

Zero Coupon Bond Parameter Values:

Interest Rate:	$r = 0.08$ (Target).
Speed Of Adjustment:	$b\ (\kappa) = 0.08$.
Maturity Time:	$T = 0.25$.
Mean:	$\Theta = 0.6$.
Volatility:	$\sigma = 0.4$.
a:	$\Theta\kappa = 0.048$.

6.1 Closed Form Solutions

Applying the closed form equation in chapter 2 yields the following results for European Options and CIR zero-coupon bonds:

Option Prices:

Put Option:	5.846285626870	Call Option:	2.133371861931

Sensitivities:

Put Delta:	0.627517137751	Call Delta:	0.372482862249
Put Gamma:	0.042042755754	Call Gamma:	0.042042755754
Put Theta:	3.331141320761	Call Theta:	8.428174421956
Put Vega:	11.351544053522	Call Vega:	11.351544053522
Put Rho:	-10.874328472975	Call Rho:	5.053899968259

CIR Bond:
Zero-coupon Bond Price: 0.978966810291096

Sensitivities:
Duration: -0.241911478036699
Convexity: 0.059778495645321
Maturity: 0.087893036824247
Volatility: 1.653066503747697e-04
SoA: 0.002527984560075

6.2 Forward AD, CSM & Closed Form Solutions

The similarities of both Forward Automatic Differentiation and the Complex-Step Method bodes the question regarding which one of the methods is better suited to approximate the sensitivities presented within this paper.

Consider the following results from both Forward Automatic Differentiation and the Complex Step Method for the calculation of option and bond price sensitivities:

Black-Scholes	Forward Automatic Differentiation		Complex Step Method	
Sensitivities	Call	Put	Call	Put
Delta	0.372482797962	-0.627517202038	0.372482797962	-0.627517202038
Gamma	0.042042755754	0.042042755754	0.042043026838	0.042043021509
Theta	-8.428174386737	-3.331141285542	-8.428174386737	-3.331141285542
Rho	5.053899858201	-10.874328583034	5.053899858201	-10.874328583034
Vega	11.351544053522	11.351544053522	11.351544053522	11.351544053522

Black-Scholes	Forward Automatic Differentiation		Complex Step Method	
Sensitivities	Call Error	Put Error	Call Error	Put Error
Delta	2.70339306E-14	2.69784195E-14	0.00000000E+00	0.00000000E+00
Gamma	6.19987815E-07	6.19987815E-07	3.48903854E-07	3.54232925E-07
Theta	1.00438033E-04	1.00438028E-04	1.00438033E-04	1.00438028E-04
Rho	5.49782442E-13	3.42001982E-11	5.40012479E-13	3.42001982E-11
Vega	1.67396678E-04	1.67396678E-04	1.67396678E-04	1.67396678E-04

TABLE 6.4: Black-Scholes Sensitivity FAD and CSM Results

CIR Bond Price Sensitivities	Forward Automatic Differentiation Sensitivity Values	Complex Step Method Sensitivity Values
Duration	-0.241911478037	-0.241911478037
Convexity	0.059778495645	0.059778495487
Expiry	-0.087998279937	-0.087998279937
SoA	0.002527984560	0.002527984560
Volatility	0.000165306650	0.000165306650

CIR Bond Price Sensitivities	Forward Automatic Differentiation Sensitivity Value Errors	Complex Step Method Sensitivity Value Errors
Duration	0.000000E+00	0.000000E+00
Convexity	0.000000E+00	1.583259E-10
Expiry	2.914335E-16	2.914335E-16
SoA	0.000000E+00	0.000000E+00
Volatility	3.359943E-16	4.959954E-16

TABLE 6.5: CIR Bond Sensitivity FAD and CSM Results

Both tables 6.4 and 6.5 produce results that closely match the closed-form sensitivity values, demonstrating the effectiveness of both the Forward AD and CSM method. Each result contains the absolute error, and in each case, results exist that are identical to the exact sensitivity value.

Table 6.4 contains two sets of less accurate sensitivity approximations for both Theta and Vega. A possible solution to increase the accuracy of these sensitivity calculations would be to implement higher-order approximations given in [1]. Through Excel comparisons, it was found that the Complex-Step Method produced more accurate results at a precision higher than 12 decimal places making it a recommended method in the case of the Black-Scholes model. However, the CIR model Duration, Expiry and SoA sensitivities favoured the CSM with the remaining sensitivities favouring the Forward AD method. Overall, the CSM marginally produced more accurate results than the Forward AD method making it the recommended method in the case of this thesis.

6.3 Black-Scholes Numerical Method Approximations

6.3.1 Black-Scholes Option Price

As alluded to in previous methods, the sensitivity calculations require two steps: the approximation of the option price and the subsequent approximation of the option price sensitivities. It is intuitive that a low quality (i.e. inaccurate) option/bond result will result in a poor approximation of the

sensitivity; therefore, an analysis of the bond price approximations must first take place.

Consider table 6.6 containing a set of put option prices calculated using the Crank Nicolson, Alternating Direction Explicit and Method of Lines scheme. The results vary by the interval in which the spatial and time domain has been separated. Through a comparison of the absolute errors it is clear that a change in the time interval does not affect the MOL approximation which is intuitive, thus escalating the number of spatial steps increases the accuracy of the approximation and subsequently reduces the absolute error. For this reason, the MOL produces the 'best' approximation at 1000 time and spatial steps.

Despite this, table 6.6 contains a set of interesting results concerning the performance of the Crank Nicolson and Alternating Direction Explicit in relation to the increase of the number of spatial and time steps. The Crank Nicolson method obtains its most accurate approximation of the put option price when $J = 100$ and $N = 200$; an unexpected result and perhaps an anomaly in the data. ADE behaves in an expected manner by obtaining its most accurate approximation when $J = 500$ and $N = 1000$. Neither the CN or ADE methods achieve their most accurate approximations at 1000 spatial/time steps. A reasonable explanation is when the roundoff error of the computer begins to dominate, and an additional increase in the number of spatial steps will no longer improve the approximation but negatively impact it.

A final comparison in 6.6 between the approximations at each step increment shows that the ADE/MOL produce the most accurate results, each yielding the best result six times. The CN method only obtains the best result four times, making it the worst-performing method within this particular set of data.

Put Option	Spatial Interval (J)	100	200	500	1000
Time Interval (N)	Methods				
100	CN	5.850361245723	5.853532834391	5.854420105830	5.854546831240
	ADE	5.848273678455	5.845459673760	5.807754717854	5.728201747043
	MOL	5.842043564537	5.845223266417	5.846112810890	5.846239861066
200	CN	5.846207650964	5.849383290494	5.850271696838	5.850398584397
	ADE	5.845663176295	5.847326888653	5.837785722436	5.803888094062
	MOL	5.842043564537	5.845223266417	5.846112810890	5.846239861066
500	CN	5.843710462090	5.846888537456	5.847777626234	5.847904611291
	ADE	5.843613212214	5.846548553923	5.845742036718	5.839868764614
	MOL	5.842043564537	5.845223266417	5.846112810890	5.846239861066
1000	CN	5.842877224322	5.846056112542	5.846945429053	5.847072446648
	ADE	5.842848733569	5.845966899100	5.846431327839	5.845044157073
	MOL	5.842043564538	5.845223266417	5.846112810890	5.846239861066

Absolute Error	Spatial Interval (J)	100	200	500	1000
Time Interval (N)	Methods				
100	CN	0.004075618853	0.007247207521	0.008134478960	0.008261204370
	ADE	0.001988051585	0.000825953110	0.038530909016	0.118083879827
	MOL	0.004242062333	0.001062360453	0.000172815980	0.000045765804
200	CN	0.000077975906	0.003097663624	0.003986069968	0.004112957527
	ADE	0.000622450575	0.001041261783	0.008499904434	0.042397532808
	MOL	0.004242062333	0.001062360453	0.000172815980	0.000045765804
500	CN	0.002575164780	0.000602910586	0.001491999364	0.001618984421
	ADE	0.002672414656	0.000262927053	0.000543590152	0.006416862256
	MOL	0.004242062333	0.001062360453	0.000172815980	0.000045765804
1000	CN	0.003408402548	0.000229514328	0.000659802183	0.000786819778
	ADE	0.003436893301	0.000318727770	0.000145700969	0.001241469797
	MOL	0.004242062332	0.001062360453	0.000172815980	0.000045765804

TABLE 6.6: Put Option Price Approximations

6.3.2 Divided Difference Sensitivity Approximation

The following section introduces the approximations obtained from the Divided Difference Method using the ADE, CN and MOL schemes. As explained in section 4.1, the limitation of using Divided Difference is that all parameters which are not explicitly defined in the FDM mesh, parameters such as σ and r, are challenging to approximate and do not produce useful results due to the inherent inaccuracies. Therefore, only the sensitivities related to the state variable and time will only be covered within this section.

6.7 contains a table of sensitivity results and a secondary table of absolute errors for each approximation. The limitations associated with the integrate boost library prevented the approximation of Theta sensitivity using the Method Of Lines scheme. A possible remedy would be to use the observer operator overload methods, but incorporating the same method of extracting approximations for a specific underlying yielded unusable results.

Sensitivity Approximations	Method	Equal Time (N) and Spatial Steps (J)			
		100	200	500	1000
Delta	CN	-0.626368000000	-0.626879050074	-0.627246662265	-0.627379382409
	ADE	-0.627047000000	-0.627564000814	-0.627933600507	-0.628066666327
	MOL	-0.627768000000	-0.627579795032	-0.627527214945	-0.627519705330
Gamma	CN	0.041916000000	0.041971187260	0.042012168143	0.042027134857
	ADE	0.041958000000	0.042013316573	0.042053978880	0.042068811245
	MOL	0.042075000000	0.042050787148	0.042044040734	0.042043076903
Theta	CN	-3.365345234190	-3.347102858400	-3.337252517880	-3.334151372240
	ADE	-3.364262097420	-3.346181287780	-3.336417332440	-3.333343230900
	MOL	- - -	- - -	- - -	- - -

Absolute Errors	Method	Equal Time (N) and Spatial Steps (J)			
		100	200	500	1000
Delta	CN	0.001149202038	0.000638151964	0.000270539773	0.000137819629
	ADE	0.000470202038	0.000046798776	0.000416398469	0.000549464289
	MOL	0.000250797962	0.000062592994	0.000010012907	0.000002503292
Gamma	CN	0.000127375742	0.000072188482	0.000031207599	0.000016240885
	ADE	0.000085375742	0.000030059169	0.000010603138	0.000025435503
	MOL	0.000031624258	0.000007411406	0.000000664992	0.000000298839
Theta	CN	0.034103510620	0.015861134830	0.006010794310	0.002909648670
	ADE	0.033020373850	0.014939564210	0.005175608870	0.002101507330
	MOL	- - -	- - -	- - -	- - -

TABLE 6.7: Divided Difference Sensitivity Approximations

Each of the time/spatial step count increments yielded an improvement on the accuracy of the approximations; the Method Of Lines scheme benefited the most from more time/spatial steps. At 1000 spatial/time steps, the ADE method produced the least accurate results in comparison to the CN and MOL scheme.

The approximations of Theta resulted in a higher absolute error compared to the errors of other sensitivity approximations. Possible explanations stem from the extraction of the result within *IBvpSolver* and the additional possibility of the approximation not working well when multiple approximations are taken along the spatial axis at each time step.

6.3.3 Cubic Spline Sensitivity Approximation

As the Divided Difference and Cubic Spline are carried out similarly, the approximation of Theta suffers from the same issue associated with the Method Of Lines scheme, and for this reason, no Theta approximations were obtained.

The results obtained from the Cubic Spline yielded near identical approximations in comparison to 6.7, with exceptions coming from the Theta sensitivity results. It is clear that variations in the results arise from the differences in option price approximations as identified in 6.3.1, it can be determined that the quality of approximations yielded from both methods are equally accurate for this set of data.

Sensitivity Approximations	Method	Equal Time (N) and Spatial Steps (J)			
		100	200	500	1000
Delta	CN	-0.626367985791	-0.626879050074	-0.627246662265	-0.627379382409
	ADE	-0.627046570414	-0.627564000814	-0.627933600507	-0.628066666327
	MOL	-0.627767762978	-0.627579795032	-0.627527214945	-0.627519705330
Gamma	CN	0.041915917578	0.041971187260	0.042012168143	0.042027134857
	ADE	0.041958304582	0.042013316573	0.042053978880	0.042068811245
	MOL	0.042074873787	0.042050787148	0.042044040734	0.042043076903
Theta	CN	-3.365345234190	-3.332227217523	-3.331315032449	-3.331184707114
	ADE	-3.364262097420	-3.331371666915	-3.330505918924	-3.330389548339
	MOL	- - -	- - -	- - -	- - -

Absolute Errors	Method	Equal Time (N) and Spatial Steps (J)			
		100	200	500	1000
Delta	CN	0.001149216247	0.000638151964	0.000270539773	0.000137819629
	ADE	0.000470631624	0.000046798776	0.000416398469	0.000549464289
	MOL	0.000250560940	0.000062592994	0.000010012907	0.000002503292
Gamma	CN	0.000127458164	0.000072188482	0.000031207599	0.000016240885
	ADE	0.000085071160	0.000030059169	0.000010603138	0.000025435503
	MOL	0.000031498045	0.000007411406	0.000000664992	0.000000298839
Theta	CN	0.034103510620	0.000985493953	0.000073308879	0.000057016456
	ADE	0.033020373850	0.000129943345	0.000735804646	0.000852175231
	MOL	- - -	- - -	- - -	- - -

TABLE 6.8: Cubic Spline Sensitivity Approximations

6.3.4 Continuous Sensitivity Equations

The approximation of the Greeks using CSEs faces many challenges, one of which is the determination of the boundary conditions and whether such solutions are unique within the truncated/transformed domain. The following results are obtained using the assumptions presented in section 4.5 and produce varying results which can often be explained by choice of boundary conditions and initial conditions applied to the CSEs.

Table 6.9 contains a set of results obtained at varying spatial/time step counts beginning at 1000 and ending at 2000. Each CSE is approximated using the Crank Nicolson, Alternating Direction Explicit and Method Of Lines scheme with their associated absolute errors in table 6.9. An immediate observation of the results is that an increase in the number of steps does not significantly improve the accuracy of the approximation; in some cases, the absolute error tends to climb as in Delta CSE ADE at $2000N/J$.

A secondary observation from 6.9 reveals the absolute errors associated with the Vega and Rho CSEs are much larger compared to the absolute errors of the Delta and Gamma CSE. A logical explanation is that a source term introduces additional inaccuracies into the approximation and thus inhibits any further improvement.

Sensitivity Approximations	Method	Equal Time (N) and Spatial Steps (J)			
		1000	1500	1800	2000
Delta CSE	CN	-0.625438435989	-0.626130906879	-0.626361830302	-0.628417381420
	ADE	-0.626053266879	-0.626747561726	-0.626979089992	-0.629038827791
	MOL	-0.625578023164	-0.626224139098	-0.626439572218	-0.628487738892
Gamma CSE	CN	0.041489592519	0.041489592519	0.041489382904	0.041489288552
	ADE	0.042060973178	0.042060973178	0.042062852560	0.042063914507
	MOL	0.039293324940	0.039293324940	0.039293279548	0.039293259949
Vega CSE	CN	10.962063113919	10.967161537103	10.961620369578	10.973885386542
	ADE	11.316844851069	11.322066520084	11.316364993956	11.328967001420
	MOL	11.309714630206	11.317380748314	11.312473063999	11.325466054044
Rho CSE	CN	-10.838654360702	-10.856028218188	-10.857301087458	-10.863104298683
	ADE	-10.838669999515	-10.856039567039	-10.857310907448	-10.863113528672
	MOL	-10.829672060403	-10.850039742616	-10.852309495406	-10.858613418865

Absolute Errors	Method	Equal Time (N) and Spatial Steps (J)			
		1000	1500	1800	2000
Delta CSE	CN	0.002078766049	0.001386295159	0.001155371736	0.000900179382
	ADE	0.001463935159	0.000769640312	0.000538112046	0.001521625753
	MOL	0.001939178874	0.001293062940	0.001077629820	0.000970536854
Gamma CSE	CN	0.000553783223	0.000553783223	0.000553992838	0.000554087190
	ADE	0.000017597436	0.000017597436	0.000019476818	0.000020538765
	MOL	0.002750050802	0.002750050802	0.002750096194	0.002750115793
Vega CSE	CN	0.389648336281	0.384549913097	0.390091080622	0.377826063658
	ADE	0.034866599131	0.029644930116	0.035346456244	0.022744448780
	MOL	0.041996819994	0.034330701886	0.039238386201	0.026245396156
Rho CSE	CN	0.035674222298	0.018300364812	0.017027495542	0.011224284317
	ADE	0.035658583485	0.018289015961	0.017017675552	0.011215054328
	MOL	0.044656522597	0.024288840384	0.022019087594	0.015715164135

TABLE 6.9: Greek CSE Approximations

A further insight into the behaviour of the CSE equation approximations can be obtained by varying the number of spatial/time steps by an increment of 25 steps up until $N/J = 2000$ (80×25). The following figures present the approximations separately for each sensitivity:

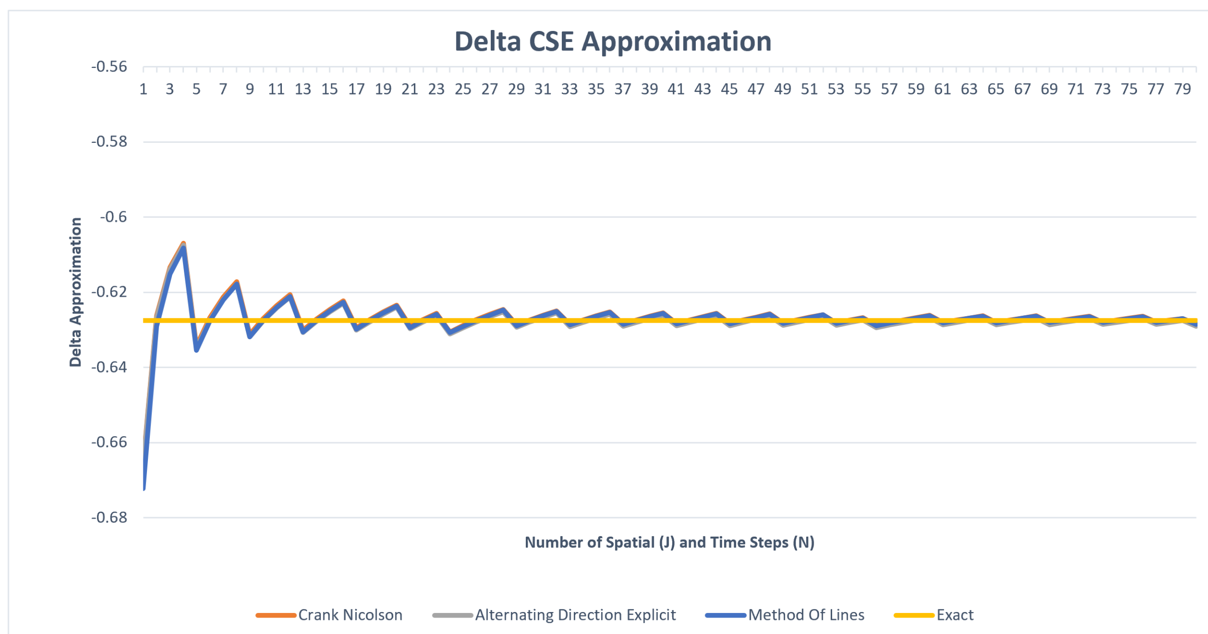

TABLE 6.10: Delta CSE Plot

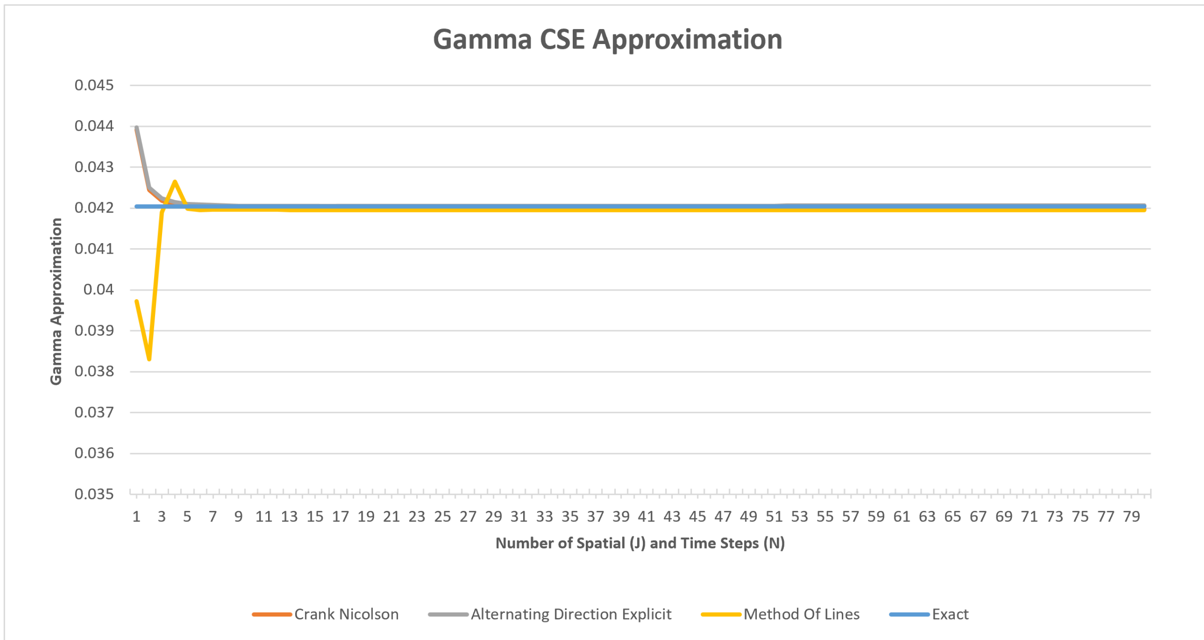

TABLE 6.11: Gamma CSE Plot

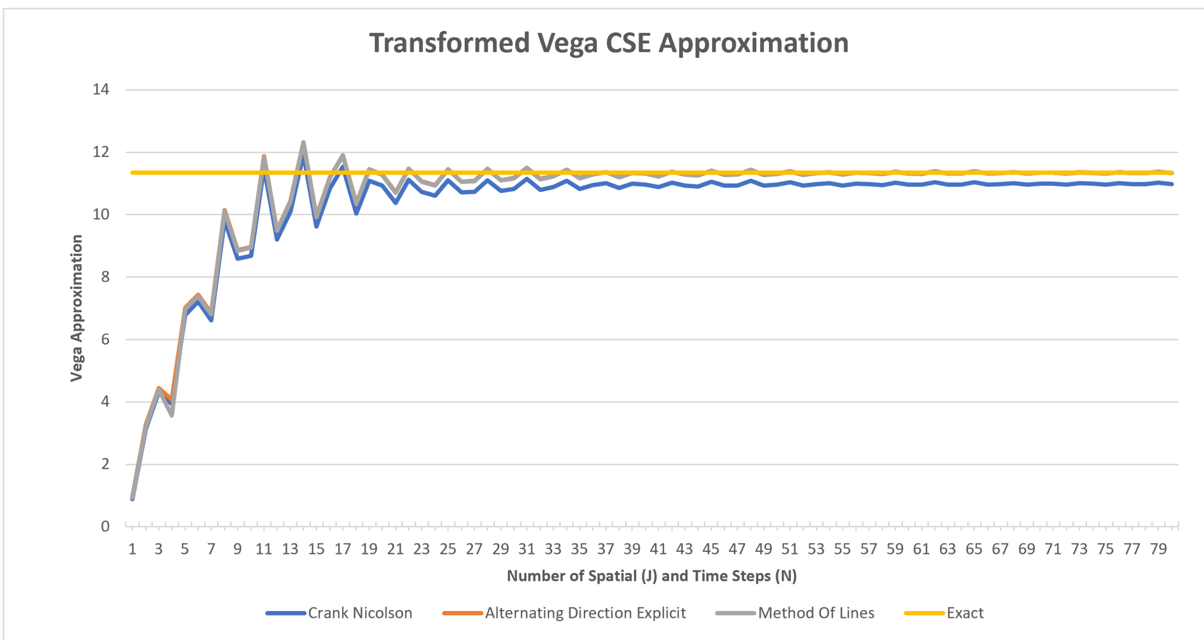

TABLE 6.12: Vega CSE Plot

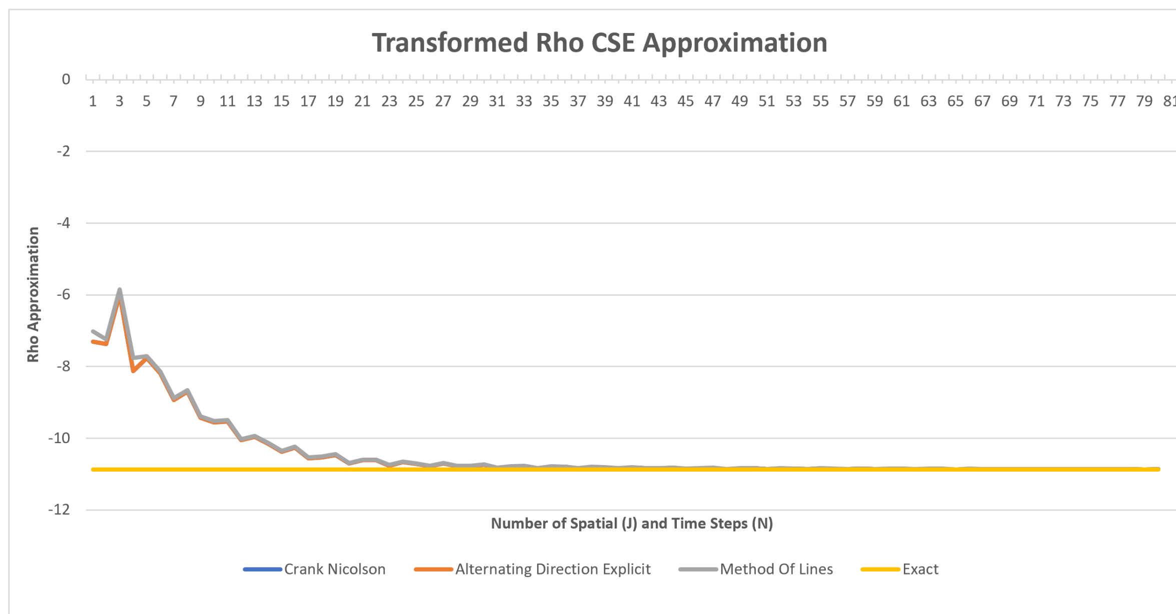

TABLE 6.13: Rho CSE Plot

Figures 6.10, 6.12 and 6.13 display similar oscillatory behaviour where at low spatial/time step counts the approximation begins to revert towards the mean. At around 975 spatial/time steps (39) the approximation begins to smooth off with little observable variation, thus indicating a minimum number of steps required to obtain a sensible approximation. Figure 6.10 continues to oscillate at high step counts which may be the result of the Delta CSE containing no reaction term and the Heaviside function representing the initial condition of the CSE.

Both figures 6.11 and 6.12 demonstrate the importance of correct boundary condition approximations. The Gamma CSE uses an approximation of the Dirac delta function where the user sets the steepness parameter (λ); the addition of any user-defined parameter can lead to significant loses in the accuracy of the approximation and render the CSE impractical. Similarly, the boundary conditions of Vega are based on the assumptions of it decreasing to zero at expiry and is therefore used as an initial condition. As previously discussed in 4.5, Vega may never in practice, be zero and requires further research to determine the exact boundary conditions.

The final figure 6.13 demonstrates the results of the CSE through a trial and error approach where the initial condition has been varied to obtain a suitable approximation. Figure 6.14 demonstrates the effect of different initial conditions on the Crank Nicolson approximation of Rho.

Under the assumption that Rho decreases, the initial condition used in 4.13 varied around zero, beginning at $\rho = 0.04$ and ending at $\rho = -0.04$ (at option expiry) with the exact put option bond price V fixed at 5.846285626870. The exact solution line demonstrates the importance of selecting the correct initial condition since Rho is negative concerning a put option; the initial condition would also be negative, which is supported by the results.

Figure 6.14 supports the idea that increasing the number of spatial/time steps requires an optimal initial condition closer to zero. Therefore it could be hypothesised that an initial condition of zero (as assumed in the Vega CSE) would yield optimal approximations with a higher number of spatial/steps.

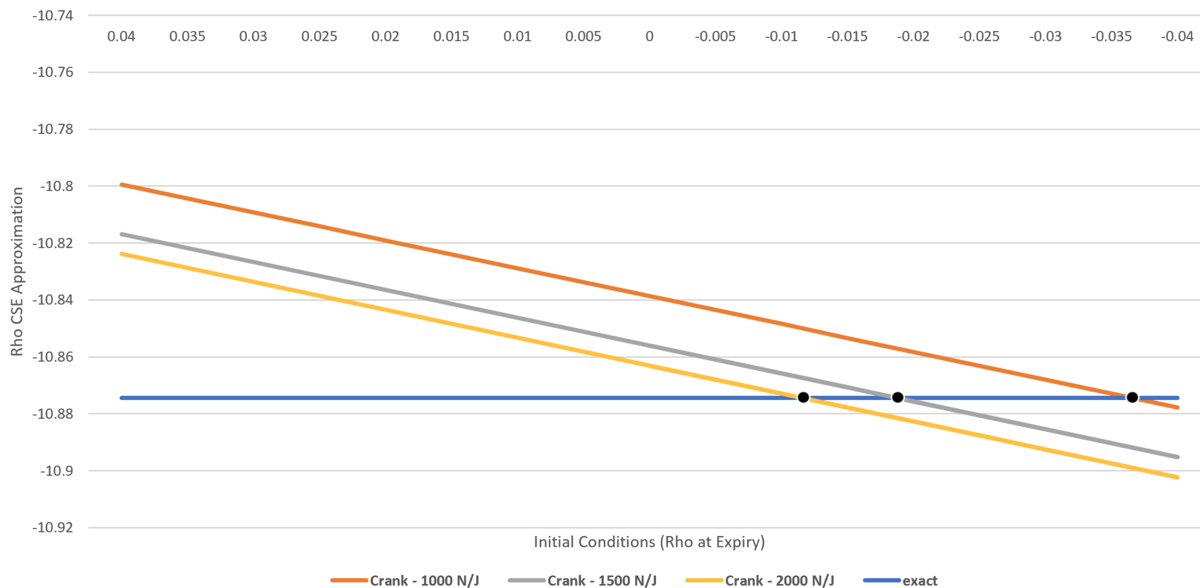

TABLE 6.14: Rho CSE Approximations with varied initial conditions

6.4 Cox-Ingersoll-Ross Numerical Approximations

The CIR zero-coupon bond pricing PDE posed a challenge in obtaining suitable approximations to the sensitivities presented within this thesis. Implementations into both C++ and MATLAB were carried out with MATLAB producing acceptable Bond pricing approximations. Therefore the following bond price approximations are obtained using the Alternating Direction Explicit and Crank Nicolson MATLAB implementations (full code is in appendix E):

Exact Bond Price	0.978966810291

Time / Spatial Steps	ADE	Error	Crank Nicolson	Error
250 N/J	0.978963415112	3.39518E-06	0.978966689980	1.20311E-07
500 N/J	0.978963058298	3.75199E-06	0.978966939504	1.29213E-07
1000 N/J	0.978962677106	4.13319E-06	0.978966844584	3.42932E-08
1200 N/J	0.978962638629	4.17166E-06	0.978966853070	4.27792E-08
1400 N/J	0.978962577104	4.23319E-06	0.978966824929	1.46377E-08

FIGURE 6.1: CIR bond price results table

The results show that the Crank Nicolson method produces the most accurate results with smaller absolute errors for each case. The substantial differences between the ADE and CN methods potentially arise from the different implementations of the Thomee method that chapter 3 introduces to approximate the CIR boundary condition PDE. Such differences in the approximation of the boundary conditions may lead to further variations in the results.

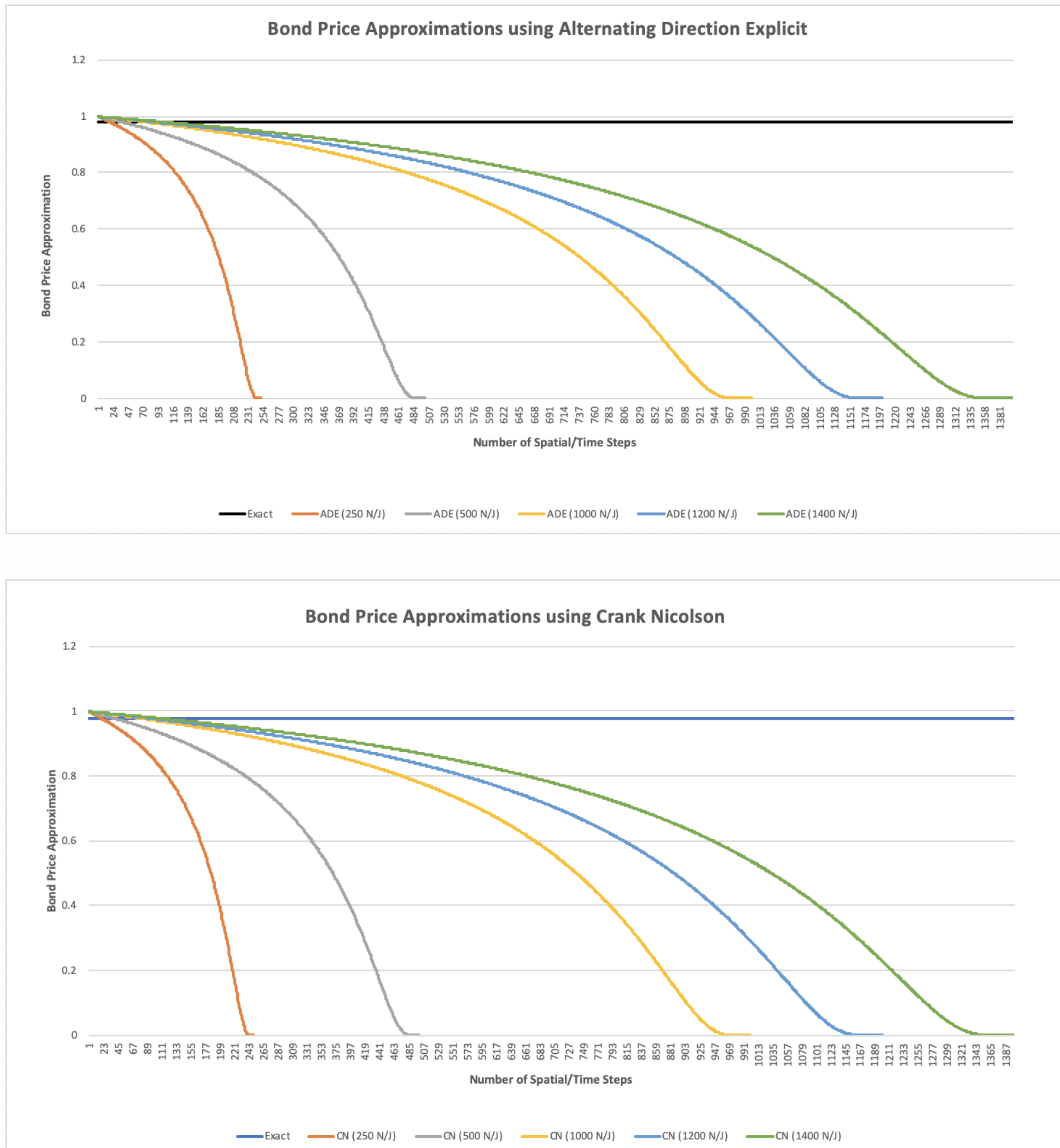

FIGURE 6.2: CIR bond price approximations

Figure 6.2 Demonstrates the behaviour of the approximations by beginning at a unit approximation and decreasing steadily until reaching a bond price approximation of 0. The figures show that as a bond approaches maturity, it will steadily decrease in value. This is intuitive as the price of the bond and its payoff are negatively correlated. It is also clear that the approximation

is obtained almost instantly, even at a lower time/spatial step count. This directly connects with the nature of the zero-coupon bond as the bond price will be close to its payoff at $t = 0$.

If the lifetime of a zero-coupon bond is observed up until maturity, the highest price of the bond will be at time t = 0. The subsequent bond price will be close if not equivalent to its payoff at maturity. The issue of using zero-coupon bonds to approximate sensitivities arises from the lack of variation in the approximation of the bond price. This is shown through the approximation of the Duration and Convexity sensitivities using the Divided Difference method, consider the following:

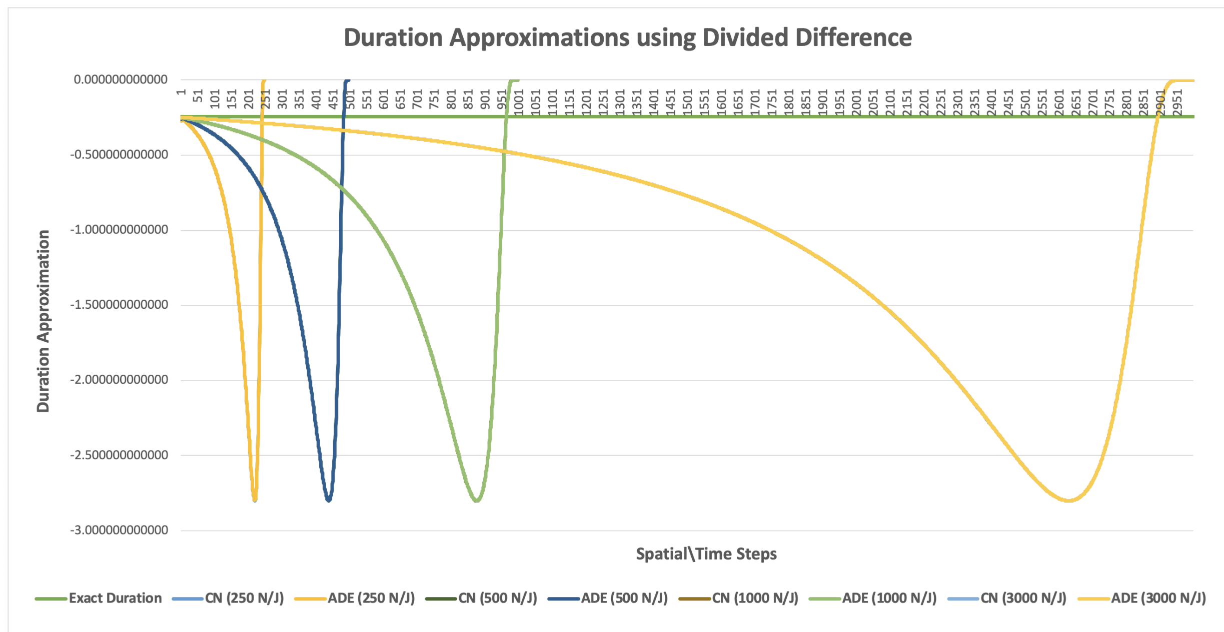

FIGURE 6.3: CIR Divided Difference Duration approximations

Figure 6.3 contains the approximations of the Duration sensitivity using the Crank Nicolson and Alternating Direction Explicit methods for varying spatial and time steps. The dark green line represents the exact value of the Duration sensitivity, and by observation, it is clear that from the initial estimate, the Divided Difference method has over-approximated the value. Despite implementation into C++ and MATLAB, the explanation stems from the lack of variation in the first set of bond price approximations such that the application of any central difference scheme results in an immediate overshooting. This was found to be an inherent issue with zero-coupon bond prices with a similar result coming from the Convexity sensitivity approximation using the same approximation method.

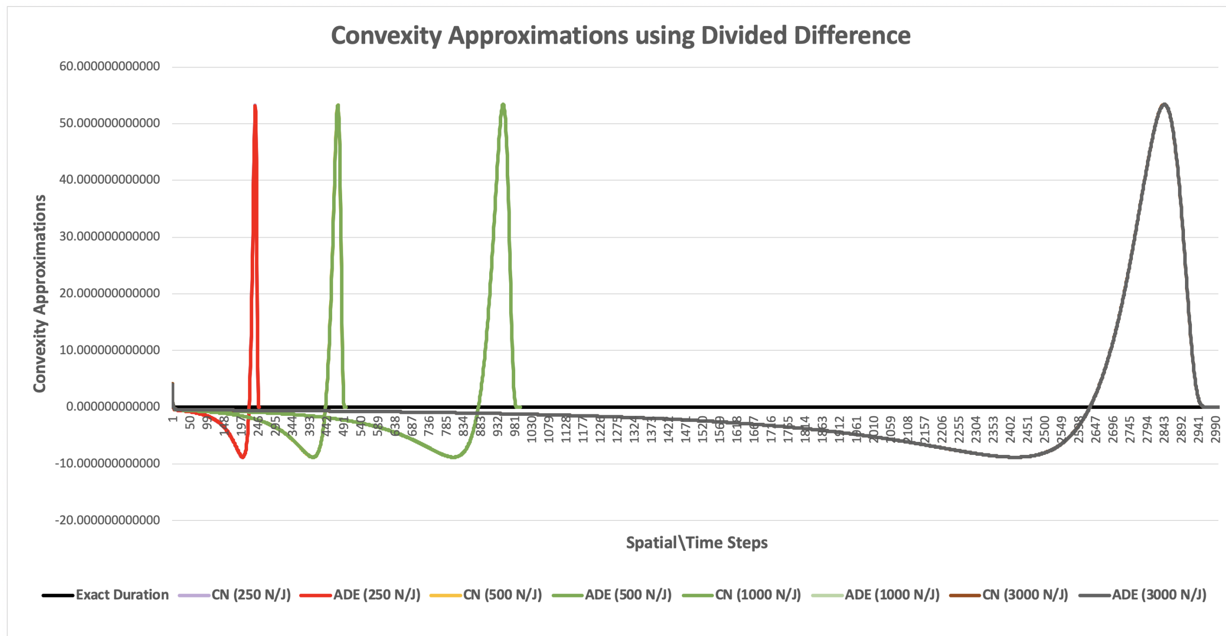

FIGURE 6.4: CIR Divided Difference Convexity approximations

Figure 6.4 shows that the Divided Difference underestimates the sensitivity as highlighted by the exact (black line) Convexity value. Subsequent methods that implement FDM grids can be expected to behave similarly. A remedy would be to approximate the sensitivity of Bond options which hold similarities to the Black-Scholes approximation methodology. Therefore the issues associated with the zero-coupon bond will be negated as the initial approximations will not be similar to the payoff the bond at maturity but the payoff of the option at expiry.

This was further tested by implementing the CIR Volatility continuous sensitivity equation as a proof of concept. The CSE closely resembles the standard CIR bond approximation PDE with an identical boundary PDE and d/c/r terms with the addition of an inhomogeneous source term component.

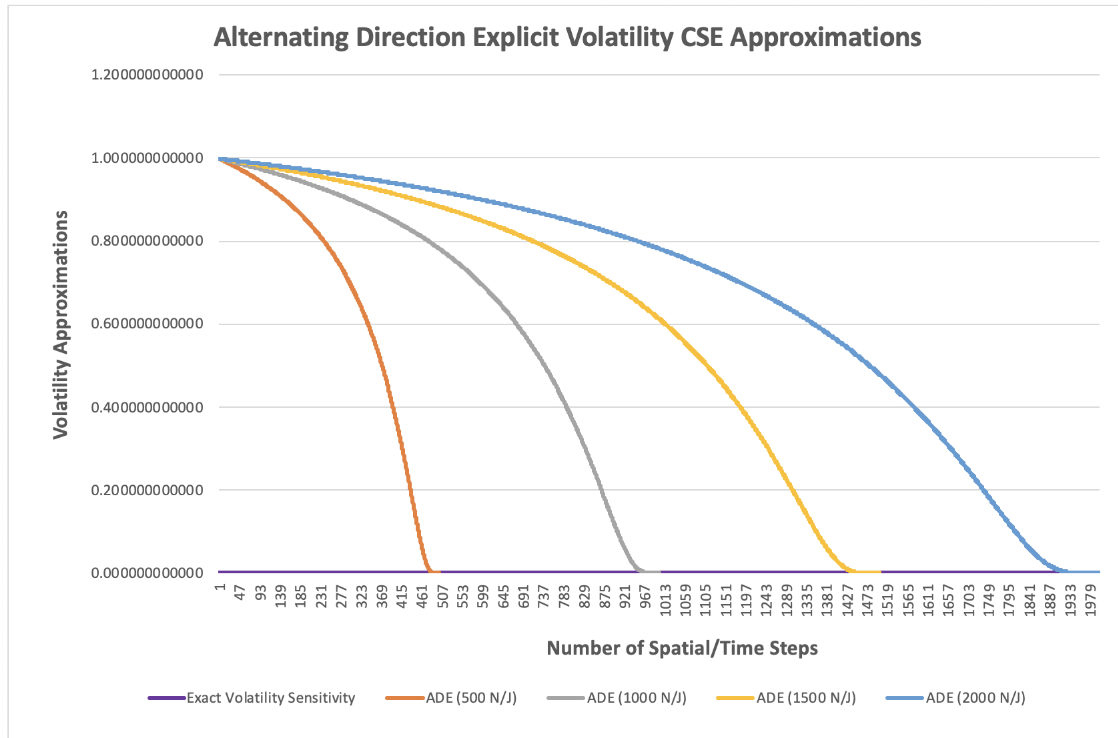

FIGURE 6.5: CIR ADE Volatility CSE approximations

Given the exact value of the volatility sensitivity in 6.1, the results obtained from the approximations are entirely off and resemble the bond price approximations in figure 6.2. Once again, the results are affected by two possible issues — first, the existence of multiple solutions which result in the poor estimate. Or secondly, the problems where the application of central differencing (either second or first-order) will result in poor approximations due to the lack of variations in the bond price at around time t = 0.

A Option Greek Derivations

The derivation of the option Greeks stem from the introduction of the analytical solution of the Black-Scholes PDE (2.2). The following closed form Greeks are provided in Espen Haug's (2009) book on Option Pricing Formulas [15].

Consider the following identities:

Gaussian CDF: $\mathcal{N}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{(\frac{-z^2}{2})} dz,$

Gaussian PDF: $n(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}},$

CDF Derivative: $\frac{\partial \mathcal{N}(x)}{\partial \eta} = n(x) \frac{\partial x}{\partial \eta}.$

Consider the components d_1 and d_2 of 2.2. Time for d_2^2 must be calculated by the following:

$$\begin{aligned} d_2^2 &= (d_1 - \sigma\sqrt{T})(d_1 - \sigma\sqrt{T}) = d_1^2 - 2d_1\sigma\sqrt{T} + \sigma^2 T \\ &= d_1^2 - 2\left(\frac{\ln\frac{S}{K} + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}}\right)\sigma\sqrt{T} + \sigma^2 T = d_1^2 - 2\ln\frac{S}{K} - 2rT - \sigma^2 T + \sigma^2 T \\ &= d_1^2 - 2\ln\left(\frac{Se^{rT}}{K}\right). \end{aligned}$$

Now consider the normal PDF with respect to d_2:

$$n(d_2) = \frac{1}{\sqrt{2\pi}} e^{\frac{-d_2^2}{2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(d_1^2 - 2\ln\left(\frac{Se^{rT}}{K}\right)\right)} = n(d_1)\frac{S}{K}e^{rT}.$$

The closed form Greeks can now be derived longhand.

A.1 Greek Delta

Differentiating 2.2 with respect to the underlying S yields:

$$\frac{\partial C}{\partial S} = \mathcal{N}(d_1) + S\frac{\partial \mathcal{N}(d_1)}{\partial S} - Ke^{-rT}\frac{\partial \mathcal{N}(d_2)}{\partial S}$$

$$\begin{aligned}
&= \mathcal{N}(d_1) + S\tfrac{\partial \mathcal{N}(d_1)}{\partial d_1}\tfrac{\partial d_1}{\partial S} - Ke^{-rT}\tfrac{\partial \mathcal{N}(d_2)}{\partial d_2}\tfrac{\partial d_2}{\partial S} \\
&= \mathcal{N}(d_1) + Sn(d_1)\tfrac{\partial d_1}{\partial S} - Ke^{-rT}n(d_2)\tfrac{\partial d_2}{\partial S} \\
&= \mathcal{N}(d_1) + Sn(d_1)\tfrac{\partial d_1}{\partial S} - Ke^{-rT}n(d_1)\tfrac{Se^{rT}}{K}\tfrac{\partial d_2}{\partial S} \\
&= \mathcal{N}d_1 + Sn(d_1)\tfrac{\partial d_1}{\partial S} - Sn(d_1)\tfrac{\partial d_2}{\partial S}.
\end{aligned}$$

Now evaluate the derivative of d_1 and d_2 with respect to the underlying:

$$\frac{\partial d_1}{\partial S} = \frac{\partial}{\partial S}\left(\frac{\ln\left(\frac{S}{K}\right) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}}\right) = \frac{1}{S}\frac{1}{\sigma\sqrt{T}} = \frac{\partial d_2}{\partial S}.$$

Moving attention back to the call price derivative:

$$\frac{\partial C}{\partial S} = \mathcal{N}(d_1) + Sn(d_1)\left(\frac{\partial d_1}{\partial S} - \frac{\partial d_2}{\partial S}\right) = \mathcal{N}(d_1) > 0.$$

The following closed form solution is obtained for Greek Delta (Haug, 2009) [15, pg.21]:

$$\begin{aligned}
\Delta_{call} &= \tfrac{\partial C}{\partial S} = \mathcal{N}(d_1) > 0. \\
\Delta_{Put} &= \tfrac{\partial P}{\partial S} = \mathcal{N}(d_1) - 1 > 0.
\end{aligned}$$

A.2 Greek Gamma

Since Gamma is the second derivative of the analytical Black-Scholes solution (2.2) with respect to the underlying S. The closed form solution of Gamma is obtained by simply differentiating Greek Delta again with respect to S.

$$\frac{\partial^2 C}{\partial S^2} = \frac{\partial \mathcal{N}(d_1)}{\partial S} = n(d_1)\frac{\partial d_1}{\partial S}.$$

Differentiating d_1 with respect to S yields the following:

$$\frac{\partial^2 C}{\partial S^2} = n(d_1)\frac{1}{S\sigma\sqrt{T}} > 0.$$

The following closed form solution is obtained for Greek Gamma (Haug, 2009 [15, pg.38]):

$$\Gamma_{call} = \Gamma_{put} = \frac{1}{S\sigma\sqrt{2\pi T}}e^{\frac{-d_1^2}{2}} > 0.$$

A.3 Greek Rho

Differentiate 2.2 with respect to the risk free interest rate r:

$$\begin{aligned}
\frac{\partial C}{\partial r} &= S\frac{\partial \mathcal{N}(d_1)}{\partial r} + TKe^{-rT}\mathcal{N}(d_2) - Ke^{-rT}\frac{\partial \mathcal{N}(d_2)}{\partial r} \\
&= Sn(d_1)\frac{\partial d_1}{\partial r} + TKe^{-rT}\mathcal{N}(d_2) - Ke^{-rT}n(d_2)\frac{\partial d_2}{\partial r} \\
&= Sn(d_1)\frac{\partial d_1}{\partial r} + TKe^{-rT}\mathcal{N}(d_2) - Ke^{-rT}n(d_1)\frac{S}{K}e^{rT}\frac{\partial d_2}{\partial r} \\
&= Sn(d_1)\left(\frac{\partial d_1}{\partial r} - \frac{\partial d_2}{\partial r}\right) + TKe^{-rT}\mathcal{N}(d_2), \\
& \text{Note: } \left(\frac{\partial d_1}{\partial r} - \frac{\partial d_2}{\partial r}\right) = \frac{1}{\sigma}\left(\sqrt{T} - \sqrt{T}\right). \\
&= TKe^{-rT}\mathcal{N}(d_2).
\end{aligned}$$

Greek Rho has the following closed form (Haug, 2009) [15, pg.69]:

$$\begin{aligned}
\rho_{call} &= TKe^{-rT}\mathcal{N}(d_2) > 0. \\
\rho_{Put} &= -TKe^{-rT}\mathcal{N}(-d_2) < 0.
\end{aligned}$$

A.4 Greek Theta

Theta is often represented by a negative derivative due to the time decay on an options value that the Greek represents. Therefore differentiate 2.2 by the expiration date of the option T...

$$\begin{aligned}
-\frac{\partial C}{\partial T} &= -S\frac{\partial \mathcal{N}(d_1)}{\partial T} - rKe^{-rT}\mathcal{N}(d_2) + Ke^{-rT}\frac{\partial \mathcal{N}()d_2}{\partial T} \\
&= -Sn(d_1)\frac{\partial d_1}{\partial T} - rKe^{-rT}\mathcal{N}(d_2) + Ke^{-rT}n(d_2)\frac{\partial d_2}{\partial T} \\
&= -Sn(d_1)\frac{\partial d_1}{\partial T} - rKe^{-rT}\mathcal{N}(d_2) + Ke^{-rT}n(d_1)\frac{S}{K}e^{rT}\frac{\partial d_2}{\partial T} \\
&= Sn(d_1)\left(\frac{\partial d_2}{\partial T} - \frac{\partial d_1}{\partial T}\right) - rKe^{-rT}\mathcal{N}(d_2).
\end{aligned}$$

The derivatives of d_1 and d_2 with respect to the expiration date are:

$$\frac{\partial d_{1,2}}{\partial T} = -\frac{1}{2\sigma}\ln\left(\frac{S}{K}\right)T^{-\frac{3}{2}} + \frac{1}{2\sigma}\left(r \pm \frac{\sigma^2}{2}\right)T^{-\frac{1}{2}}.$$

Substituting it into the Theta derivative yields:

$$-\frac{\partial C}{\partial T} = Sn(d_1)\Big(-\frac{1}{2\sqrt{T}}\sigma\Big) - rKe^{-rT}\mathcal{N}(d_2) = -\frac{Sn(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}\mathcal{N}(d_2).$$

Greek Theta has the following closed form [15, pg.64]:

$$\theta_{call} = -\frac{Sn(d_1)\sigma}{2\sqrt{T}} - rKe^{-rT}\mathcal{N}(d_2).$$

$$\theta_{Put} = -\frac{Sn(d_1)\sigma}{2\sqrt{T}} + rKe^{-rT}\mathcal{N}(-d_2).$$

A.5 Greek Vega

The final Greek is Vega ($\mathcal{V}$) and is the differential of 2.2 with respect to its volatility σ.

$$\frac{\partial C}{\partial \sigma} = S\frac{\partial \mathcal{N}(d_1)}{\partial d_\sigma} - Ke^{-rT}\frac{\partial \mathcal{N}(d_2)}{\partial \sigma} = Sn(d_1)\frac{\partial d_1}{\partial \sigma} - Ke^{-rT}n(d_2)\frac{\partial d_2}{\partial \sigma}.$$

Recalling that $n(d_2) = n(d_1)\frac{Se^{rT}}{K}\ldots$

$$\frac{\partial C}{\partial \sigma} = Sn(d_1)\frac{\partial d_1}{\partial \sigma} - Ke^{-rT}n(d_1)\frac{S}{K}e^{rT} = Sn(d_1)\Big(\frac{\partial d_1}{\partial \sigma} - \frac{\partial d_2}{\partial \sigma}\Big).$$

The derivatives of d_1 and d_2 are as follows:

$$\frac{\partial d_{1,2}}{\partial \sigma} = -\frac{1}{\sigma^2\sqrt{T}}\ln\left(\frac{Se^{rT}}{K}\right) \pm \frac{1}{2}\sqrt{T}.$$

Substituting the above into the Greek calculation yields the closed form solution of Greek Vega [15, pg.50]:

$$\mathcal{V}_{call} = \mathcal{V}_{put} = \frac{S}{\sqrt{2\pi}}e^{-\frac{-d_1^2}{2}}\sqrt{T} > 0.$$

B Sensitivity Components

B.1 Speed of Adjustment Closed Form

$$
\begin{aligned}
\frac{\partial a(t,T,\kappa)}{\partial \kappa} &= \frac{1}{\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)^{\frac{2\theta\kappa}{\sigma^2}}} \\
&\times \left(\left(\kappa^2+2\sigma^2\right)^{\frac{\theta\kappa}{\sigma^2}} \cdot 2^{\frac{2\theta\kappa}{\sigma^2}} e^{\frac{t\theta\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)}{\sigma^2}} \left(-\frac{2\theta\ln\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)}{\sigma^2}\right.\right. \\
&- \left.\left.\frac{2\theta\kappa\left(\frac{t\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)e^{t\sqrt{\kappa^2-2\sigma^2}}}{\sqrt{\kappa^2-2\sigma^2}}+\left(\frac{\kappa}{\sqrt{\kappa^2+2\sigma^2}}+1\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+\frac{2\kappa}{\sqrt{\kappa^2+2\sigma^2}}\right)}{\sigma^2\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)}\right)\right) \\
&+ \frac{\left(\kappa^2+2\sigma^2\right)^{\frac{\theta\kappa}{\sigma^2}} \cdot 2^{\frac{2\theta\kappa}{\sigma^2}} e^{\frac{t\theta\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)}{\sigma^2}} \left(\frac{\theta\ln\left(\kappa^2+2\sigma^2\right)}{\sigma^2}+\frac{2\theta\kappa^2}{\sigma^2\left(\kappa^2+2\sigma^2\right)}\right)}{\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)^{\frac{2\theta\kappa}{\sigma^2}}} \\
&+ \frac{\ln(2)\,\theta\left(\kappa^2+2\sigma^2\right)^{\frac{\theta\kappa}{\sigma^2}} \cdot 2^{\frac{2\theta\kappa}{\sigma^2}+1} e^{\frac{t\theta\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)}{\sigma^2}}}{\sigma^2\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)^{\frac{2\theta\kappa}{\sigma^2}}} \\
&+ \frac{\left(\kappa^2+2\sigma^2\right)^{\frac{\theta\kappa}{\sigma^2}}\left(\frac{t\theta\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)}{\sigma^2}+\frac{t\theta\kappa\left(\frac{\kappa}{\sqrt{\kappa^2+2\sigma^2}}+1\right)}{\sigma^2}\right) \cdot 2^{\frac{2\theta\kappa}{\sigma^2}} e^{\frac{t\theta\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)}{\sigma^2}}}{\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)\left(e^{t\sqrt{\kappa^2-2\sigma^2}}-1\right)+2\sqrt{\kappa^2+2\sigma^2}\right)^{\frac{2\theta\kappa}{\sigma^2}}}.
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial b(t,T,\kappa)}{\partial \kappa} &= \frac{2\kappa e^{\sqrt{\kappa^2+2\sigma^2}}}{\sqrt{\kappa^2+2\sigma^2}\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)e^{T\sqrt{\kappa^2-2\sigma^2}}+\sqrt{\kappa^2+2\sigma^2}-\kappa\right)} \\
&- \frac{\left(\frac{T\kappa\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)e^{T\sqrt{\kappa^2-2\sigma^2}}}{\sqrt{\kappa^2-2\sigma^2}}+\left(\frac{\kappa}{\sqrt{\kappa^2+2\sigma^2}}+1\right)e^{T\sqrt{\kappa^2-2\sigma^2}}+\frac{\kappa}{\sqrt{\kappa^2+2\sigma^2}}-1\right)\left(2e^{\sqrt{\kappa^2+2\sigma^2}-2}\right)}{\left(\left(\sqrt{\kappa^2+2\sigma^2}+\kappa\right)e^{T\sqrt{\kappa^2-2\sigma^2}}+\sqrt{\kappa^2+2\sigma^2}-\kappa\right)^2}.
\end{aligned}
$$

B.2 Volatility Closed Form

$$
\begin{aligned}
\frac{\partial a(t,T,\sigma)}{\partial\sigma} &= \frac{1}{\sigma^3\left(\left(\sqrt{2\sigma^2+\kappa^2}+\kappa\right)\left(e^{t\sqrt{2\sigma^2+\kappa^2}}-1\right)+2\sqrt{2\sigma^2+\kappa^2}\right)^{\frac{2\kappa\theta}{\sigma^2}}\left(\left(\sqrt{2\sigma^2+\kappa^2}+\kappa\right)e^{t\sqrt{2\sigma^2+\kappa^2}}+\sqrt{2\sigma^2+\kappa^2}-\kappa\right)} \\
&\times \kappa\theta\left(2\sigma^2+\kappa^2\right)^{\frac{\kappa\theta}{\sigma^2}-\frac{3}{2}}\cdot 2^{\frac{2\kappa\theta}{\sigma^2}+1}e^{\frac{\kappa t\theta\left(\sqrt{2\sigma^2+\kappa^2}+\kappa\right)}{\sigma^2}} \\
&\times \Bigg(\bigg(\Big(8\sigma^4+\sqrt{2\sigma^2+\kappa^2}\Big((4\kappa\sigma^2+2\kappa^3\Big)+8\kappa^2\sigma^2+2\kappa^4\Big)e^{t\sqrt{2\sigma^2+\kappa^2}}+8\sigma^4 \\
&+ \sqrt{2\sigma^2+\kappa^2}\left(-4\kappa\sigma^2-2\kappa^3\right)+8\kappa^2\sigma^2+2\kappa^4\Big)\ln\Big(\left(\sqrt{2\sigma^2+\kappa^2}+\kappa\right)\left(e^{t\sqrt{2\sigma^2+\kappa^2}}-1\right) \\
&+ 2\sqrt{2\sigma^2+\kappa^2}\Big)+\Big(\Big(-4\sigma^4+\sqrt{2\sigma^2+\kappa^2}\Big(-2\kappa\sigma^2-\kappa^3\Big)-4\kappa^2\sigma^2-\kappa^4\Big)e^{t\sqrt{2\sigma^2+\kappa^2}} \\
&- 4\sigma^4+\sqrt{2\sigma^2+\kappa^2}\Big(2\kappa\sigma^2+\kappa^3\Big)-4\kappa^2\sigma^2-\kappa^4\Big)\ln\Big(2\sigma^2+\kappa^2\Big) \\
&+ \Big(\sqrt{2\sigma^2+\kappa^2}\Big(-2t\sigma^4+\Big(\Big(2-4\ln(2)\Big)\kappa-3\kappa^2t\Big)\sigma^2-\kappa^4t-2\ln(2)\kappa^3\Big) \\
&+ \Big(-10\kappa t-8\ln(2)\Big)\sigma^4+\Big(2\sigma^2+\kappa^2\Big)^{\frac{3}{2}}\Big(-2t\sigma^2-\kappa^2t\Big)+\Big(-9\kappa^3t-8\ln(2)\kappa^2\Big)\sigma^2 \\
&- 2\kappa^5t-2\ln(2)\kappa^4\Big)e^{t\sqrt{2\sigma^2+\kappa^2}}+\sqrt{2\sigma^2+\kappa^2}\Big(2t\sigma^4+\Big(3\kappa^2t+\Big(4\ln(2)-2\Big)\kappa\Big)\sigma^2+\kappa^4t \\
&+ 2\ln(2)\kappa^3\Big)+\Big(-2\kappa t-8\ln(2)\Big)\sigma^4+\Big(2\sigma^2+\kappa^2\Big)^{\frac{3}{2}}\Big(-2t\sigma^2-\kappa^2t\Big) \\
&+ \Big(-\kappa^3t-8\ln(2)\kappa^2\Big)\sigma^2-2\ln(2)\kappa^4\Bigg).
\end{aligned}
$$

$$
\frac{\partial b(t,T,\sigma)}{\partial\sigma} = -\frac{4\sigma\left(e^{2t\sqrt{2\sigma^2+\kappa^2}}-2t\sqrt{2\sigma^2+\kappa^2}e^{t\sqrt{2\sigma^2+\kappa^2}}-1\right)}{\sqrt{2\sigma^2+\kappa^2}\left(\left(\sqrt{2\sigma^2+\kappa^2}+k\right)e^{t\sqrt{2\sigma^2+\kappa^2}}+\sqrt{2\sigma^2+\kappa^2}-k\right)^2}.
$$

C Continuous Sensitivity Equation Derivations

C.1 Black-Scholes Continuous Sensitivity Equation Derivations

Each derivation will begin from the Black-Scholes PDE.

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV, \qquad \text{Where: } V = V(S,t).$$

C.1.1 Greek Delta CSE

Differentiate with respect to S:

$$\frac{\partial^2 V}{\partial S \partial t} = \sigma^2 S \frac{\partial^2 V}{\partial S^2} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^3 V}{\partial S^3} + r\frac{\partial V}{\partial S} - r\frac{\partial V}{\partial S} + rS\frac{\partial^2 V}{\partial S^2}. \tag{C.1}$$

Make the following substitution: $\Delta = \frac{\partial V}{\partial S}$ and rearrange to obtain:

$$\frac{\partial \Delta}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \Delta}{\partial S^2} + (\sigma^2 + r) S \frac{\partial \Delta}{\partial S}.$$

C.1.2 Greek Gamma CSE

As Gamma is by definition the second derivative of Delta. Differentiating equation C.1 with respect to the underlying will yield the Gamma CSE:

$$\frac{\partial^3 V}{\partial S^2 \partial t} = \sigma^2 \frac{\partial^2 V}{\partial S^2} + 2\sigma^2 S \frac{\partial^3 V}{\partial S^3} \frac{1}{2}\sigma^2 S^2 \frac{\partial^4 V}{\partial S^4} + r\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial^3 V}{\partial S^3}.$$

Make the following substitution: $\Gamma = \frac{\partial^2 V}{\partial S^2}$ and rearrange:

$$\frac{\partial \Gamma}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \Gamma}{\partial S^2} + \left(2\sigma^2 S + rS\right)\frac{\partial \Gamma}{\partial S} + (\sigma^2 + r)\Gamma.$$

C.1.3 Greek Vega CSE

Considering the Black-Scholes PDE, the option price V is represented as $V(S,t)$. However, it relies on all the parameters in the formation of the option price, it can therefore be denoted as $V = V(S,t,\sigma,K,r)$.

Now the differentiating Black-Scholes equation with respect to σ yields:

$$\frac{\partial^2 V}{\partial t \partial \sigma} = \frac{1}{2}\sigma^2 \frac{\partial^3 V}{\partial \sigma \partial S^2} + rS\frac{\partial^2 V}{\partial \sigma \partial S} - r\frac{\partial V}{\partial \sigma} + \sigma S^2 \frac{\partial^2 V}{\partial S^2}.$$

Making the following substitution: $\mathcal{V} = \frac{\partial V}{\partial \sigma}$ and rearranging yields:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \mathcal{V}}{\partial S^2} + rS\frac{\partial \mathcal{V}}{\partial S} - r\mathcal{V} + \sigma S^2 \frac{\partial^2 V}{\partial S^2}.$$

C.1.4 Greek Rho CSE

Differentiate the Black-Scholes PDE with respect to the interest rate r:

$$\frac{\partial^2 V}{\partial r \partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^3 V}{\partial r \partial S^2} + rS\frac{\partial^2 V}{\partial r \partial S} - r\frac{\partial V}{\partial r} + S\frac{\partial V}{\partial S} - V.$$

Make the following substitution: $\rho = \frac{\partial V}{\partial r}$ and rearrange:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 \rho}{\partial S^2} + rS\frac{\partial \rho}{\partial S} - r\rho + S\frac{\partial V}{\partial S} - V.$$

C.2 CIR Continuous Sensitivity Equation Derivations

Each of the following CSEs will be derived using the CIR zero coupon bond pricing PDE:

$$\frac{\partial B}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial r^2} + (a - br)\frac{\partial B}{\partial r} - rB.$$

Notably, for each derivation the CSE of the standard CIR PDE defined on the semi-infinite domain will be calculated. The domain transformation discussed in 3 will be applied resulting in the domain transformed CSE equation for the respective sensitivity.

C.2.1 Duration CSE

Differentiate the CIR PDE with respect to the interest rate:

$$\frac{\partial^2 B}{\partial r \partial t} = \frac{1}{2}\sigma^2 \frac{\partial^2 B}{\partial r^2} + \frac{1}{2}\sigma^2 r \frac{\partial^3 B}{\partial r^3} + (a - br)\frac{\partial^2 B}{\partial r^2} - b\frac{\partial B}{\partial r} - r\frac{\partial B}{\partial r} - B. \tag{C.2}$$

Substituting in $\mathcal{D} = \frac{\partial B}{\partial r}$ and rearranging yields:

$$\frac{\partial \mathcal{D}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{D}}{\partial r^2} + \left(\frac{1}{2}\sigma^2 + a - br\right)\frac{\partial \mathcal{D}}{\partial r} - (r + b)\mathcal{D} - B.$$

Duration CSE Domain Transformation

Consider the following domain transformation identities from 3.8:

$$\frac{\partial \mathcal{D}}{\partial r} = (1-y)^2 \frac{\partial \mathcal{D}}{\partial y}, \quad \frac{\partial^2 \mathcal{D}}{\partial r^2} = -2(1-y)^3 \frac{\partial \mathcal{D}}{\partial y} + (1-y)^4 \frac{\partial^2 \mathcal{D}}{\partial y^2}. \tag{C.3}$$

Applying the transformation to the Duration CSE yields:

$$\frac{\partial \mathcal{D}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{D}}{\partial y^2} + \left(\frac{1}{2}\sigma^2 (1-y)^2 - \sigma^2 y(1-y)^2 + a(1-y)^2 - by(1-y)\right)\frac{\partial \mathcal{D}}{\partial y}$$
$$- \left(\frac{y}{1-y} + b\right)\mathcal{D} - B.$$

C.2.2 Convexity CSE

In the similar case to Gamma, Convexity can be derived by differentiating C.2 again with respect to the interest rate.

$$\frac{\partial^3 B}{\partial r^2 \partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^4 B}{\partial r^4} + (\sigma^2 + a - br)\frac{\partial^3 B}{\partial r^3} - (2b + r)\frac{\partial^2 B}{\partial r^2} - 2\frac{\partial B}{\partial r}.$$

Making the substitution: $\mathcal{C} = \frac{\partial^2 B}{\partial r^2}$ and rearranging yields:

$$\frac{\partial \mathcal{C}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{C}}{\partial r^2} + (\sigma^2 + a - br)\frac{\partial \mathcal{C}}{\partial r} - (2b + r)\mathcal{C} - 2\frac{\partial B}{\partial r}.$$

Convexity CSE Domain Transformation

Applying the domain transformation identities (3.8) results in:

$$\frac{\partial \mathcal{C}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{C}}{\partial y^2} + (\sigma^2(1-y)^3 + a(1-y)^2 - by(1-y))\frac{\partial \mathcal{C}}{\partial y}$$
$$- \left(2b + \frac{y}{1-y}\right)\mathcal{C} - 2(1-y)^2 \frac{\partial B}{\partial y}.$$

C.2.3 Speed of Adjustment Sensitivity CSE

Differentiating the CIR zero coupon bond PDE with respect to b (κ) leads to the following:

$$\frac{\partial^2 B}{\partial b \partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial b \partial r^2} + (a - br)\frac{\partial^2 B}{\partial b \partial r} - r\frac{\partial B}{\partial b} - r\frac{\partial B}{\partial r}.$$

Making the substitution $\mathcal{K} = \frac{\partial B}{\partial b}$ and rearranging yields:

$$\frac{\partial \mathcal{K}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{K}}{\partial r^2} + (a - br)\frac{\partial \mathcal{K}}{\partial r} - r\mathcal{K} - r\frac{\partial B}{\partial r}.$$

Speed Of Adjustment CSE Domain Transformation

Applying domain transformation to the above CSE yields:

$$\frac{\partial \mathcal{K}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3 \frac{\partial^2 \mathcal{K}}{\partial y^2} + (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)\frac{\partial \mathcal{K}}{\partial y}$$
$$- \left(\frac{y}{1-y}\right)\mathcal{K} - y(1-y)\frac{\partial B}{\partial y}.$$

C.2.4 Volatility Sensitivity CSE

Differentiating with respect to the volatility parameter (σ) yields:

$$\frac{\partial^2 B}{\partial \sigma \partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 B}{\partial \sigma \partial r^2} + \sigma r \frac{\partial^2 B}{\partial r^2} + (a - br)\frac{\partial^2 B}{\partial \sigma \partial r} - r\frac{\partial B}{\partial \sigma}.$$

Applying the following substitution $\mathcal{V} = \frac{\partial B}{\partial \sigma}$ and rearranging yields:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 r \frac{\partial^2 \mathcal{V}}{\partial r^2} + (a - br)\frac{\partial \mathcal{V}}{\partial r} - r\mathcal{V} + \sigma r \frac{\partial^2 B}{\partial r^2}.$$

Volatility CSE Domain Transformation

Applying domain transformation to the above CSE yields:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 y(1-y)^3\frac{\partial^2 \mathcal{V}}{\partial y^2} + (a(1-y)^2 - by(1-y) - \sigma^2 y(1-y)^2)\frac{\partial \mathcal{V}}{\partial y}$$
$$- \Big(\frac{y}{1-y}\Big)\mathcal{V} + \sigma r\Big(-2(1-y)^3\frac{\partial B}{\partial y} + (1-y)^4\frac{\partial^2 B}{\partial y^2}\Big).$$

C.3 Domain Transformation of the Black-Scholes Equation

Consider the Black-Scholes equation:

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV.$$

Using the domain transformation (3.1), the following identities are obtained through the use of the chain rule:

$$S = \frac{y}{1-y}, \quad \frac{\partial V}{\partial S} = (1-y)^2\frac{\partial V}{\partial y}, \quad \frac{\partial^2 V}{\partial S^2} = -2(1-y)^3\frac{\partial V}{\partial y} + (1-y)^4\frac{\partial^2 V}{\partial y^2}. \tag{C.4}$$

$$\frac{\partial V}{\partial t} = \frac{1}{2}\sigma^2\Big(\frac{y}{1-y}\Big)^2\Big(-2(1-y)^3\frac{\partial V}{\partial y} + (1-y)^4\frac{\partial^2 V}{\partial y^2}\Big) + r\Big(\frac{y}{1-y}\Big)(1-y)^2\frac{\partial V}{\partial y} - rV.$$

$$= \frac{1}{2}\sigma^2 y^2(1-y)^2\frac{\partial^2 V}{\partial y^2} + \big(ry(1-y) - \sigma^2 y^2(1-y)\big)\frac{\partial V}{\partial y} - rV.$$

Where the payoff function is given by:

$$V\Big(\Big(\frac{y}{1-y}\Big), T\Big) = \max\Big(\Big(\frac{y}{1-y}\Big) - K, 0\Big).$$

C.3.1 Transformed Vega CSE

Considering the standard Vega CSE:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 S^2\frac{\partial^2 \mathcal{V}}{\partial S^2} + rS\frac{\partial \mathcal{V}}{\partial S} - r\mathcal{V} + \sigma S^2\frac{\partial^2 V}{\partial S^2}.$$

Similar transformations to C.4 can be applied to yield:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2\Big(\frac{y}{1-y}\Big)^2\Big(-2(1-y)^3\frac{\partial \mathcal{V}}{\partial y} + (1-y)^4\frac{\partial^2 \mathcal{V}}{\partial y^2}\Big) + r\Big(\frac{y}{1-y}\Big)(1-y)^2\frac{\partial \mathcal{V}}{\partial y}$$
$$- r\mathcal{V} + \sigma\Big(\frac{y}{1-y}\Big)^2\Big(-2(1-y)^3\frac{\partial V}{\partial y} + (1-y)^4\frac{\partial^2 V}{\partial y^2}\Big).$$

Rearranging yields the transformed Vega CSE defined on a spatial unit interval:

$$\frac{\partial \mathcal{V}}{\partial t} = \frac{1}{2}\sigma^2 y^2(1-y)^2\frac{\partial^2 \mathcal{V}}{\partial y^2} + \Big(ry(1-y) - \sigma^2 y^2(1-y)\Big)\frac{\partial \mathcal{V}}{\partial y} - r\mathcal{V}$$
$$+ \sigma y^2(1-y)^2\frac{\partial^2 V}{\partial y^2} - 2\sigma y^2(1-y)\frac{\partial V}{\partial y}.$$

C.3.2 Transformed Rho CSE

Following the same procedure as in chapter D.3.1, consider the Rho CSE:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2 S^2\frac{\partial^2 \rho}{\partial S^2} + rS\frac{\partial \rho}{\partial S} - r\rho + S\frac{\partial V}{\partial S} - V.$$

Applying similar domain transformation identities to C.4, the following is obtained:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2\Big(\frac{y}{1-y}\Big)^2\Big(-2(1-y)^3\frac{\partial \rho}{\partial y} + (1-y)^4\frac{\partial^2 \rho}{\partial y^2}\Big) + r\Big(\frac{y}{y-1}\Big)\Big((1-y)^2\frac{\partial \rho}{\partial y}\Big)$$
$$- r\rho - V + \Big(\frac{y}{1-y}\Big)(1-y)^2\frac{\partial V}{\partial y}.$$

Rearranging yields:

$$\frac{\partial \rho}{\partial t} = \frac{1}{2}\sigma^2 y^2(1-y)^2\frac{\partial^2 \rho}{\partial y^2} + \Big(ry(1-y) - \sigma^2 y^2(1-y)\Big)\frac{\partial \rho}{\partial y} - r\rho - V + y(1-y)\frac{\partial V}{\partial y}.$$

D C++ Code Appendix

D.1 Crank Nicolson Definitions C++ Code

```
15 CNIBVP::CNIBVP(IBvp& source, long NSteps, long JSteps)
16     : IBvpSolver(source, NSteps, JSteps),
17         A(vec(J + 1)), B(vec(J + 1)), C(vec(J + 1)), F(vec(J + 1)),
18             inA(vec(J + 1)), inB(vec(J + 1)), inC(vec(J + 1)), inF(vec(J + 1))
19 {}
20
21 void CNIBVP::ThomeeScheme()
22 {
23     vec check = { 0,1,2,3,4,8};
24     if (std::find(check.begin(), check.end(), ibvp->Signature()) == check.end())
25     {
26         inhomogOld[0] = 1.0; //(Zero_Coupon Bond)
27     }
28     else if (ibvp->Signature() == 8)
29     {
30         vecOld[0] = 1.0;
31     }
32 }
33
34 val_type CNIBVP::NearField()
35 {
36     val_type lftbnd, diffval, convval, reacval;
37     std::size_t i = 1;
38     if (ibvp->Signature() == 8)
39     {
40         lftbnd = ibvp->LftBnd(tnow);
41         diffval = ibvp->Diffusion(xarr[i], tnow);
42         convval = ibvp->Convection(xarr[i], tnow);
43         reacval = ibvp->Reaction(xarr[i], tnow);
44     }
45     else
46     {
47         lftbnd = ibvp->inLftBnd(tnow);
48         diffval = ibvp->inDiffusion(xarr[i], tnow);
49         convval = ibvp->inConvection(xarr[i], tnow);
50         reacval = ibvp->inReaction(xarr[i], tnow);
51     }
52
53     Lam = (lftbnd * k) / h; // Thomee Lambda Term
54     val_type t1, t2, t3, ptvLam, ntvLam, temp0, temp1, temp2, temp3;
55
56     t1 = 0.5 * k * diffval;
57     t2 = 0.25 * k * h * convval;
58     t3 = 0.5 * k * h2 * reacval;
59
60     // First row of matrix
61     ptvLam = (1.0 + Lam);
62     ntvLam = (1.0 - Lam);
63
64     // Second row of matrix
65     temp0 = t1 - t2;
66     temp1 = -h2 - 2 * t1 + t3;
67
68     // Create LHS Matrix components.
69     vec Ldiag = { temp0 };
70     vec Mdiag = { ptvLam, temp1 };
71     vec Udiag = { ntvLam };
72
73     // Create RHS Matrix
74     temp2 = ntvLam * vecOld[0] + ptvLam * vecOld[1];
75     temp3 = (-t1 + t2) * vecOld[0] + (-h2 + t1 - t3) * vecOld[1] + (-t1 - t2) * vecOld[2];
76     vec RHS = { temp2, temp3 };
77
78     GausElim<val_type> matSolver(Ldiag, Mdiag, Udiag, RHS);
79     vec M = matSolver.Solver();
80
81     vecNew[0] = M.at(0);
82     return M.at(0);
83 }
```

FIGURE D.1: C++ Crank Nicolson Definitions File

```
87  inline val_type CNIBVP::InhomogTerm(val_type x, val_type t, int i) const
88  {
89      int type = ibvp->Signature();
90      val_type source = ibvp->Inhomog(x, t);
91
92      // Inhomogeneous Terms
93      if (type == 0){ return 0.0; }
94      else if (type == 1) { return 0.0; }
95      else if (type == 2) { return 0.0; }
96      else if (type == 3) // Vega
97      {
98          return -source * (1.0 / h2)* (inhomogOld[i - 1] - (2.0 * inhomogOld[i]) + inhomogOld[i + 1]
99              + inhomogNew[i - 1] - (2.0 * inhomogNew[i]) + inhomogNew[i + 1]);
100     }
101     else if (type == 4) // Rho
102     {
103         return ((inhomogNew[i] - inhomogOld[i]) - ((source * 0.5) / h) * (inhomogOld[i + 1] -
104             inhomogOld[i - 1] + inhomogNew[i + 1] - inhomogNew[i - 1]));
105     }
106     else if (type == 5) // Duration
107     {
108         return source * (inhomogNew[i] - inhomogOld[i]);
109     }
110     else if (type == 6) // Convexity
111     {
112         return -(source / h) * (inhomogOld[i + 1] - inhomogOld[i - 1] + inhomogNew[i + 1] - inhomogNew[i - 1]);
113     }
114     else if (type == 7) // Transformed Delta
115     {
116         return source * (1.0 / h2) * (inhomogOld[i - 1] - (2.0 * inhomogOld[i]) + inhomogOld[i + 1]
117             + inhomogNew[i - 1] - (2.0 * inhomogNew[i]) + inhomogNew[i + 1]) - ((ibvp->DiffSig(x, t) * 0.5) / h) * (inhomogOld[i + 1] -
118                 inhomogOld[i - 1] + inhomogNew[i + 1] - inhomogNew[i - 1]);
119     }
120     else if (type == 8) { return 0.0; }
121 }
122
123 void CNIBVP::calculate()
124 { // Tells how to calculate sol. at n+1
125
126     // If there exists a source term in the given PDE which is option price based.
127     // CN will be applied to the standard BlackScholes equation to produce a vector
128     // of option prices to calculate the RHS.
129     val_type RHS = ibvp->Inhomog(ibvp->xrange().high(), ibvp->trange().high());
130     if (RHS != 0)
131     {
132         CrankInhomog();
133     }
134
135     double t1, t2, t3, Low, Mid, Upp;
136
137     for (std::size_t i = 1; i < F.size() - 1; ++i)
138     {
139         t1 = (0.5 * k * ibvp->Diffusion(xarr[i], tnow));
140         t2 = 0.25 * k * h * ibvp->Convection(xarr[i], tnow);
141         t3 = 0.5 * k * h2 * ibvp->Reaction(xarr[i], tnow);
142
143         // Coefficients of the U terms
144         A[i] = t1 - t2;             // Lower Diagonal
145         B[i] = -h2 - 2.0 * t1 + t3;     // Domin Diagonal
146         C[i] = t1 + t2;             // Upper Diagonal
147
148         // Coefficients of the U terms
149         double t1A = 0.5 * k * ibvp->Diffusion(xarr[i], tprev);
150         double t2A = 0.25 * k * h * ibvp->Convection(xarr[i], tprev);
151         double t3A = 0.5 * k * h2 * ibvp->Reaction(xarr[i], tprev);
152
153         Low = -t1A + t2A;
154         Mid = -h2 + 2.0 * t1A - t3A;
155         Upp = -t1A - t2A;
```

FIGURE D.2: C++ Crank Nicolson Definitions File

```
157         // Approximate Option Price wrt the source term.
158         F[i] = Low * vecOld[i - 1] + Mid * vecOld[i] + Upp * vecOld[i + 1]
159             + 0.5 * k * h2 * InhomogTerm(xarr[i], tnow, i);
160 
161     }
162 
163     // Define boundary conditions
164     val_type BCL;
165     val_type BCR = ibvp->RhtBnd(tnow);
166 
167     // If statement to calculate boundary condition if CIR process is in use.
168     int Typ = ibvp->Signature();
169     if (Typ == 8)
170     {   BCL = NearField(); }
171     else
172     {   BCL = ibvp->LftBnd(tnow); }
173 
174     // Create option template for the double sweep tridiagonal matrix solver
175     DoubleSweep<double> mySolver(A, B, C, F, BCL, BCR);
176 
177     vecNew = mySolver.solve();
178 }
179 
180 // Method use to apply CN to BlackScholes PDE to obtain option prices for use
181 // in the RHS term.
182 
183 void CNIBVP::CrankInhomog()
184 {
185     val_type int1, int2, int3, inLow, inMid, inUpp;
186 
187     for (std::size_t j = 1; j < inF.size() - 1; ++j)
188     {
189         val_type xval = xarr[j];
190 
191         int1 = 0.5 * k * ibvp->inDiffusion(xval, tnow);
192         int2 = 0.25 * k * h * ibvp->inConvection(xval, tnow);
193         int3 = 0.5 * k * h2 * ibvp->inReaction(xval, tnow);
194 
195         inA[j] = int1 - int2;
196         inB[j] = -h2 - 2.0 * int1 + int3;
197         inC[j] = int1 + int2;
198 
199         // Coefficients of the U terms
200         double int1A = 0.5 * k * ibvp->inDiffusion(xval, tprev);
201         double int2A = 0.25 * k * h * ibvp->inConvection(xval, tprev);
202         double int3A = 0.5 * k * h2 * ibvp->inReaction(xval, tprev);
203 
204         inLow = -int1A + int2A;
205         inMid = -h2 + 2.0 * int1A - int3A;
206         inUpp = -int1A - int2A;
207         inF[j] = inLow * inhomogOld[j - 1] + inMid * inhomogOld[j]
208             + inUpp * inhomogOld[j + 1];
209     }
210 
211     double inBCL;
212     double inBCR = ibvp->inRhtBnd(tnow);
213 
214     vec check = { 0,1,2,3,4 };
215     if (std::find(check.begin(), check.end(), ibvp->Signature()) == check.end())
216     {
217         inBCL = NearField();
218     }
219     else
220     {
221         inBCL = ibvp->inLftBnd(tnow);
222     }
223 
224     // Create option template for the double sweep tridiagonal matrix solver
225     DoubleSweep<double> InSolver(inA, inB, inC, inF, inBCL, inBCR);
226     inhomogNew = InSolver.solve();
227 }
```

FIGURE D.3: C++ Crank Nicolson Definitions File

D.2 Method Of Lines Definitions C++ Code

```
6   MethodOfLines::MethodOfLines(IBvp& source, long NSteps, long JSteps)
7   {
8       ibvp = &source;
9
10      // Initiatise Mesh Interval
11      N = NSteps; J = JSteps;
12
13      // Time Components
14      T = ibvp->trange().spread();
15      T0 = ibvp->trange().low();
16      k = T / static_cast<val_type>(N); // double
17
18      // Space Components
19      h = ibvp->xrange().spread() / static_cast<val_type>(J);
20      h2 = 1.0 / (h * h);
21      hm1 = 1.0 / (2.0 * h);
22
23      // Inhomogeneous vector set to zero vector.
24      inhomog.resize(J + 1, 0.0);
25
26      // Mesh Creation
27      // Use the mesh method in xrange to create a grid in the x direction
28      xarr = ibvp->xrange().mesh(J);
29
30      // Array in t direction
31      tarr = ibvp->trange().mesh(N);
32  }
33
34  MethodOfLines::~MethodOfLines()
35  {}
36
37  // Method to create initial conditions and subsequently output IC vector.
38  state_vec MethodOfLines::IntCnd()
39  {
40      state_vec U(J + 1, 0.0);
41
42      // Apply boundaries
43      U[0] = ibvp->LftBnd(T0); // Left
44      U[U.size() - 1] = ibvp->RhtBnd(ibvp->trange().high());
45
46      // Fill in Domain
47      for (std::size_t j = 1; j < xarr.size() - 1; ++j)
48      {
49          U[j] = ibvp->IntCnd(xarr[j]);
50      }
51      return U;
52  }
53
```

FIGURE D.4: C++ Method Of Lines Definitions File

```
 54 state_mat MethodOfLines::MatIntCnd()
 55 {
 56     // Creates a matrix of zeros (J+1 X 2)
 57     state_mat U(J + 1, 2, 0.0);
 58
 59     // Apply boundaries for both the source and the sensitivity vectors
 60     U(0, 0) = ibvp->inLftBnd(0.0);  U(U.size1() - 1, 0) = ibvp->inRhtBnd(0.0);
 61     U(0, 1) = ibvp->LftBnd(0.0);    U(U.size1() - 1, 1) = ibvp->RhtBnd(0.0);
 62
 63     // Fill in Domain
 64     for (std::size_t j = 1; j < U.size1() - 1; ++j)
 65     {
 66         U(j, 0) = ibvp->inIntCnd(xarr[j]);
 67         U(j, 1) = ibvp->IntCnd(xarr[j]);
 68     }
 69     return U;
 70 }
 71
 72 // System Methods
 73 // --------------
 74
 75 // Standard MOL procedure
 76 void MethodOfLines::operator()(const state_vec& U, state_vec& dUdt,
 77     const val_type t)
 78 {
 79     // Define Variables
 80     val_type df, cn, rc;
 81
 82     // Size variable
 83     std::size_t pnt = 1;
 84
 85     // Left Boundary Condition
 86     dUdt[pnt] = ibvp->Diffusion(xarr[pnt], t) * h2 * (ibvp->LftBnd(t) - 2.0 * U[pnt]
 87         + U[pnt + 1]) + 0.5 * ibvp->Convection(xarr[pnt], t) * (U[pnt + 1]
 88         - ibvp->LftBnd(t)) / h + ibvp->Reaction(xarr[pnt], t) * U[pnt];
 89
 90     // Size variable
 91     pnt = U.size() - 1;
 92
 93     // Right Boundary Condition
 94     dUdt[pnt] = ibvp->Diffusion(xarr[pnt], t) * h2 * (U[pnt - 1] - 2.0 * U[pnt]
 95         + ibvp->RhtBnd(t)) + 0.5 * ibvp->Convection(xarr[pnt], t) * (ibvp->RhtBnd(t)
 96         - U[pnt - 1]) / h + ibvp->Reaction(xarr[pnt], t) * U[pnt];
 97
 98     // Fill in the interior of the domain.
 99     for (std::size_t pnt = 2; pnt < U.size() - 2; pnt += 1)
100     {
101         df = ibvp->Diffusion(xarr[pnt], t) * h2;
102         cn = ibvp->Convection(xarr[pnt], t) * hm1;
103         rc = ibvp->Reaction(xarr[pnt], t);
104
105         // Using the simplified MOL D/C/R equation..
106         dUdt[pnt] = (df + cn) * U[pnt + 1] + (-2.0 * df + rc) * U[pnt]
107             + (df - cn) * U[pnt - 1];
108     }
109 }
110
```

FIGURE D.5: C++ Method Of Lines Definitions File

```
111 //Inhomogeneous term equations require a matrix calculation.
112 void MethodOfLines::operator()(const state_mat& U, state_mat& dUdt, const val_type t)
113 {
114     val_type df, cn, rc;
115
116     //-------------------------------------------------------------------------------
117     //                              Source Term
118     //-------------------------------------------------------------------------------
119
120     std::size_t pnt = 1;
121
122     dUdt(pnt, 0) = ibvp->inDiffusion(xarr[pnt], t) * h2 * (ibvp->inLftBnd(t)
123         - 2.0 * U(pnt, 0) + U(pnt + 1, 0)) + 0.5 * ibvp->inConvection(xarr[pnt], t)
124         * (U(pnt + 1, 0) - ibvp->inLftBnd(t)) / h + ibvp->inReaction(xarr[pnt], t)
125         * U(pnt, 0);
126
127     pnt = U.size1() - 2;
128
129     dUdt(pnt, 0) = ibvp->inDiffusion(xarr[pnt], t) * h2 * (U(pnt - 1, 0)
130         - 2.0 * U(pnt, 0) + ibvp->inRhtBnd(t)) + 0.5 * ibvp->inConvection(xarr[pnt], t)
131         * (ibvp->inRhtBnd(t) - U(pnt - 1,0)) / h + ibvp->inReaction(xarr[pnt], t)
132         * U(pnt, 0);
133
134     // Interior Domain
135     for (std::size_t pnt = 2; pnt < U.size1() - 2; pnt += 1)
136     {
137         df = ibvp->inDiffusion(xarr[pnt], t) * h2;
138         cn = ibvp->inConvection(xarr[pnt], t) * hm1;
139         rc = ibvp->inReaction(xarr[pnt], t);
140
141         dUdt(pnt, 0) = (df + cn) * U(pnt + 1, 0) + (-2.0 * df + rc) * U(pnt, 0)
142             + (df - cn) * U(pnt - 1, 0);
143     }
144
145
146     //-------------------------------------------------------------------------------
147     //                             Sensitivity
148     //-------------------------------------------------------------------------------
149
150     pnt = 1;
151
152     dUdt(pnt, 1) = ibvp->inDiffusion(xarr[pnt], t) * h2 * (ibvp->inLftBnd(t)
153         - 2.0 * U(pnt, 1) + U(pnt + 1, 1)) + 0.5 * ibvp->inConvection(xarr[pnt], t)
154         * (U(pnt + 1, 1) - ibvp->inLftBnd(t)) / h + ibvp->inReaction(xarr[pnt], t)
155         * U(pnt, 1) + InhomogTerm(xarr[pnt], t, pnt, Lft, U);
156
157     pnt = U.size1() - 2;
158
159     dUdt(pnt, 1) = ibvp->inDiffusion(xarr[pnt], t) * h2 * (U(pnt - 1, 1)
160         - 2.0 * U(pnt, 1) + ibvp->inRhtBnd(t)) + 0.5 * ibvp->inConvection(xarr[pnt], t)
161         * (ibvp->inRhtBnd(t) - U(pnt - 1, 1)) / h + ibvp->inReaction(xarr[pnt], t)
162         * U(pnt, 1) + InhomogTerm(xarr[pnt], t, pnt, Rht, U);
163
164
165
166     for (std::size_t pnt = 2; pnt < U.size1() - 2; pnt += 1)
167     {
168         df = ibvp->Diffusion(xarr[pnt], t) * h2;
169         cn = ibvp->Convection(xarr[pnt], t) * hm1;
170         rc = ibvp->Reaction(xarr[pnt], t);
171
172         dUdt(pnt, 1) = (df + cn) * U(pnt + 1, 1) + (-2.0 * df + rc) * U(pnt, 1)
173             + (df - cn) * U(pnt - 1, 1)
174             + InhomogTerm(xarr[pnt], t, pnt, Inner, U);
175     }
176
177 }
178
```

FIGURE D.6: C++ Method Of Lines Definitions File

```
205 inline val_type MethodOfLines::InhomogTerm(val_type x, val_type t, std::size_t pnt,
206     bnd bound, state_mat U) const
207 {
208
209     // Identify the PDE type...
210     int type = ibvp->Signature();
211     val_type source = ibvp->Inhomog(x, t);
212
213     if (type == 0) { return 0.0; }
214     else if (type == 1) { return 0.0; }
215     else if (type == 2) { return 0.0; }
216     else if (type == 3)
217     {
218         if (bound == Lft)
219         {
220             return source * h2 * (ibvp->inLftBnd(t) - 2.0 * U(pnt, 0) + U(pnt + 1, 0));
221         }
222         else if (bound == Rht)
223         {
224             return source * h2 * (U(pnt - 1, 0) - 2.0 * U(pnt, 0) + ibvp->inRhtBnd(t));
225         }
226         else
227         {
228             return source * h2 * (U(pnt - 1, 0) - 2.0 * U(pnt, 0) + U(pnt + 1, 0));
229         }
230     }
231     else if (type == 4)
232     {
233         if (bound == Lft)
234         {
235             return source * hm1 * (ibvp->inLftBnd(t) + U(pnt + 1, 0))  - U(pnt, 0);
236         }
237         else if (bound == Rht)
238         {
239             return source * hm1 * (U(pnt - 1, 0) + ibvp->inRhtBnd(t)) - U(pnt, 0);;
240         }
241         else
242         {
243             return source * hm1 * (U(pnt - 1, 0)  + U(pnt + 1, 0)) - U(pnt, 0);
244         }
245     }
246     else if (type == 9) // hm1 = 1/2h     h2 = 1/h^2
247     {
248         if (bound == Lft)
249         {
250             return (source * (1.0 - x) * (1.0 - x) * h2 * (U(pnt + 1, 0) - 2 * U(pnt, 0) + ibvp-
>inLftBnd(t)))
251                 - 2.0 * source * (1.0 - x) * hm1 * (U(pnt + 1, 0) - ibvp->inLftBnd(t));
252         }
253         else if (bound == Rht)
254         {
255             return (source * (1.0 - x) * (1.0 - x) * h2 * (ibvp->inRhtBnd(t) - 2 * U(pnt, 0) + U(pnt -
1, 0)))
256                 - 2.0 * source * (1.0 - x) * hm1 * (ibvp->inRhtBnd(t) - U(pnt - 1, 0));
257         }
258         else // source = sig y^2
259         {
260             return (source * (1.0 - x) * (1.0 - x) * h2 * (U(pnt + 1, 0) - 2 * U(pnt, 0) + U(pnt - 1,
0)))
261                 - 2.0 * source * (1.0 - x) * hm1 * (U(pnt + 1, 0) - U(pnt - 1, 0));
262         }
263     }
264     else if (type == 10)
265     {
266         if (bound == Lft)
267         {
268             return source * hm1 * (U(pnt + 1, 0) - ibvp->inLftBnd(t)) - U(pnt, 0);
269         }
270         else if (bound == Rht)
271         {
272             return source * hm1* (ibvp->inRhtBnd(t) - U(pnt - 1, 0)) - U(pnt, 0);
273         }
```

FIGURE D.7: C++ Method Of Lines Definitions File

D.3 Black-Scholes Forward AD C++ Code

```
4   #include <boost/math/differentiation/autodiff.hpp>
5   using namespace boost::math::differentiation;
6
7   class BsForwardAD
8   {
9   public:
10      BsForwardAD(char type, double K, const double price, const double sigma, const double tau,
11          const double rate) : variable(K), target(price)
12      {
13          Calculate(type, K, price, sigma, tau, rate);
14      }
15
16      void Calculate(char type, double K, const double price, const double sigma, const double tau,
17          const double rate)
18      {
19          auto const variables = make_ftuple<double, 2, 1, 1, 1>
20              (price, sigma, tau, rate);
21          auto const& underlying = std::get<0>(variables);
22          auto const& sig = std::get<1>(variables);
23          auto const& T = std::get<2>(variables);
24          auto const& r = std::get<3>(variables);
25
26          auto output = [](char type, auto target, auto sens1, auto sens2, auto sens3,
27              auto sens4, auto sens5)
28          {
29              std::cout << "\n\n======================================================\n";
30              std::cout << "                          Forward AD\n";
31              std::cout << "======================================================\n\n";
32              std::cout << std::setprecision(std::numeric_limits<double>::digits10)
33                  << "S = " << target << ", Delta: " << sens1 << "\n"
34                  << "S = " << target << ", Gamma: " << sens2 << "\n"
35                  << "S = " << target << ", Theta: " << sens3 << "\n"
36                  << "S = " << target << ", Rho:   " << sens4 << "\n"
37                  << "S = " << target << ", Vega:  " << sens5 << "\n";
38          };
39
40          if (type == 'C')
41          {
42              auto const call_price = BlackScholesPrice('C', variable, underlying, sig, T, r);
43              double const adDelta = call_price.derivative(1, 0, 0, 0);
44              double const adGamma = call_price.derivative(2, 0, 0, 0);
45              double const adTheta = -call_price.derivative(0, 0, 1, 0);
46              double const adRho = call_price.derivative(0, 0, 0, 1);
47              double const adVega = call_price.derivative(0, 1, 0, 0);
48              output(type, price, adDelta, adGamma, adTheta, adRho, adVega);
49          }
50          else
51          {
52              auto const put_price = BlackScholesPrice('P', variable, underlying, sig, T, r);
53              double const adDelta = put_price.derivative(1, 0, 0, 0);
54              double const adGamma = put_price.derivative(2, 0, 0, 0);
55              double const adTheta = -put_price.derivative(0, 0, 1, 0);
56              double const adRho = put_price.derivative(0, 0, 0, 1);
57              double const adVega = put_price.derivative(0, 1, 0, 0);
58              output(type, price, adDelta, adGamma, adTheta, adRho, adVega);
59          }
60      }
61
```

FIGURE D.8: C++ Black-Scholes Forward AD Header File

```
62     template<typename Price, typename Sigma, typename Tau, typename Rate>
63     promote<Price, Sigma, Tau, Rate> BlackScholesPrice(char cp, double K, Price const& S,
64         Sigma const& sigma, Tau const& tau, Rate const& r)
65     {
66         auto const d1 = (log(S / K) + (r + sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
67         auto const d2 = (log(S / K) + (r - sigma * sigma / 2) * tau) / (sigma * sqrt(tau));
68
69         if (cp == 'C')
70         {
71             return S * Phi(d1) - exp(-r * tau) * K * Phi(d2);
72         }
73         else if (cp == 'P')
74         {
75             return exp(-r * tau) * K * Phi(-d2) - S * Phi(-d1);
76         }
77     }
78
79     template <typename X>
80     X Phi(X const& x)
81     {
82         return 0.5 * erfc(-boost::math::constants::one_div_root_two<X>() * x);
83     }
84
85 private:
86     double variable;
87     double target;
88 };
89
```

FIGURE D.9: C++ Black-Scholes Forward AD Header File

E MATLAB Code Appendix

E.1 MATLAB Crank Nicolson Code

```
classdef CirCrank

    % Thesis parameters

    % a = 0.048      % T = 0.25      % sig = 0.4
    % b = 0.08       % r = 0.08

    properties (Access = private)
        interest; T; J; N        % CIR Parameters
        h2; k                    % Step Parameters
        t1; t2; t3; t4           % Crank Components
        pLam; nLam               % Box Method Components
        A; B                     % System Matrices
    end

    properties (Access = public)
        spatialVec               % X-axis vector
        priceMesh                % Mesh containing bond prices
        h                        % Step size
    end

    methods
        function obj = CirCrank(a, b, r, T, sig, J, N)
            % Define Step Parameters
            obj.J = J;
            obj.N = N;
            obj.interest = r;

            obj.k = T/N;
            obj.h = 1 / (J+1);
            obj.h2 = obj.h*obj.h;

            obj = initMatrix(obj);            % Initialise vectors/matricies
            obj = initPDE(obj, sig, a, b);    % Initialise PDE components
            obj = initBox(obj, a);            % Initialise Box components
        end
    end

    methods (Access = private)

        % Function used to initialise vectors used in Crank Nicolson
        function obj = initMatrix(obj)

            % Create pricing mesh (NxJ matrix) (preallocate memory)
            obj.priceMesh(1:obj.J+1, 1:obj.N+1) = zeros('int8');

            % Create spatial vector
            obj.spatialVec = 0:obj.h:obj.J/(obj.J + 1);

            % Initialise values in pricing mesh.
            obj.priceMesh(end, :) = 0; % Far-field Boundary

            % Intial Condition applied to near-field and domain values.
            obj.priceMesh(:, 1) = 1;

        end

        % Function used to initialise the PDE terms in the calculation
```

FIGURE E.1: C++ CN MATLAB Definitions File

```
        function obj = initPDE(obj, sig, a, b)

            % Diffusion Term
            diff = 0.5*sig*sig.* obj.spatialVec.*power((1-obj.spatialVec),3) * obj.k/obj.↙
h2;
            % Convection Term
            conv = 0.5*(obj.k/obj.h)*(a.*power(1.0-obj.spatialVec,2) - b.*obj.spatialVec.↙
*(1.0-obj.spatialVec) - sig*sig.*obj.spatialVec.*power(1.0-obj.spatialVec,2));
            % Reaction Term
            reac = - obj.k.* obj.spatialVec./(1-obj.spatialVec);

            % Create Crank Terms (associated to RHS of matrix)
            obj.t1 = 0.5*conv - 0.5*diff;
            obj.t2 = 1.0 + diff - 0.5*reac;
            obj.t3 = -(0.5*conv + 0.5*diff);
            obj.t4 = 1.0 - diff + 0.5*reac;

        end

        % Function to initialise Thomée Method components
        function obj = initBox(obj, a)

            % Thomée Components
            obj.pLam = 1 + (a*obj.k)/obj.h; %[1,1]
            obj.nLam = 1 - (a*obj.k)/obj.h; %[1,2]

            % Initialise Crank matrix LHS using above components
            obj.A(1,1) = obj.pLam;
            obj.A(1,2) = obj.nLam;

            % Initialise Crank matrix RHS using above components
            obj.B(1,1) = obj.nLam;
            obj.B(1,2) = obj.pLam;

        end
    end

    methods

        function [BondPrice, BondVec, PriceMatrix] = Result(obj)

            % Assign Crank matrices associated near-field condition
            obj.A(obj.J+1, obj.J+1) = 1;
            obj.B(obj.J+1, obj.J+1) = 0;

            % Assign solution

            % Assign t1, t2, t3 to LHS Crank Matrix A
            for j = 2:1:obj.J

                obj.A(j, j-1) = obj.t1(j); % lower diagonal
                obj.A(j, j) = obj.t2(j);   % diagonal
                obj.A(j, j+1) = obj.t3(j); % upper diagonal

            end

            % Assign t1, t4, t3 to RHS Crank Matrix B
            for j = 2:1:obj.J
```

FIGURE E.2: C++ CN MATLAB Definitions File

```
            obj.B(j, j-1) = -obj.t1(j); % lower diagonal
            obj.B(j, j) = obj.t4(j);   % diagonal
            obj.B(j, j+1) = -obj.t3(j); % upper diagonal

        end

        bndB(1:obj.J+1,1) = 0; % Create bnd vector for new vector cnd.

        % Take the pricing vector and iterate along each time step
        for n = 1:obj.N

            % Update boundaries after each iteration
            bndB(end) = obj.priceMesh(end,n+1);

            % Solver system using updated boundary and bond
            % approximations
            obj.priceMesh(:,n+1) = obj.A\(obj.B*obj.priceMesh(:,n)+bndB);

        end

        % Interpolate the results to obtain the bond price for a target
        % interest rate (i.e 0.08). First transform the variable for
        % compatibility on transformed domain.

        % Duffy transform identity
        y = obj.interest / (obj.interest + 1.0);

        % Interpolate
        BondPrice = interp1(obj.spatialVec, obj.priceMesh(:,end), y);

        % Used in CSE Calculations
        PriceMatrix = obj.priceMesh;
        BondVec = obj.priceMesh(:,end);
    end

    % Takes in a target interest value and returns approximations along
    % the time interval.

    function [ThetaHoldResult] = TimeVec(obj, PriceMatrix)

        % Identify target location
        y = obj.interest / (obj.interest + 1.0);
        location = round(y / obj.h);
        ThetaHoldResult = PriceMatrix(location, :);

    end


    end
end
```

FIGURE E.3: C++ CN MATLAB Definitions File

E.2 MATLAB Alternating Direction Explicit Code

```
classdef CirADE

    % Call constructor with required CIR parameters.
    % Call Result to obtain results.

    % Thesis parameters

    % a = 0.048      % T = 0.25      % sig = 0.4
    % b = 0.08       % r = 0.08

    properties (Access = private)
        interest; T; J; N       % CIR Parameters
        h2; k                   % Step Parameters
        U; UOld; V; VOld        % Create ADE solution vectors
        diff; conv; reac        % PDE d/c/r components
        denom1; denom2          % ADE up/downwind denominators
        pLam; nLam; sqr         % Box Method Components
    end

    properties (Access = public)
        spatialVec              % X-axis vector
        priceMesh               % Mesh containing bond prices
        h                       % Step parameter
    end

    methods
        function obj = CirADE(a, b, r, T, sig, J, N)
            % Define Step Parameters
            obj.J = J;
            obj.N = N;
            obj.interest = r;

            obj.k = T/N;
            obj.h = 1 / (J+1);
            obj.h2 = obj.h*obj.h;

            obj = initMatrix(obj);          % Initialise vectors/matricies
            obj = initPDE(obj, sig, a, b);  % Initialise PDE components
            obj = initBox(obj, a);          % Initialise Box components
        end
    end

    methods (Access = private)

        function obj = initMatrix(obj)
            % Create pricing mesh (NxJ matrix) (preallocate memory)
            obj.priceMesh(1:obj.J+1, 1:obj.N+1) = zeros('int8');

            % Create spatial vector
            obj.spatialVec = 0:obj.h:(obj.J/(obj.J + 1));

            % Initialise values in pricing mesh.
            obj.priceMesh(end, :) = 0; % Far-field Boundary

            % Intial Condition applied to near-field and domain values.
            obj.priceMesh(:, 1) = 1;

            % Initialise the crank nicolson solution vectors
```

FIGURE E.4: C++ CN MATLAB Definitions File

```
            obj.U(1:obj.J+1) = zeros;
            obj.V(1:obj.J+1) = zeros;

            % Apply initial and boundary conditions to old solution vecs
            obj.UOld(1,obj.J+1) = 0; % Far-field
            obj.VOld(1,obj.J+1) = 0; % Far-field

            obj.UOld(1:obj.J) = 1;
            obj.VOld(1:obj.J) = 1;
        end

        function obj = initPDE(obj, sig, a, b)

            % Diffusion Term
            obj.diff = 0.5*sig*sig.* obj.spatialVec.*power((1-obj.spatialVec),3) * obj.↙
k/obj.h2;
            % Convection Term
            obj.conv = 0.5*(obj.k/obj.h)*(a.*power(1.0-obj.spatialVec,2) - b.*obj.↙
spatialVec.*(1.0-obj.spatialVec) - sig*sig.*obj.spatialVec.*power(1.0-obj.spatialVec,2));
            % Reaction Term
            obj.reac = - obj.k.* obj.spatialVec./(1-obj.spatialVec);

            % Define the up/downwind denominators (find use in box scheme)
            obj.denom1 = 1.0 - obj.conv + obj.diff - obj.reac;
            obj.denom2 = 1.0 + obj.conv + obj.diff - obj.reac;

        end

        function obj = initBox(obj, a)

            % Thomée Components
            obj.pLam = 1 + (a*obj.k)/obj.h; %[1,1]
            obj.nLam = 1 - (a*obj.k)/obj.h; %[1,2]

            % Form 2x2 matrix (Using upwind)
            obj.sqr = [obj.pLam, obj.nLam; obj.conv(2) - obj.diff(2), obj.denom1(2)];

        end
    end

    methods

        function[BondPrice, BondVec, PriceMatrix] = Result(obj)

            % Iterate through each time step, calculating upwind and
            % downwind equation values in opposing spatial directions.

            % Start from 2 as the first step has been populated using
            % initial and boundary conditions.

            for n = 2:1:obj.N+1

                % Assign far-field boundary condition
                obj.U(obj.J+1) = obj.priceMesh(end,n);
                obj.V(obj.J+1) = obj.priceMesh(end,n);

                b_vec = [(obj.nLam*obj.UOld(1)+obj.pLam*obj.UOld(2)); (1-obj.diff(2)-obj.↙
conv(2))*obj.UOld(2)+(obj.diff(2)+obj.conv(2))*obj.UOld(3)];
```

FIGURE E.5: C++ CN MATLAB Definitions File

```
                % Solve system and apply boundary conditions
                NearBnd = obj.sqr\b_vec;

                obj.U(1) = NearBnd(1);  obj.U(2) = NearBnd(2);
                obj.V(1) = NearBnd(1);  obj.V(2) = NearBnd(2);

                % Upwind
                for j = 3:1:obj.J

                    obj.U(j) = (1 - obj.diff(j) - obj.conv(j))*obj.UOld(j) + (obj.diff(j)↙
- obj.conv(j))*obj.U(j-1)+(obj.diff(j) + obj.conv(j))*obj.UOld(j+1);
                    obj.U(j) = obj.U(j)/obj.denom1(j);

                end

                % Downwind
                for j = obj.J:-1:3

                    obj.V(j) = (1 - obj.diff(j) + obj.conv(j))*obj.VOld(j) + (obj.diff(j)↙
- obj.conv(j))*obj.VOld(j-1)+(obj.diff(j)+obj.conv(j))*obj.V(j+1);
                    obj.V(j) = obj.V(j)/obj.denom2(j);

                end

                % Average, fill price mesh column with averaged values.
                obj.priceMesh(:,n) = 0.5*(obj.V + obj.U);

                % Set new approximations to the old solution vector.
                obj.UOld = obj.priceMesh(:,n);
                obj.VOld = obj.priceMesh(:,n);
            end

            % Intepolate to obtain final values
            y = obj.interest / (1.0 + obj.interest);

            % Set solution variables.
            BondPrice = interp1(obj.spatialVec, obj.priceMesh(:,end), y);
            BondVec = obj.priceMesh(:,end);
            PriceMatrix = obj.priceMesh;
        end
    end
end
```

FIGURE E.6: C++ CN MATLAB Definitions File

References

[1] Rafael Abreu, Daniel Stich, and José Morales. "On the generalization of the Complex Step Method". *Journal of Computational and Applied Mathematics* 241 (2013), pp. 84–102. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2012.10.001. URL: http://www.sciencedirect.com/science/article/pii/S0377042712004207 (cit. on pp. 28, 67).

[2] K. Ahnert and M. Mulansky. *odeint: Integrate Functions*. 2015. URL: https://www.boost.org/doc/libs/1_65_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/odeint_in_detail/integrate_functions.html (visited on 08/22/2019) (cit. on p. 54).

[3] F Black and M. Scholes. "The Pricing of Options and Corporate Liabilities". *The Journal of Political Economy,* 81 (3 June 1973), pp. 637–654. URL: http://www.jstor.org/stable/1831029 (visited on 08/07/2019) (cit. on p. 2).

[4] M. Buckova Z. Ehrhardt and Gunther M. "Fichera Theory and Its Application in Finance". *Progress in Industrial Mathematics at ECMI 2014*. Vol. 22. Springer, Cham, July 2016. URL: https://doi.org/10.1007/978-3-319-23413-7_13 (cit. on pp. 10, 11).

[5] Zuzana Bučková, Matthias Ehrhardt, and Michael Günther. "Alternating direction explicit methods for convection diffusion equations". 84 (Sept. 2015), pp. 309–325 (cit. on pp. 17, 18).

[6] S. Byrne and A. Greenwell. *Automatic Differentiation for the Greeks*. 2017. URL: https://wilmott.com/automatic-for-the-greeks/ (visited on 08/22/2019) (cit. on p. 29).

[7] M. Choudhry. *Frequently Asked Questions in Quantitative Finance: Analysis and Valuation*. 1st ed. Bloomberg Press, 2005. URL: www.bloomberg.com/books (cit. on pp. 6, 7).

[8] J. Cox J. Ingersoll and S. Ross. "A Theory of the Term Structure of Interest Rates". *Econometrics,* 53 (2 Mar. 1985), pp. 385–407. URL: http://www.jstor.org/stable/1911242 (visited on 08/09/2019) (cit. on pp. 5, 6).

[9] J. Dormand and P. Prince. "A family of embedded Runge-Kutta formulae". *Journal of Computational and Applied Mathematics* 6.1 (1980), pp. 19–26. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0771-050X(80)90013-3. URL: http://www.sciencedirect.com/science/article/pii/0771050X80900133 (cit. on pp. 21, 22).

[10] D. Duffy. *Financial Instrument Pricing Using C++*. 2nd ed. John Wiley Sons, 2018. URL: https://www.wiley.com/en-gb/Financial+Instrument+Pricing+Using+C++,+2nd+Edition-p-9781119170488 (cit. on pp. 14, 18, 41, 46, 49, 51, 59).

[11] D Duffy. "Unconditionally Stable and Second-Order Accurate Explicit Finite Difference Schemes Using Domain Transformation: Part I One-Factor Equity Problems". Sept. 2009. URL: https://ssrn.com/abstract=1552926 (cit. on pp. 10, 17).

[12] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (cit. on p. 41).

[13] F. Gibson R. Lhabitant and D. Talay. "Modeling the Term Structure of Interest Rates: A Review of the Literature". *Foundations and Trends in Finance* 5 (2010), pp. 1–156. URL: http://dx.doi.org/10.1561/0500000032 (visited on 08/09/2019) (cit. on p. 17).

[14] S. Gleadall. *Option Greeks For Traders Part 1: Delta, Vega Theta*. 1st ed. Volcube, 2014. URL: http://www.volcube.com (cit. on p. 4).

[15] E. Haug. *The Complete Guide to Option Pricing Formulas*. 2nd ed. McGraw-Hill, 2009 (cit. on pp. 80–83).

[16] T Kimura. "On Dormand-Prince Method" (2009). URL: http://depa.fquim.unam.mx/amyd/archivero/DormandPrince_19856.pdf (visited on 08/22/2019) (cit. on pp. 21, 22).

[17] F Lu. "Alternating Direction Explicit Methods for Zero-coupon Bond Pricing in the Cox-Ingersoll-Ross Model". Supervisor: Daniel J. Duffy. MA thesis. University of Birmingham: Birmingham Business School, Sept. 2014 (cit. on pp. 10, 14–16, 18, 53).

[18] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley Professional Computing Series. Pearson Education, 2005. ISBN: 9780132702065. URL: https://books.google.co.uk/books?id=Qx5oyB49poYC (cit. on p. 29).

[19] Matthew Pulver. *autodiff: Automatic Differentiation C++ Library*. https://github.com/pulver/autodiff. 2019 (cit. on p. 60).

[20] E. Radkevic and O. Oleinik. *Second Order Equations with Non-negative Characteristic Form*. 1st ed. American Mathematical Society, 1973 (cit. on p. 11).

[21] W. Schiesser and G. Griffiths. *A Compendium of Partial Differential Equation Models: Method of Lines Analysis with Matlab*. 1st ed. New York, NY, USA: Cambridge University Press, 2009. ISBN: 0521519861, 9780521519861 (cit. on pp. 20, 21).

[22] W. Squire and G. Trapp. "Using Complex Variables to Estimate Derivatives of Real Functions". *SIAM Rev.* 40.1 (Mar. 1998), pp. 110–112. ISSN: 0036-1445. DOI: 10.1137/S003614459631241X. URL: http://dx.doi.org/10.1137/S003614459631241X (cit. on p. 28).

[23] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. 2nd ed. Springer-Verlag New York, 1993 (cit. on pp. 22, 26, 27).

[24] B. Towler and R. Yang. "Numerical stability of the classical and the modified Saul'yev's finite-difference methods". *Computers Chemical Engineering* 2.1 (1978), pp. 45–51. ISSN: 0098-1354. DOI: https://doi.org/10.1016/0098-1354(78)80006-4. URL: http://www.sciencedirect.com/science/article/pii/0098135478800064 (cit. on p. 18).

[25] J. Waldén. "On the approximation of singular source terms in differential equations". *Numerical Methods for Partial Differential Equations* 15.4 (1999), pp. 503–520. DOI: 10.1002/(SICI)1098-2426(199907)15:4<503::AID-NUM6>3.0.CO;2-Q. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%5C%28SICI%5C%291098-2426%5C%28199907%5C%2915%5C%3A4%5C%3C503%5C%3A%5C%3AAID-NUM6%5C%3E3.0.CO%5C%3B2-Q (cit. on p. 33).

[26] N Webber and J. James. *Frequently Asked Questions in Quantitative Finance: Analysis and Valuation*. 1st ed. John Wiley Sons, 2000 (cit. on p. 7).

[27] P. Wilmott. *Frequently Asked Questions in Quantitative Finance*. 2nd ed. John Wiley Sons, 2009. URL: https://www.wilmott.com (cit. on p. 2).

[28] P. Wilmott. *Paul Wilmott introduces Quantitative Finance*. 2nd ed. John Wiley Sons, 2007. URL: https://www.wilmott.com (cit. on p. 6).

[29] S. Wilmott P. Howison and J. Dewynne. *The Mathematics of Financial Derivatives: A Student Introduction*. 1st ed. Press Syndicate of the University of Cambridge, 1995 (cit. on p. 12).