Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London
Strand Campus, London,
United Kingdom

7CCSMPRJ

Individual Project Submission 2024

Name:	Raphael Frach
Student Number:	20023625
Degree Programme:	Computational Finance
Project Title:	Efficient solutions to Heston by ADI schemes
Supervisor:	Dr Riaz Ahmad
Word Count:	12085

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

☑ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: Raphael Frach **Date:** August 18, 2024

Department of Informatics
King's College London
United Kingdom

7CCSMPRJ Individual Project

Efficient solutions to Heston by ADI schemes

Name: **Raphael Frach**
Student Number: 20023625
Course: Computational Finance

Supervisor: Dr Riaz Ahmad

This dissertation is submitted for the degree of MSc in Computational Finance.

Acknowledgements

I would like to express my deepest gratitude to my supervisor and professor Dr Riaz Ahmad. Without his invaluable teaching and introduction to the Quantitative Methods in Finance module as well as his patience throughout, this project would have not been possible. I also thank my fellow students for their continuous feedback and advice.

Moreover, I am deeply indebted to my family, especially my sister and mum, for not only providing moral support but providing me with opportunities to get to this point.

Abstract

This dissertation focuses on the application of Alternating Direction Implicit (ADI) schemes to the Heston stochastic volatility model. The Heston partial differential equation is presented along with its closed form solution. Then finite difference methods are outlined, which coupled with the ADI schemes and a linear algebra based approach can lead to efficient solutions. Further, the options sensitivities (the Greeks) are studied as an extension.

The implementation demonstrates robust and efficient solutions by comparing to the closed form solution as well as another implementation. Global spatial and temporal discretisation errors align with the literature while also experimentation on upwinding schemes reveal their importance.

Nomenclature

S_t	Stock price at time t
v_t	Variance process
$C(S,T)$	European call option price
D	Dirichlet boundary matrix
$G(t)$	Boundary condition matrix
I_v	Identity matrix for variance direction
I_s	Identity matrix for underlying asset direction
K	Strike price
S_t	Stock price at time t
T	Maturity time
U_0	Initial condition vector
κ	Rate at which variance returns to θ
λ	Volatility risk premium
μ	Drift of the underlying asset
ρ	Correlation between the two Brownian motions
σ	Volatility of the variance process
θ	Long term average volatility

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In terms of pricing options, the key aspect has been to somehow capture the behaviour of stock prices which have generally been assumed to follow stochastic processes. This started in the 1970s with the Black-Scholes [1] model developed by Fischer Black, Robert Merton and Myron Scholes; the underlying asset was modelled by Geometric Brownian Motion:

$$dS_t = rS_t dt + \sigma S_t dW_t, \tag{1.1}$$

where S_0 is the current price of the stock, S_t is the price at a time t, while dW_t is known as a Wiener process. The stock price is driven by a drift rate r and a term to represent the volatility of the stock, σ. With this approach, closed-form solutions became available for European options resulting in many purposefully built Hewlett-Packard calculators to fill trading floors. The only seemingly unknown variable in (1.1) was σ, which could be selected to ensure that the model produced prices consistent with those observed in the market. This is known as implied volatility. Shortly after, traders began to notice discrepancies between both prices, where a so-called volatility smile (or more generally skew) was observed for different strike prices of contracts in the aftermath of the 1987 Black Monday crash. Consequently, mathematicians realised that treating σ as its own stochastic process could capture this smile, leading to a plethora of stochastic volatility models (e.g., Hull and White, Stein and Stein, CEV models) where this dissertation focuses on the famous Heston model developed in 1993.

A reason for Heston's popularity can come down to its semi-analytical solution for European options [2], which we study later on. Not only does this allow straight forward pricing, but also to verify the validity of numerical methods that can be extended for further tasks, like computing the Greeks. Furthermore, it allows for rather elegant calibration of the model's additional parameters, see for example Mikhailov and Nögel's approach [3].

1.2 Aims and Objectives

The main objective of this dissertation is to lay out a finite difference technique to value European options under the Heston model, with an emphasis on efficiency. We only implement this in Python on a standard personal computer however the design choices can simply be adapted to more dedicated environments. We further extend the approach to compute and plot the surfaces for a selection of the Greeks.

A challenge that has plagued derivative pricing, specifically for more complicated models such as Heston, is the need to price options and their Greeks in near-real-time. This is especially prevalent due to the growing derivatives market [4], where a multitude of

contracts need to be evaluated for their use in hedging tools or for traders. The Black-Scholes model, while being computationally more efficient, can lack due to its strong assumptions. To that extent, in Heston's framework we make use of a linear algebra based approach to the finite difference method. This is motivated by the properties, presented by Van Loan [5], of "The ubiquitous Kronecker product" and a presentation by Karel in't Hout [6] on their applications to the Heston model where we provide further vectorised design choices. These all aid in producing more efficient solutions.

The structure of this dissertation firstly aims to comprehensibly understand the semi-analytical solution of the Heston model in Chapter 2, with a short note on how to numerically compute the price for European call options. An imperative computation, as we mentioned above it will be used to check for validity of our approximation. As a by-product, we take note of Heston's partial differential equation (PDE) that is fundamental to the next chapter. In Chapter 3 the finite difference method employed is introduced theoretically where key points are communicated in an implementation specific manner. We make choices to aid in accuracy based on previous research in the field. Some are older techniques, such as finite difference discretisation on non-uniform grids that can be seen in textbooks such as "Computational Fluid Dynamics" by John D. Anderson [7] or more specific to our area of research "Pricing Financial Instruments: The Finite Difference Method" by Tavella and Randall [8]. Following this, we make use of a crucial revelation from fluid mechanics to use a time discretisation technique called the alternating direction implicit (ADI) method. They have relatively recently made their way to quantitative finance as numerically fast parabolic PDE solvers that circumvent many efficiency issues. Furthermore, newer ideas, like the aforementioned Kronecker product are coupled with these time-stepping schemes and a short extension of the Greeks are presented. Lastly, in Chapter 4 we provide numerical experiments to confirm reliable accuracy and touch on the orders of each technique used. Our program's speed is compared against that of another which has taken different theoretical and design choices.

Throughout this dissertation we aim to provide a cogent understanding of the implementation as some aspects, such as non-uniform grids and ADI methods can become rather complicated, also serious thought and care must be taken regarding the boundaries and their consequences. A literature survey is not included in this introduction as it is incorporated throughout the work, where relevant algorithms or approaches are discussed as they appear appropriately.

2 Background Theories

2.1 Heston Dynamics

The underlying asset, S_t, in the Heston model follows similarly to the well-known Black-Scholes case, however with stochastic variance v_t following a Cox-Ingersoll-Ross process (CIR) [9]. This results in the following representation of a system of bivariate stochastic differential equations (SDE).

$$dS_t = \mu S_t\, dt + \sqrt{v_t} S_t\, dW_{1,t} \tag{2.1}$$
$$dv_t = \kappa(\theta - v_t)\, dt + \sigma\sqrt{v_t}\, dW_{2,t} \tag{2.2}$$

where $\mathbb{E}^{\mathbb{P}}[dW_{1,t}dW_{2,t}] = \rho\, dt$. The correlation between the two Brownian motions W_1 and W_2 is $\rho \in [-1, 1]$.

- μ drift of the underlying asset;
- $\theta > 0$ long term average volatility;
- $\kappa > 0$ rate at which variance returns to θ;
- $\sigma > 0$ volatility of the variance process;
- $v_0 > 0$ initial variance;

2.1.1 Feller condition

The CIR process has a property called the Feller condition shown below

$$2\kappa\theta > \sigma^2. \tag{2.3}$$

If the above is satisfied, the variance process is guaranteed to be strictly positive while if not satisfied v_t may reach zero and become negative at some point. It is often not satisfied in practice so in later experiments we look at both scenarios.

2.1.2 Variance process

Rouah [10] makes the important note that the volatility $\sqrt{v_t}$ is not modelled directly, instead it is modelled through the variance v_t. If we set $h_t = \sqrt{v_t}$ the variance can be seen as an Ornstein–Uhlenbeck process

$$dh_t = -\beta h_t\, dt + \delta\, dW_{2,t} \tag{2.4}$$

Where applying Itô's Lemma to $v_t = h_t^2$ results in

$$dv_t = (\delta^2 - 2\beta v_t)dt + 2\delta\sqrt{v_t}dW_{2,t} \tag{2.5}$$

If we set the following parameters $\kappa := 2\beta, \theta := \delta^2/(2\beta)$, and $\sigma := 2\delta$, we can express (2.5) as the SDE (2.2).

2.2 Change of Measure

The stock price process, (2.1), and the variance process,(2.2), are defined under the physical (or real-world) $\mathbb{P}$-measure, but to price options we need S_t and v_t under the risk-neutral measure $\mathbb{Q}$. Wong and Heyde [11] give a more detailed and technical discussion on the change of measures for stochastic volatility models so let's step through some of the key points.

Firstly, we will explicitly write out Equation (2.1) and (2.2) as

$$dS_t = \mu^{\mathbb{P}} S_t\, dt + \sqrt{v_t} S_t\, dW_{1,t}^{\mathbb{P}}, \tag{2.6}$$

$$dv_t = \kappa^{\mathbb{P}} \left(\theta^{\mathbb{P}} - v_t\right) dt + \sigma^{\mathbb{P}} \sqrt{v_t} dW_{2,t}^{\mathbb{P}}, \tag{2.7}$$

where $\mathbb{E}^{\mathbb{P}}[dW_{1,t}^{\mathbb{P}} dW_{2,t}^{\mathbb{P}}] = \rho^{\mathbb{P}} dt$.

The following adjustments need to be made to allow any asset to evolve at the interest free rate r, these are known as market prices of risk (or sometimes Girsanov kernels)

$$\varphi_S = \frac{\left(\mu^{\mathbb{P}} - r\right) S_t}{\sqrt{v_t} S_t} = \frac{\mu^{\mathbb{P}} - r}{\sqrt{v_t}}, \tag{2.8}$$

$$\varphi_v = \frac{\lambda v_t}{\sigma^{\mathbb{P}} \sqrt{v_t}} = \frac{\lambda}{\sigma^{\mathbb{P}} \sqrt{v_t}}, \tag{2.9}$$

where λ is called the volatility risk premium, it can be calibrated with market prices however that could be a topic for another dissertation, we set it as zero. Further, we can change measures by considering the Radon-Nikodym derivative from Girsanov's theorem as follows

$$\frac{d\mathbb{Q}}{d\mathbb{P}} = \exp\left(- \int_0^t \varphi_S dW_{1,s}^{\mathbb{P}} - \int_0^t \varphi_v dW_{2,s}^{\mathbb{P}} + \int_0^t \varphi_v \varphi_S\, ds - \frac{1}{2} \int_0^t \left(\varphi_S^2 + \varphi_v^2\right)\, ds \right).$$

such that

$$dW_{1,t}^{\mathbb{Q}} = dW_{1,t}^{\mathbb{P}} + \varphi_S dt,$$

$$dW_{2,t}^{\mathbb{Q}} = dW_{2,t}^{\mathbb{P}} + \varphi_v dt,$$

are $\mathbb{Q}$ Brownian Motions. Note that $dW^{\mathbb{Q}}_{1,t}dW^{\mathbb{Q}}_{2,t} = \rho^{\mathbb{P}}dt$, so when changing measures, correlation is unchanged $\rho^{\mathbb{Q}} = \rho^{\mathbb{P}}$.

Using this, we can substitute the increments of the standard Brownian motions under the $\mathbb{Q}$-measure and the market prices of risk into (2.6) and (2.7) respectively to reach our risk-neutral processes, firstly for the underlying:

$$\begin{aligned} dS_t &= \mu^{\mathbb{P}} S_t\, dt + \sqrt{v_t} S_t \left(dW^{\mathbb{Q}}_{1,t} - \frac{\mu^{\mathbb{P}} - r}{\sqrt{v_t}}\, dt \right), \\ dS_t &= r S_t\, dt + \sqrt{v_t} S_t\, dW^{\mathbb{Q}}_{1,t}, \end{aligned} \tag{2.10}$$

a similar handling to represent the log price as

$$d \ln S_t = \left(r - \frac{1}{2} \right) dt + \sqrt{v_t} dW^{\mathbb{Q}}_{1,t}, \tag{2.11}$$

and lastly for the variance process

$$\begin{aligned} dv_t &= \kappa^{\mathbb{P}} \left(\theta^{\mathbb{P}} - v_t \right) dt + \sigma^{\mathbb{P}} \sqrt{v_t} dW^{\mathbb{Q}}_{2,t} - \lambda v_t\, dt, \\ dv_t &= \left(\kappa^{\mathbb{P}} \theta^{\mathbb{P}} - (\kappa^{\mathbb{P}} + \lambda) v_t \right) dt + \sigma^{\mathbb{P}} \sqrt{v_t} dW^{\mathbb{Q}}_{2,t}, \\ dv_t &= \kappa^{\mathbb{Q}} \left(\theta^{\mathbb{Q}} - v_t \right) dt + \sigma^{\mathbb{Q}} \sqrt{v_t} dW^{\mathbb{Q}}_{2,t}. \end{aligned} \tag{2.12}$$

Reaching a square-root diffusion process with volatility of volatility also unchanged $\sigma^{\mathbb{P}} = \sigma^{\mathbb{Q}}$, and as in Heston's paper [2].

$$\begin{aligned} \kappa^{\mathbb{Q}} &= \kappa^{\mathbb{P}} + \lambda, \\ \theta^{\mathbb{Q}} &= \frac{\kappa^{\mathbb{P}} \theta^{\mathbb{P}}}{\kappa^{\mathbb{P}} + \lambda}. \end{aligned}$$

A nice observation is that when $\lambda = 0$, the parameters in the risk-neutral process exactly reflect those under the physical measure in (2.6) and (2.7).

In the following three sections, we will follow closely to "The Heston Model and its Extensions in Matlab and C#" [10], Rouah shows one of the most complete derivations of the PDE and finding the two probabilities in the discounted risk-neutral call price, which Heston [2] assumed to be true; let us start with the risk-neutral call price.

2.3 European Call Price

Prior to solving the Heston PDE we want to show that the European call option price can be expressed similarly as in the Black-Scholes framework. The payoff for a European call option is

$$C(S,T) = (S_T - K, 0)^+ \tag{2.13}$$

for a strike price K at maturity T where the underlying is valued as S_T. Now the price at any time $t < T$ is the discounted expected value of the payoff function above, under the risk-neutral measure $\mathbb{Q}$.

$$\begin{aligned} C(K) &= e^{-rT}\mathbb{E}^{\mathbb{Q}}[(S_T - K)^+], \\ &= e^{-rT}\mathbb{E}^{\mathbb{Q}}[(S_T - K)\mathbf{1}_{S_T>K}], \\ &= e^{-rT}\mathbb{E}^{\mathbb{Q}}[S_T\mathbf{1}_{S_T>K}] - Ke^{-rT}\mathbb{E}^{\mathbb{Q}}[\mathbf{1}_{S_T>K}], \\ &= S_tP_1 - Ke^{-rT}P_2. \end{aligned} \tag{2.14}$$

1 is the indicator function, that is only activated to represent 1 if its condition is met or 0 otherwise. As in the Black-Scholes case P_1 and P_2 represent the probability of the call expiring in-the-money, see Rouah [10] on how they get to the last line of (2.14) but to summarise, for the Heston model the probabilities are different to the Black-Scholes definitions as they arise by modelling the variance process as stochastic.

2.4 Heston PDE

Here we present a derivation of the Heston partial differential equation. When doing this in the case of Black-Scholes, there was only one source of randomness as one of the assumptions was made that volatility is constant. There, we would form a portfolio consisting of the underlying stock and a single financial derivative that hedges the stock resulting in a riskless portfolio. For Heston's model, we cannot simply trade the underlying as variance is a stochastic process. We need to introduce an additional derivative to hedge this process. The portfolio will be formed of one option $V = V(S, v, t)$, Δ units of the underlying, and Δ_1 units of the additional option $U = U(S, v, t)$.
We assume the dynamics of the risk-free rate are

$$dB_t = rB_t dt, \tag{2.15}$$

and the value of the portfolio is

$$\Pi = V + \Delta S + \Delta_1 U. \tag{2.16}$$

The change in portfolio value is

$$d\Pi = dV + \Delta dS + \Delta_1 dU \tag{2.17}$$

We can follow the Black-Scholes framework by applying Itô's Lemma to get the U and V processes to find the portfolio, Π, process. Then choosing values for Δ and Δ_1 to make the portfolio riskless, deriving the Heston PDE.

2.4.1 The Hedging Portfolio

Let's start with applying Itô's Lemma to the first derivative $V(S, v, t)$, differentiating with respect to t, S, and v and looking up to only the second-order Taylor series expansion. Consequently, dV follows the process

$$dV = \frac{\partial V}{\partial t}dt + \frac{\partial V}{\partial S}dS + \frac{\partial V}{\partial v}dv + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2}dt + \frac{1}{2}\nu^2\frac{\partial^2 V}{\partial v^2}dt + \sigma\rho vS\frac{\partial^2 V}{\partial S \partial v}dt. \tag{2.18}$$

Here we have made use of properties from Itô's table, being $(dt)^2 = 0$, $dWdt = 0$, and $(dW)^2 = dt$. We also do the same for the other derivative $U(S, v, t)$, giving an identical expression as (2.18) but in terms of U. If we substitute these two expressions into (2.17), the change in portfolio becomes

$$\begin{aligned} d\Pi &= dV + \Delta dS + \varphi dU \\ &= \left[\frac{\partial V}{\partial t} + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2} + \rho\sigma vS\frac{\partial^2 V}{\partial v \partial S} + \frac{1}{2}\sigma^2 v\frac{\partial^2 V}{\partial v^2}\right]dt \\ &\quad + \Delta_1\left[\frac{\partial U}{\partial t} + \frac{1}{2}vS^2\frac{\partial^2 U}{\partial S^2} + \rho\sigma vS\frac{\partial^2 U}{\partial v \partial S} + \frac{1}{2}\sigma^2 v\frac{\partial^2 U}{\partial v^2}\right]dt \\ &\quad + \left[\frac{\partial V}{\partial S} + \Delta_1\frac{\partial U}{\partial S} + \Delta\right]dS \\ &\quad + \left[\frac{\partial V}{\partial v} + \Delta_1\frac{\partial U}{\partial v}\right]dv. \end{aligned} \tag{2.19}$$

To eliminate risk, against movements in the stock and volatility, the two terms dS and dv (2.19) must vanish, so we set

$$\Delta_1 = -\frac{\partial V}{\partial \nu}\Big/\frac{\partial U}{\partial \nu}, \tag{2.20a}$$

$$\Delta = -\Delta_1\frac{\partial U}{\partial S} - \frac{\partial V}{\partial S}. \tag{2.20b}$$

Then substitute the above values back into (2.19) resulting in

$$d\Pi = \left[\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 V}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 V}{\partial\nu^2}\right]dt + \Delta_1\left[\frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 U}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 U}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial\nu^2}\right]dt. \tag{2.21}$$

The no-arbitrage argument has the condition that the portfolio must earn the risk-free rate, r, so the change in portfolio value is $d\Pi = r\Pi dt$ with (2.17) becoming

$$d\Pi = r(V + \Delta S + \Delta_1 U)dt \tag{2.22}$$

Equating (2.21) and (2.22), then substituting (2.20a) and (2.20b), we obtain

$$\frac{\left[\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 V}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 V}{\partial\nu^2}\right] - rV + rS\frac{\partial V}{\partial S}}{\frac{\partial V}{\partial\nu}} = \frac{\left[\frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 U}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 U}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial\nu^2}\right] - rU + rS\frac{\partial U}{\partial S}}{\frac{\partial U}{\partial\nu}} \tag{2.23}$$

We can observe that now the drift terms under the physical measure no longer appear for S_t or v_t, so the value isn't affected.

2.4.2 Option Price PDE

On purpose, in (2.23) we have made the left-hand side a function of only U while the right-hand side is a function of V. Therefore, both sides can equal a function $f(S, v, t)$. Heston [2] proposes the function as

$$f(S, \nu, t) = -\kappa(\theta - \nu) + \lambda(S, \nu, t) \tag{2.24}$$

where λ again is our variance risk premium that we encountered with Girsanov's theorem. We substitute the above into (2.23)

$$-\kappa(\theta - \nu) + \lambda(S, \nu, t) = \frac{\left[\frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 U}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 U}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial\nu^2}\right] - rU + rS\frac{\partial U}{\partial S}}{\frac{\partial U}{\partial\nu}} \tag{2.25}$$

Producing Heston's PDE like so

$$\begin{aligned}\frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 U}{\partial S^2} + \rho\sigma\nu S\frac{\partial^2 U}{\partial S\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial\nu^2} \\ - rU + rS\frac{\partial U}{\partial S} + [\kappa(\theta-\nu) - \lambda(S,\nu,t)]\frac{\partial U}{\partial\nu} = 0.\end{aligned} \tag{2.26}$$

As we are working on the European call price, with strike K and maturity T

$$U(S,v,t) = (S_T - K, 0)^+ \tag{2.27}$$

We have some well known boundary conditions:

- The call has no value when the underlying is zero.

$$U(0,v,t) = 0 \tag{2.28}$$

- As the underlying tends to infinity, the delta of the option approaches 1.

$$\frac{\partial U}{\partial S}(\infty,\nu,t) = 1 \tag{2.29}$$

- As volatility tends to infinity, the option value equals the underlying.

$$U(S,\infty,t) = S \tag{2.30}$$

Furthermore, we will express (2.26) as

$$\frac{\partial U}{\partial t} + \mathcal{A}U - rU = 0 \tag{2.31}$$

with $\mathcal{A}$ as

$$\begin{aligned}\mathcal{A} = rS\frac{\partial}{\partial S} + \frac{1}{2}\nu S^2\frac{\partial^2}{\partial S^2} \\ + [\kappa(\theta-\nu) - \lambda(S,\nu,t)]\frac{\partial}{\partial\nu} + \frac{1}{2}\sigma^2\nu\frac{\partial^2}{\partial\nu^2} + \rho\sigma\nu S\frac{\partial^2}{\partial S\partial\nu}\end{aligned} \tag{2.32}$$

We have presented it as follows just to notice the difference between the Black-Scholes model and Heston's model, where the top line is what you obtain in Black-Scholes' PDE and the bottom is an adjustment to the PDE for stochastic volatility.

Heston's PDE for Log Spot Price

So we have Heston's PDE in (2.26), but here we perform a transformation of the variables to show it in terms of the log price $x = \ln S$. Firstly, we transform by applying the chain rule

$$\frac{\partial U}{\partial S} = \frac{\partial U}{\partial x}\frac{1}{S}, \tag{2.33}$$

$$\frac{\partial^2 U}{\partial v \partial S} = \frac{\partial}{\partial v}\left(\frac{1}{S}\frac{\partial U}{\partial x}\right) = \frac{1}{S}\frac{\partial^2 U}{\partial v \partial x}, \tag{2.34}$$

and using the product rule

$$\frac{\partial^2 U}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{1}{S}\frac{\partial U}{\partial x}\right) = \frac{1}{S^2}\left(\frac{\partial^2 U}{\partial x^2} - \frac{\partial U}{\partial x}\right). \tag{2.35}$$

Plugging these terms into (2.26) we get

$$\frac{\partial U}{\partial t} + \frac{1}{2}v\frac{\partial^2 U}{\partial x^2} + \left(r - \frac{1}{2}v\right)\frac{\partial U}{\partial x} + \rho\sigma v\frac{\partial^2 U}{\partial v \partial x} + \frac{1}{2}\sigma^2 v\frac{\partial^2 U}{\partial v^2} + \{\kappa(\theta - v) - \lambda v\}\frac{\partial U}{\partial v} - rU = 0, \tag{2.36}$$

where all the asset prices S have cancelled out.

2.5 P1 and P2 PDEs

A vanilla call, (2.14), satisfies the Heston PDE in terms of log spot price so we can express the PDE in terms of P_1 and P_2. To do this first represent the last line of (2.14) with $x = \ln S_t$ like so

$$C(K) = e^x P_1 - K e^{-rT} P_2. \tag{2.37}$$

Then we calculate the necessary derivatives which we can then substitute into (2.36)

$$\frac{\partial U}{\partial t} = e^x \frac{\partial P_1}{\partial t} - Ke^{-r(T-t)}\left[rP_2 + \frac{\partial P_2}{\partial t}\right] \tag{2.38}$$

$$\frac{\partial U}{\partial v} = e^x \left[\frac{\partial P_1}{\partial v} - Ke^{-r(T-t)}\frac{\partial P_2}{\partial v}\right] \tag{2.39}$$

$$\frac{\partial^2 U}{\partial v^2} = e^x \left[\frac{\partial^2 P_1}{\partial v^2} - Ke^{-r(T-t)}\frac{\partial^2 P_2}{\partial v^2}\right] \tag{2.40}$$

$$\frac{\partial U}{\partial x} = e^x \left[P_1 + \frac{\partial P_1}{\partial x}\right] - Ke^{-r(T-t)}\frac{\partial P_2}{\partial x} \tag{2.41}$$

$$\frac{\partial^2 U}{\partial x^2} = e^x \left[P_1 + 2\frac{\partial P_1}{\partial x} + \frac{\partial^2 P_1}{\partial x^2}\right] - Ke^{-r(T-t)}\frac{\partial^2 P_2}{\partial x^2} \tag{2.42}$$

$$\frac{\partial^2 U}{\partial x \partial v} = e^x \left[\frac{\partial P_1}{\partial v} + \frac{\partial^2 P_1}{\partial x \partial v}\right] - Ke^{-r(T-t)}\frac{\partial^2 P_2}{\partial x \partial v} \tag{2.43}$$

Now Rouah plugs them into log spot PDE (2.36), rearranges and cancels terms to get two equations for $j = 1, 2$

$$\frac{\partial P_i}{\partial t} + \rho\sigma v \frac{\partial^2 P_i}{\partial v \partial x} + \frac{1}{2} v \frac{\partial^2 P_i}{\partial x^2} + \frac{1}{2}\sigma^2 v \frac{\partial^2 P_i}{\partial v^2} + (r + u_1 v)\frac{\partial P_i}{\partial x} + (a - b_1 v)\frac{\partial P_i}{\partial v} = 0 \tag{2.44}$$

where $u_1 = \frac{1}{2}, u_2 = \frac{-1}{2}, a = \kappa\theta, b_1 = \kappa + \lambda - \rho\sigma$ and $b_2 = \kappa\lambda$.

2.6 Heston's European closed form solution

For brevity we will cut out many steps of [10] and step through important points. The first being that there is only a semi-analytical solution for the European call price, this is due to the fact there aren't direction solutions of the PDEs for P_1 and P_2. Heston's approach was to recover the solutions via use of characteristic functions.

For P_1 and P_2, as long as the characteristic function φ_1 and φ_2 exist, the probabilities can be found using the Gil-Peláez inversion formula [12]

$$P_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \Re\left[\frac{e^{-iu\ln K}\varphi_j(x, v)}{iu}\right] du, \quad \text{for } j = 1, 2. \tag{2.45}$$

where $i = \sqrt{-1}$ is the imaginary number. The characteristic functions, for the log spot price, were then assumed to have the following form

$$\varphi_j(x, v; \phi) = \exp\{C_j(T - t; \phi) + D_j(T - t; \phi)v_t + i\phi x_t\}. \tag{2.46}$$

Getting the required derivatives of the above, we sub them into (2.36), you end up with the following system of ordinary differential equations (ODEs)

$$\frac{\partial C_j(\tau;\phi)}{\partial \tau} = aD_j(\tau;\phi) + r\phi i, \tag{2.47a}$$

$$\frac{\partial D_j(\tau;\phi)}{\partial \tau} = \frac{\sigma^2 D_j^2(\tau;\phi)}{2} - b_j D_j(\tau;\phi) + u_j\phi i - \frac{\phi^2}{2} + \rho\sigma i\phi D_j. \tag{2.47b}$$

where $\tau = T - t$ and the coeffecients defined as before, with initial conditions

$$C_j(0,\phi) = 0, \quad D_j(0,\phi) = 0.$$

(2.47a) is a more straight forward first-order PDE while (2.47b) is a Ricatti equation. The solution to the system is the following

$$C_j(\tau,\phi) = r\phi i\tau + \frac{a}{\sigma^2}\left\{(b_j - \rho\sigma\phi i + d_j)\,\tau - 2\ln\left[\frac{1-g_j e^{d_j\tau}}{1-g_j}\right]\right\}, \tag{2.48}$$

$$D_j(\tau,\phi) = \frac{b_j - \rho\sigma\phi i + d_j}{\sigma^2}\left[\frac{1-e^{d_j\tau}}{1-g_j e^{d_j\tau}}\right], \tag{2.49}$$

with

$$g_j = \frac{b_j - \rho\sigma\phi i + d_j}{b_j - \rho\sigma\phi i - d_j},$$
$$d_j = \sqrt{(\rho\sigma\phi i - b_j)^2 - \sigma^2(2u_j\phi i - \phi^2)},$$

The above combined with the characterstic function is the closed-form solution to the Heston model.

Pricing European Call Options
To actually price options computationally, we break it down to a few steps:

- *Step 1:* Create a function for the characteristic function (2.46) making use of the definitions of C_j and D_j from (2.48) and (2.49) respectively.
- *Step 2:* Create a function for the integrand from both probabilities (2.45)

  $$\Re\left[\frac{e^{-iu\ln K}\varphi_j(x,v)}{iu}\right]$$

  making use of the previously defined characteristic function, then take the real part of the integrand.
- *Step 3:* Use numerical integration techniques to solve both integrand functions, then multiplying by $\frac{1}{\pi}$ and adding $\frac{1}{2}$ gives us P_1 and P_2.
- *Step 4:* Plugging P_1 and P_2 into (2.37) gives you the European call option price.

We notice that there are two times where numerical integration is required (*Step 3*). There are ways to speed up this process by combining the integrals, for example see Emerick's [13] approach. However, this dissertation's focus is on finite difference approximation, we will simply make use of Quantlib's Python library [14] when needed, it is reliable.

3 Finite Difference Approximation

In the previous chapter we saw that there holds a closed-form solution to the Heston model for European call options. In this section we will instead explore a numerical technique to approximate the solution, known as finite difference methods. They can be advantageous to efficiently price an entire surface for many initial values S_0 and v_0, additionally the Greeks are a simple and natural extension. The finite difference method works by approximating the solution of Heston's partial differential equation, so we begin by looking at its characteristics. We then follow the general Method of Lines (MOL) approach. We begin by transforming the continuous region where the PDE is defined, $\Omega \in \mathbb{R}^n$, to a discrete version. This gives us a discrete space made up of grid points. With function values of u corresponding to grid points, we can approximate all continuous functions $u \in C(\Omega)$. Influential to the accuracy of the approximation is the granularity of the grid; a finer grid allows for a more precise representation of the underlying functions and subsequently the solution. In this dissertation non-uniform grids are considered, we will step through the mesh generation process, as well as the finite difference schemes to approximate the derivatives of u. This allows us to construct a finite number of ordinary differential equations (ODEs) which time stepping techniques, utilising boundary conditions and an initial state, can be used to arrive at a final solution of Heston's PDE at maturity of the option.

3.1 Heston PDE as a convection-diffusion-reaction equation

Recall from Chapter 3 $U(S, v, t)$ is the fair price of a European call option and can be represented as a PDE, note we are switching to the notation $u(S, v, t)$ as the capitals will be reserved for vectors and matrices later.

$$\frac{\partial u}{\partial \tau} = \underbrace{\frac{1}{2}S^2\nu\frac{\partial^2 u}{\partial S^2} + \rho\sigma Sv\frac{\partial^2 u}{\partial S \partial v} + \frac{1}{2}\sigma^2 v\frac{\partial^2 u}{\partial v^2}}_{(1)} + \underbrace{rS\frac{\partial u}{\partial S} + \kappa(\theta - v)\frac{\partial u}{\partial v}}_{(2)} - \underbrace{ru}_{(3)} \tag{3.1}$$

for $S > 0, v > 0, 0 < t <= T$

We have transformed this PDE to have $\tau = T - t$ represent time to maturity, the sign change can be thought of intuitively as $\frac{\partial u}{\partial \tau}$ is the opposite of $\frac{\partial u}{\partial t}$ or using the following

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial \tau}\frac{\partial \tau}{\partial t} = -\frac{\partial u}{\partial \tau}, \tag{3.2}$$

on (2.26). This transformation is more suitable to work with in terms of finite difference methods, as the payoff function will become an initial condition.

$$u(S, v, 0) = (S - K, 0)^+ \tag{3.3}$$

There is a point of discontinuity at the strike price for which we will later show a smoothing method after defining our mesh.

Observations: This is a time dependent PDE of the convection-diffusion-reaction type.

- (1): The diffusion terms are second order derivatives capturing the random fluctuations in S and v and their relationship governed by ρ.
- (2): The convection terms, in a physical sense represents the flow within the system, capturing the drift rates of both processes.
- (3): The reaction term plays a smaller role, it essentially discounts with respect to the risk-free rate.

Additionally, if σ is small, Haentjens [15] mentions the PDE becomes convection dominated in the v direction, due to the terms involving σ. Let's just note that numerically diffusion is easier to solve than convection dominated problems due to their smoothness while we also later explore oscillations triggered by larger fluctuations in convection dominated situations.

Note: For the rest of this dissertation we will actually use t to represent time to maturity instead of τ, just to align with the common literature.

3.2 Method of Lines approach

The MOL [16] is a general approach that will allow us to view and subsequently solve the multi-dimensional Heston PDE as a system of ODEs. There are two main steps we will follow, firstly discretizing the PDE into its spatial variables keeping time continuous, and secondly applying some time-stepping techniques to the system of equations where we choose Alternating Direction Implicit (ADI) splitting schemes [1].

The two successive steps:

1. Discretization in space:

   Finite Difference schemes on non-uniform grids

2. Discretization in time:

   Alternating Direction Implicit (ADI) schemes

[1] Hout [17] notes that ADI schemes, as opposed to the common Crank-Nicolson scheme, have become necessary as a result of the large semi-discrete systems generated by multi-dimensional PDEs such as from Heston.

3.3 Discretization in space

A preliminary step is to choose a spatial domain on which we perform a finite difference approximation to our function $u(S, v, t)$ from our two spatial variables S and v. Initially, this domain is unbounded in the positive direction as S and v cannot take negative values in practice, $(0, \infty) \times (0, \infty)$. Thus we truncate it $[S_{\min}, S_{\max}] \times [V_{\min}, V_{\max}]$, as a result of computational limits of course. We will take a heuristic approach for the values of S_{max} and V_{max} and additionally we present boundary conditions

$$\frac{\partial u}{\partial S}(S_{\max}, v, t) = 1, \tag{3.4}$$

$$u(S, V_{\max}, t) = S \tag{3.5}$$

(3.4) being the Neumann condition and (3.5) the Dirichlet condition, which we will go into more detail later and present a sketch of the bounded domain.

3.3.1 Mesh Generation

With the preliminary step out of the way, there is a choice between uniform and non-uniform grids. Our aim is to increase approximation accuracy, where a higher number of grid points around a particular area leads us closer to the exact solution. With this, we choose non-uniform grids, which are slightly trickier to handle. The financial motivation here is that we can have a finer grid in our region of interest, commonly defined around the strike price. This allows for more accurate price approximations with fewer grid points all together, also alleviating the problem of the discontinuity from a non-smooth initial condition in (3.3). Furthermore we want more grid points at $v = 0$ as the PDE becomes convection dominated allowing to capture sharp gradients (or discontinuities)[2]. Lastly, we can save on computation time by having a sparser grid at S_{max} and V_{max} which are chosen to be quite large.

The mesh generating function we employ is based on the hyperbolic sine function, which has seen extensive use in the literature. We choose the approach of Haentjen and Hout[18], so this section will closely follow their work. It is worthy to note that this process of discretisation has roots in many popular textbooks, such as [7] and [8]. There are also slight variations with different mappings, for example considered by Kluge [19]. We stick with Haentjen and Hout's approach as it has been extensively referenced and their results reproduced. Howbeit, the fundamental idea between approaches remains the same; essentially one starts with an artificial uniform mesh and maps it to a non-uniform mesh with some desired requirements.

[2]Recall convection dominated problems in a physical sense represent a more rapid flow rather than slower diffusion.

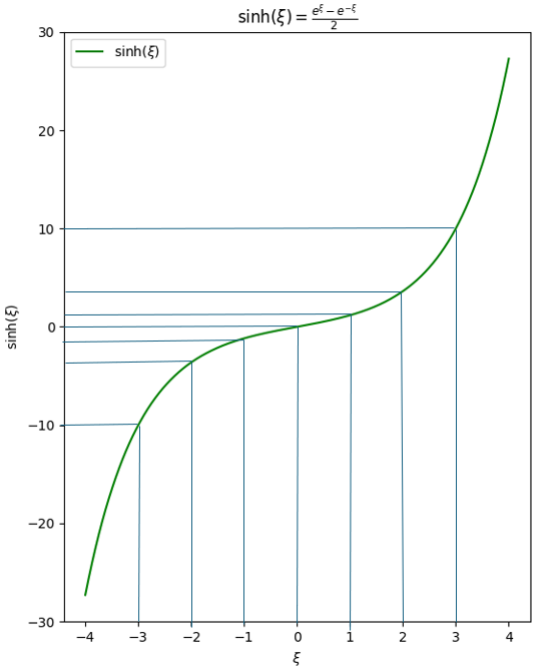


Figure 1: Mesh generating function

As can be seen in Figure 1, the uniform steps in the x-axis direction are transformed to non-uniform steps in the y-axis.

Mesh in S-direction
The non-uniform mesh we employ will not exactly have an increased density at the strike, but rather a uniform mesh with relatively many points in an interval $[S_{\text{left}}, S_{\text{right}}]$ around the strike price, and outside of this a non-uniform mesh which will gradually increase in width to the maximum values. As mentioned earlier, this adjustment was introduced by [18] so we follow on from them:

Let the number of grid points in S be the integer $m_1 \geq 2$, constant $c > 0$ relating to the number of points in the interval and $0 < S_{\text{left}} < K < S_{\text{right}} < S_{\max}$.

Let our artificial equidistant points $\xi_{\min} = \xi_0 < \xi_1 < \cdots < \xi_{m_1} = \xi_{\max}$ be given by

$$\xi_{\min} = \sinh^{-1}\left(\frac{-S_{\text{left}}}{c}\right), \tag{3.6}$$

$$\xi_{\text{int}} = \frac{S_{\text{right}} - S_{\text{left}}}{c}, \tag{3.7}$$

$$\xi_{\max} = \xi_{\text{int}} + \sinh^{-1}\left(\frac{S_{\max} - S_{\text{right}}}{c}\right). \tag{3.8}$$

Note that $\xi_{\min} < 0 < \xi_{\text{int}} < \xi_{\max}$. Then the non-uniform mesh

$$0 = s_0 < s_1 < \cdots < s_{m_1} = S_{\max},$$

is defined through the transformation

$$S_i = \varphi(\xi_i), \quad (0 \leq i \leq m_1), \tag{3.9}$$

where

$$\varphi(\xi) = \begin{cases} S_{\text{left}} + c \cdot \sinh(\xi), & (\xi_{\min} \leq \xi \leq 0), \\ S_{\text{left}} + c \cdot \xi, & (0 < \xi < \xi_{\text{int}}), \\ S_{\text{right}} + c \cdot \sinh(\xi - \xi_{\text{int}}), & (\xi_{\text{int}} \leq \xi \leq \xi_{\max}). \end{cases} \tag{3.10}$$

The second case gives us the uniform mesh in our interval $[S_{left}, S_{right}]$ as it is just a linear function in ξ and the first and third case give a non-uniform mesh outside. From [17] we make the following heuristic choices

$$S_{\text{left}} = \max\left(\frac{1}{2}, e^{-rT}\right) K, \tag{3.11}$$

$$S_{\text{right}} = \min\left(\frac{3}{2}, e^{+rT}\right) K \tag{3.12}$$

where $r = \frac{1}{10}$.

Mesh in v-direction
In the v-direction it is slightly simpler as we don't include any specific interval. Let the number of grid points in v be an integer $m_2 \geq 2$ and constant $d > 0$ relating to the number of grid points that are close to $v = 0$.

Let our artificial equidistant points $\psi_0 < \psi_1 < \ldots < \psi_{m_2}$ be given by

$$\psi_j = j \cdot \Delta\psi \quad (0 \leq j \leq m_2) \tag{3.13}$$

with

$$\Delta\psi = \frac{1}{m_2} \sinh^{-1}\left(\frac{V_{\max}}{d}\right). \tag{3.14}$$

Then the non-uniform mesh

$$0 = v_0 < v_1 < \ldots < v_{m_2} = V_{\max}$$

is defined by our hyperbolic mapping

$$v_j = d \cdot \sinh(\psi_j) \quad (0 \leq j \leq m_2). \tag{3.15}$$

Smoothing European call payoff function
Relevant to our implementation, we quickly present a method, presented by Tavella and Randall [8], to smooth the area around the strike price as there lies a point of discontinuity.

$$\frac{1}{h} \int_{S_{i-1/2}}^{S_{i+1/2}} (0, S - K)^+ \, dS,$$

where

$$S_{i-1/2} = \frac{1}{2}(S_{i-1} + S_i), \quad S_{i+1/2} = \frac{1}{2}(S_i + S_{i+1}), \quad h = S_{i+1/2} - S_{i-1/2}.$$

We find the point on our grid nearest to the strike and take an average over the presented interval above to evaluate the payoff function of a European call option.

Mesh Properties
Let's go through the properties and choices of parameters in our implementation, considering non-uniform mesh widths in both directions $\Delta S_i = S_i - S_{i-1}$ and $\Delta v_j = v_j - v_{j-1}$ and as previously mentioned parameters c and d controling the fraction of points near $S = K$ and $v = 0$ respectively.

$$\Delta s_i = c \cdot \Delta\xi \quad (\text{if } s_i \approx K) \quad \text{and} \quad \Delta v_j \approx d \cdot \Delta\psi \quad (\text{if } v_j \approx 0).$$

Here we see precise relations for S in the interval, where the widths are fixed of size c multiplied by the artificial mesh width. Similarly, in the v-direction just noting it is only approximately equal to a factor of the artificial mesh width due to the lack of a defined uniform interval. Now based on previous literature: [20], [21] and [17] we choose

$$c = \frac{K}{10}, \quad d = \frac{V_{\max}}{500}, \quad S_{\max} = 30K, \quad V_{\max} = 15.$$

One thing to note here is that S_{max} amd V_{max} are very large. This choice is due to the fact our approximation is surrounded by boundaries that are artificial, later in Section 3.3.3 we consider values for our approximation function when our spatial variables S or v tend to infinity but we of course can't have an infinite grid. It is impossible to know the exact option value at those boundaries, so this error can propagate into our region of interest if our spatial variables aren't sufficiently large.

Another property, as shown by [17], is that both meshes are smooth. For example in the S-direction, the mesh widths are directly proportional to the artificial mesh widths, for real constants C_0, C_1 and C_2

$$C_0\Delta\xi \leq \Delta S_i \leq C_1\Delta\xi$$

and the changes in the mesh widths are gradual, shown below, are both satisfied.

$$|\Delta s_{i+1} - \Delta s_i| \leq C_2(\Delta\xi)^2.$$

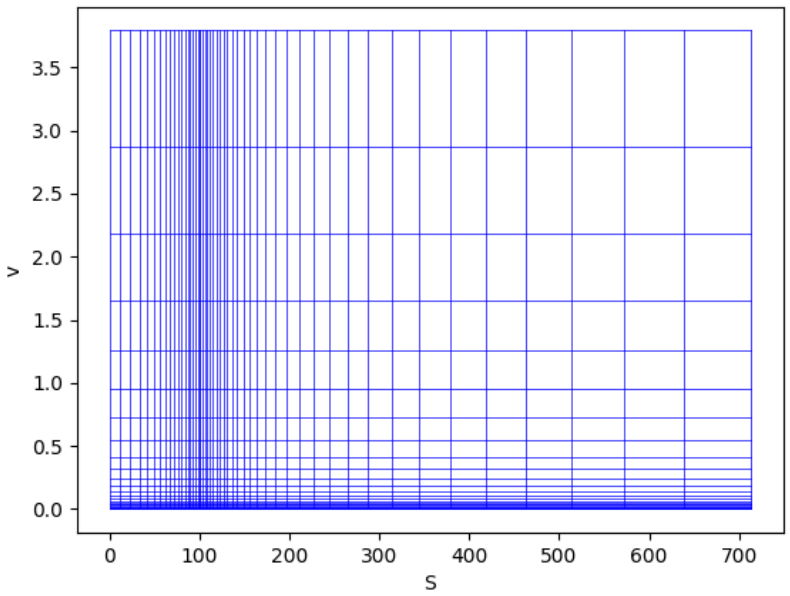


Figure 2: Non-uniform mesh with $S_{max} = 720$, $V_{max} = 4$ and $K = 100$

Now all together in Figure 2 is an example mesh to illustrate our requirements being met. You can see a higher density of points at $v = 0$, and a uniform mesh for S around the strike price $K = 100$ and non-uniform otherwise. When experimenting, our implementation of course uses larger S_{max} and V_{max}.

3.3.2 Non-uniform difference schemes

As mentioned previously, we only implement non-uniform grids requiring us to use appropriate finite difference schemes for approximating derivatives, used to discretise the Heston PDE on our grid. So, we follow the previously mentioned tailored approach [18] adapted from Hundsdorfer and Verwer's textbook [16].

Let $f : \mathbb{R} \to \mathbb{R}$ be any given function, let $x_0 < x_1 < x_2 < \cdots < x_m$ be any given mesh points and $\Delta x_i = x_i - x_{i-1}$. To approximate the first derivative $f'(x_i)$, we consider three finite difference schemes:

$$\text{(Backward)} \quad f'(x_i) \approx \alpha_{i,-2} f(x_{i-2}) + \alpha_{i,-1} f(x_{i-1}) + \alpha_{i,0} f(x_i), \tag{3.16a}$$

$$\text{(Central)} \quad f'(x_i) \approx \beta_{i,-1} f(x_{i-1}) + \beta_{i,0} f(x_i) + \beta_{i,1} f(x_{i+1}), \tag{3.16b}$$

$$\text{(Forward)} \quad f'(x_i) \approx \gamma_{i,0} f(x_i) + \gamma_{i,1} f(x_{i+1}) + \gamma_{i,2} f(x_{i+2}), \tag{3.16c}$$

where

$$\alpha_{i,-2} = \frac{\Delta x_i}{\Delta x_{i-1}(\Delta x_{i-1} + \Delta x_i)}, \quad \alpha_{i,-1} = \frac{-\Delta x_{i-1} - \Delta x_i}{\Delta x_{i-1}\Delta x_i}, \quad \alpha_{i,0} = \frac{\Delta x_{i-1} + 2\Delta x_i}{\Delta x_i(\Delta x_{i-1} + \Delta x_i)},$$

$$\beta_{i,-1} = \frac{-\Delta x_{i+1}}{\Delta x_i(\Delta x_i + \Delta x_{i+1})}, \quad \beta_{i,0} = \frac{-\Delta x_{i+1} + \Delta x_i}{\Delta x_i \Delta x_{i+1}}, \quad \beta_{i,1} = \frac{\Delta x_i}{\Delta x_{i+1}(\Delta x_i + \Delta x_{i+1})},$$

$$\gamma_{i,0} = \frac{-2\Delta x_{i+1} - \Delta x_{i+2}}{\Delta x_{i+1}(\Delta x_{i+1} + \Delta x_{i+2})}, \quad \gamma_{i,1} = \frac{\Delta x_{i+1} + \Delta x_{i+2}}{\Delta x_{i+1}\Delta x_{i+2}}, \quad \gamma_{i,2} = \frac{\Delta x_{i+1}}{\Delta x_{i+2}(\Delta x_{i+1} + \Delta x_{i+2})}.$$

The backward scheme employs three points, two of which are on the left, x_{i-2} and x_{i-1}, where we are approximating $f'(x_i)$. The forward scheme is the reverse utilising two points to the right x_{i+1}, x_{i+2}, while the central scheme uses a point on either side x_{i-1}, x_{i+1}, all of which of course use x_i as well. We will show the necessity of having the backward and forward, so-called one-sided schemes, later on. Furthermore, it is well-known now, as mentioned also by Kluge [19], that convection dominated parabolic PDEs with "central difference approximation exhibit oscillating behaviour" so the central scheme isn't always applicable.

Uniform grids as a special case

Let us shortly digress to understand where these schemes come from. We will only take as an example the central approximation scheme, see how to derive it from Taylor expansions and notice the relationship between uniform and non-uniform grids, all other schemes above can be derived similarly.

For more clarity we will show the following with the spatial variable x defined for its purpose as S or v. As before define m_1 and m_2 , the distances in the S and v direction respectively and ΔS_i and Δv_j as

$$\Delta S_i = S_i - S_{i-1} \quad \text{and} \quad \Delta v_j = v_j - v_{j-1}.$$

In the uniform case the points are equally distributed in both directions with mesh widths as

$$\Delta s = \frac{S_{\max}}{m_1} \quad \text{and} \quad \Delta v = \frac{V_{\max}}{m_2}$$

Giving a spatial domain

$$\mathcal{G} = \{(S_i, v_j) : 0 \leq i \leq m_1, 0 \leq j \leq m_2\}.$$

At a grid point (S_i, v_j) denote its value as $u_{i,j}(t)$, again for clarity we note this is our general function f from before. Now to approximate the value of our function u at, say a grid point (S_{i+1}, v_j), in terms of the function's value and its derivatives at a nearby point, namely (S_i, v_j), we apply a Taylor expansion

$$u_{i+1,j} = u_{i,j} + \Delta S_{i+1} \left(\frac{\partial u}{\partial S}\right)_{i,j} + \frac{\Delta S_{i+1}^2}{2!} \left(\frac{\partial^2 u}{\partial S^2}\right)_{i,j} + \frac{\Delta S_{i+1}^3}{3!} \left(\frac{\partial^3 u}{\partial S^3}\right)_{i,j} + \cdots \tag{3.17}$$

Due to computational limits, we have to truncate the Taylor series via 'big O' notation, we will do this up to third order accuracy. This allows us to represent (3.17) as

$$u_{i+1,j} = u_{i,j} + \Delta S_{i+1} \left(\frac{\partial u}{\partial S}\right)_{i,j} + \frac{\Delta S_{i+1}^2}{2!} \left(\frac{\partial^2 u}{\partial S^2}\right)_{i,j} + \mathcal{O}(\Delta S_{i+1}^3), \tag{3.18}$$

due to the fact that the following holds in practice

$$\left| \frac{\Delta S_{i+1}^3}{3!} \left(\frac{\partial^3 u}{\partial S^3}\right)_{i,j} + \frac{\Delta S_{i+1}^4}{4!} \left(\frac{\partial^4 u}{\partial S^4}\right)_{i,j} + \cdots \right| \leq M |\Delta S_{i+1}^3| \text{ when } \Delta S_{i+1} \to 0. \tag{3.19}$$

where M is some real constant.

Central Scheme
Now focusing on the central difference scheme, it gets its name as it approximates the derivative at the centre point (S_i, v_j) using its neighbors (S_{i+1}, v_j) and (S_{i-1}, v_j). As shown by Kluge (...) a common approach is to give each of these points a factor β so our central approximation looks as follows

$$\left(\frac{\partial u}{\partial S}\right)_{i,j} \approx \beta_{-1} u_{i-1,j} + \beta_0 u_{i,j} + \beta_{+1} u_{i+1,j} \tag{3.20}$$

Using Taylor expansion on the relevant terms gives

$$\begin{aligned}
\beta_{+1}u_{i+1,j} &= \beta_{+1}\left(u_{i,j} + \Delta S_{i+1}\left(\frac{\partial u}{\partial S}\right)_{i,j} + \frac{\Delta S_{i+1}^2}{2}\left(\frac{\partial^2 u}{\partial S^2}\right)_{i,j} + O(\Delta S_{i+1}^3)\right), \\
\beta_0 u_{i,j} &= \beta_0 u_{i,j}, \\
\beta_{-1}u_{i-1,j} &= \beta_{-1}\left(u_{i,j} - \Delta S_i\left(\frac{\partial u}{\partial S}\right)_{i,j} + \frac{\Delta S_i^2}{2}\left(\frac{\partial^2 u}{\partial S^2}\right)_{i,j} + O(\Delta S_i^3)\right).
\end{aligned} \tag{3.21}$$

Put these terms into our sum from (3.20) and rewrite as follows

$$\begin{aligned}
&\left(\beta_{+1} + \beta_0 + \beta_{-1}\right) u_{i,j} + \left(\beta_{+1}\Delta S_{i+1} - \beta_{-1}\Delta S_i\right)\left(\frac{\partial u}{\partial S}\right)_{i,j} \\
&\qquad + \left(\beta_{+1}\frac{\Delta S_{i+1}^2}{2} + \beta_{-1}\frac{\Delta S_i^2}{2}\right)\left(\frac{\partial^2 u}{\partial S^2}\right)_{i,j} + O(\Delta S^3)
\end{aligned} \tag{3.22}$$

In order to get a second-order accurate central scheme, we want the leading-order and second-order term to cancel, having only the first-order term with a factor of 1. Hence, we set the coefficients as

$$\beta_{+1} + \beta_0 + \beta_{-1} = 0, \tag{3.23a}$$
$$\beta_{+1}\Delta S_{i+1} + \beta_{-1}\Delta S_i = 1, \tag{3.23b}$$
$$\beta_{+1}\frac{\Delta S_{i+1}^2}{2} + \beta_{-1}\frac{\Delta S_i^2}{2} = 0. \tag{3.23c}$$

Solving this system of equations leads to our familiar non-uniform coefficients from (3.16b), noting that we have just dropped one of the subscripts on β below.

$$\begin{aligned}
\beta_{+1} &= \frac{\Delta S_i}{\Delta S_{i+1}(\Delta S_i + \Delta S_{i+1})}, \\
\beta_0 &= \frac{-\Delta S_i + \Delta S_{i+1}}{\Delta S_i \Delta S_{i+1}}, \\
\beta_{-1} &= \frac{-\Delta S_{i+1}}{\Delta S_i(\Delta S_i + \Delta S_{i+1})}.
\end{aligned} \tag{3.24}$$

Now if we consider the uniform case, the mesh widths ΔS_i are constant and uniform allowing us to drop the i subscript completely

$$\begin{aligned}
\beta_{-1} &= \frac{-\Delta S_{i+1}}{\Delta S_i(\Delta S_i + \Delta S_{i+1})} = \frac{-\Delta S}{\Delta S(\Delta S + \Delta S)} = \frac{-1}{2\Delta S} \\
\beta_0 &= \frac{\Delta S_{i+1} + \Delta S_i}{\Delta S_i \Delta S_{i+1}} = \frac{\Delta S - \Delta S}{\Delta S \Delta S} = 0 \\
\beta_{+1} &= \frac{\Delta S_i}{\Delta S_{i+1}(\Delta S_i + \Delta S_{i+1})} = \frac{\Delta S}{\Delta S(\Delta S + \Delta S)} = \frac{1}{2\Delta S}
\end{aligned} \tag{3.25}$$

If we then substitute these into our central difference scheme for $f'(x_i)$ or using this notation $\left(\frac{\partial u}{\partial S}\right)_{i,j}$ we get

$$\begin{aligned}
\left(\frac{\partial u}{\partial S}\right)_{i,j} &\approx \beta_{-1} u_{i-1,j} + \beta_0 u_{i,j} + \beta_{+1} u_{i+1,j} \\
&= \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta S}
\end{aligned} \tag{3.26}$$

This is a very common approximation, seen for example in Wilmott's popular textbook [22]. The same can be applied in the v direction, and a similar approach can be followed for the forward and backward schemes too.

Back to our non-uniform grids, from [18], we now meet the second derivative, for the diffusion terms with respect to one spatial direction, which is approximated as

$$f''(x_i) \approx \delta_{i,-1} f(x_{i-1}) + \delta_{i,0} f(x_i) + \delta_{i,1} f(x_{i+1}), \tag{3.27}$$

where

$$\delta_{i,-1} = \frac{2}{\Delta x_i(\Delta x_i + \Delta x_{i+1})}, \quad \delta_{i,0} = \frac{-2}{\Delta x_i \Delta x_{i+1}}, \quad \delta_{i,1} = \frac{2}{\Delta x_{i+1}(\Delta x_i + \Delta x_{i+1})}. \tag{3.28}$$

Now in the Heston PDE a mixed derivative is present, we will handle it like follows. Assume a function, $f : \mathbb{R}^2 \to \mathbb{R}$, of two spatial variables x and y. With the x-direction mesh points already described, the same goes for y, let $y_0 < y_1 < y_2 < \cdots < y_m$ be any given mesh points and $\Delta y_i = y_i - y_{i-1}$. Analogous to the x-direction, define the coefficient $\hat{\beta}_{j,l}$ but now swapping Δx with Δy

$$\hat{\beta}_{j,-1} = \frac{-\Delta y_{j+1}}{\Delta y_j(\Delta y_j + \Delta y_{j+1})}, \quad \hat{\beta}_{j,0} = \frac{\Delta y_{j+1} - \Delta y_j}{\Delta y_j \Delta y_{j+1}}, \quad \hat{\beta}_{j,1} = \frac{\Delta y_j}{\Delta y_{j+1}(\Delta y_j + \Delta y_{j+1})}. \tag{3.29}$$

The finite difference formula for the discretization of the mixed derivative is

$$\begin{aligned}\frac{\partial^2 f}{\partial x \partial y}(x_i, y_j) &\approx \sum_{k=-1}^{1}\sum_{l=-1}^{1} \beta_{i,k}\hat{\beta}_{j,l} f(x_{i+k}, y_{j+l}) \\ &= \beta_{i,0}\hat{\beta}_{j,0} f(x_i, y_j) + \beta_{i,-1}\hat{\beta}_{j,0} f(x_{i-1}, y_j) + \beta_{i,+1}\hat{\beta}_{j,0} f(x_{i+1}, y_j) \\ &\quad + \beta_{i,0}\hat{\beta}_{j,-1} f(x_i, y_{j-1}) + \beta_{i,0}\hat{\beta}_{j,+1} f(x_i, y_{j+1}) + \beta_{i,-1}\hat{\beta}_{j,-1} f(x_{i-1}, y_{j-1}) \\ &\quad + \beta_{i,-1}\hat{\beta}_{j,+1} f(x_{i-1}, y_{j+1}) + \beta_{i,+1}\hat{\beta}_{j,-1} f(x_{i+1}, y_{j-1}) + \beta_{i,+1}\hat{\beta}_{j,+1} f(x_{i+1}, y_{j+1}),\end{aligned} \tag{3.30}$$

This is a central scheme made up of eight points surrounding the approximated point (x_i, y_j) as seen in Figure 3. There are many choices to use for the mixed derivative, each of which use different stencils, where some useful ones have been collated by [23].

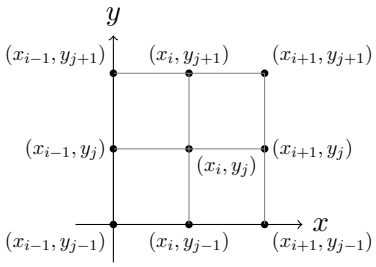


Figure 3: 9 Point stencil of the mixed derivative approximation at (x_i, y_j).

3.3.3 Implementation of boundary conditions

The finite difference formulas presented are for a generic situation over an arbitrary function f, how will we use them in the finite difference approximation for the Heston PDE? To start the boundaries on our domain will be defined followed by which finite difference scheme is applicable at each area.

We briefly met a few boundary conditions in our Background Theories Chapter, now we will build up and introduce ones which are relevant to the finite difference approximation for a European call option. There will be two different kinds of boundaries, being Dirichlet and Neumann boundary conditions as seen on Figure 4. The Dirichlet condition will specify a value of our function on the boundary of the domain, while the Neumann condition specifies a value the derivative must take on the boundary.

- $S = 0$ boundary:

$$u(0, v, t) = 0 \tag{3.31}$$

  At this lower boundary, the option will of course be worthless as the asset price is zero. This known function value leads to a Dirichlet boundary at ①.

- $v = V_{max}$ boundary:

$$u(S_i, V_{max}, t) = S_i \tag{3.32}$$

  The value of a European call increases with volatility, but is bounded by the known asset price value, leading to a Dirichlet boundary at ②.

Now taking into account the Dirichlet boundaries, we can omit the row for $v = V_{max}$ and column for $S = S_{max}$ in our non-uniform computational grid due to the fact that these values don't require approximation, while all other points will.

$$\mathcal{G} = \{(S_i, v_j) : 1 \leq i \leq m_1, 0 \leq j \leq m_2 - 1\}.$$

which looks like

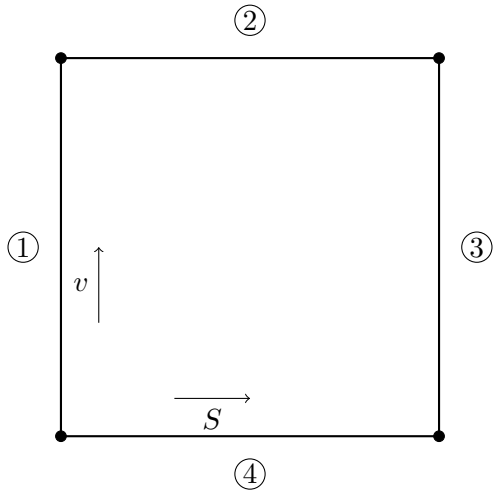


Figure 4: Illustration of the computational grid

- $S = S_{max}$ boundary

$$\frac{\partial u}{\partial S}(S_{\max}, v_j, t) = 1 \tag{3.33}$$

As presented by Heston [10]. The strike price becomes insignificant for large values of S, where the option value tends to the underlying resulting in a delta tending to one. A value for the derivative is being assumed here so ③ is therefore a Neumann boundary condition.

- $v = 0$ boundary:

$$\frac{\partial u}{\partial t}(S, 0, t) = rS\frac{\partial u}{\partial S} + \kappa\theta\frac{\partial u}{\partial v} - rU. \tag{3.34}$$

This boundary is more problematic to handle. Here the diffusion terms disappear from their factor of v, a degenerate feature, resulting in a convection dominated PDE[3]. We wont be able to exactly establish this as a boundary condition due to the presence of $\frac{\partial U}{\partial t}$, so we take Kluge's [19] recommendation to discretise the PDE with one sided schemes, where we choose the forward scheme at ④.

Slightly different variations of boundaries are present in literature, see for example the following textbook by Hirsa [25], however the ones presented are known to work well.

Implementing finite difference schemes

If we are not at the Neumann boundary ③, i.e $i < m_1$, we use the central scheme (3.16b) everywhere for $\frac{\partial u}{\partial S}$. Also, for $\frac{\partial^2 u}{\partial S^2}$, $\frac{\partial^2 u}{\partial v^2}$, $\frac{\partial^2 u}{\partial S \partial v}$ the central scheme (3.27) is used for the two second order derivatives and (3.30) for the mixed derivative.
To approximate $\frac{\partial u}{\partial v}$, as mentioned previously we use the forward scheme (3.16c) at the trickier boundary when $v = 0$. Furthermore we make the choice of using the central scheme (3.16b) only for $0 < v < 1$ and the one-sided backward scheme (3.16a) for $v \geq 1$; these choices come from Kluge and Hout's remarks on convection dominated problems we mentioned earlier. Central schemes can lead to fabricated oscillations when there is predominantly flow at a boundary, forward and backward schemes combat this.

Now instead considering we are at the Neumann boundary ③, $i = m_1$, we need to be careful to not encounter problems by choosing points outside of the grid, as previously at the Dirichlet boundary ② we perform central and backward schemes neither of which protrude outside in the v direction. Starting simply, $\frac{\partial u}{\partial S}$ is given by (3.31) so no approximation is required here. For $\frac{\partial u}{\partial v}$ we treat it the same as the $i < m_1$ case, not needing to

[3] In literature prior, it was assumed that the fair value of the option is still satisfied here but later Ekström and Tysk [24] provided a rigorous proof.

worry about points to the right-hand side as we are in the v direction, the same goes for $\frac{\partial^2 u}{\partial v^2}$. Now for $\frac{\partial^2 u}{\partial S \partial v}$, if we take the derivative, $\frac{\partial}{,}\partial v$ of (3.31) we get zero always. However, to handle $\frac{\partial^2 u}{\partial S^2}$ we need to extrapolate as shown below for the uniform case, see Appendix A of [26] for the non-uniform case.

Consider

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}. \tag{3.35}$$

At the point x we would like to approximate the second derivative, but the problem lies in the fact $x + h$ is outside of our domain. If $f'(x)$ is given, in our case as the Neumann boundary, we can approximate $f(x + h)$ by $f(x - h) + 2hf'(x)$.

3.4 Initial Value Problem

The discretisation above shows how the spatial derivatives can be expressed as discrete function values. Further, Hundsdorfer and Verwer [16] point out using an initial condition with a convection-diffusion-reaction type equation, in our case (3.3) with the Heston PDE (3.1), reduces to a system of ODEs

$$U'(t) = AU(t) + g(t) \quad (0 \leq t \leq T), \quad U(0) = U_0. \tag{3.36}$$

Hundsdorfer and Verwer considered only a one-dimensional equation however it is possible to extend to two-dimensions with a system of $m = m_1 m_2$ ODEs. $U(t)$ will be our solution vector where each element forms approximations to the option price at grid point (S, v) and the derivative with respect to time denoted as $U'(t)$. For the first time step, U_0 is a m-vector obtained by evaluating the European call's payoff function at all grid points. A is a known $m \times m$ matrix obtained from the coefficients of the Heston PDE and a discretisation of the derivatives, $g(t)$ is a known m-vector stemming from the boundary conditions, which we will later see formulas for.

The required lexicographic ordering of U is as follows

$$U = (u_{0,0}, \cdots, u_{0,m_2}, \cdots, \cdots, u_{i,0}, \cdots, u_{i,m_2}, \cdots, \cdots, u_{m_1,0}, \cdots, u_{m_1,m_2})^T. \tag{3.37}$$

The first m_2 elements contain the approximations for S_0 with all points $v_0, \cdots, v_{m_2}$, then the same again for S_1 with all the point $v_0, \cdots, v_{m_2}$ and so on, resulting in a one dimensional vector U of size m.

3.5 A prototype convection-diffusion equation

Before we consider the Heston PDE, let's first see how to convert a prototype equation into a semi-discrete system which will lead us to the required set up in (3.36), it serves as a nice introduction. Consider the convection-diffusion equation, in two dimensions for two spatial variables x and y

$$\frac{\partial u}{\partial t} = c_1 \frac{\partial u}{\partial x} + c_2 \frac{\partial u}{\partial y} + d_{11} \frac{\partial^2 u}{\partial x^2} + 2d_{12} \frac{\partial^2 u}{\partial x \partial y} + d_{22} \frac{\partial^2 u}{\partial y^2} \tag{3.38}$$

Defined on the unit square $(x, y) \in \Omega = (0, 1) \times (0, 1)$, $t > 0$, and $c_1, c_2 \in \mathbb{R}$ are convection constants, $d_{11}, d_{12}, d_{22} \in \mathbb{R}$ are diffusion constants, all given.

Assume initial condition

$$u(x, y, 0) = u_0(x, y) \quad \text{for } (x, y) \in \Omega,$$

and a Dirichlet boundary condition on every side of the grid shown in Figure 5

$$u(x, y, t) = \gamma(x, y, t) \quad \text{for } (x, y) \in \partial\Omega,\ t \geq 0,$$

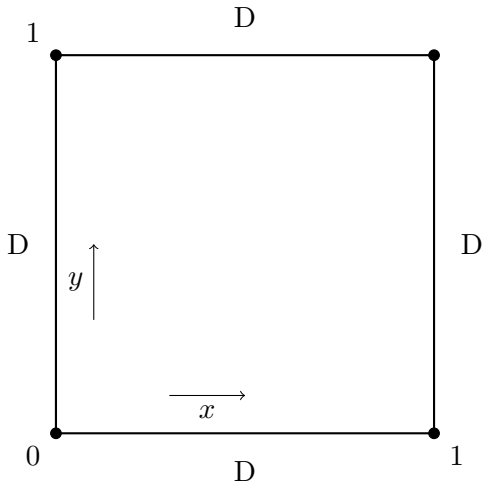


Figure 5: Illustration of the prototype computational grid

3.5.1 Finite difference discretisation of the prototype problem

For convenience, using a uniform Cartesian grid, let $m_1, m_2 \geq 2$, we define the constant mesh widths as

$$h_1 = \Delta x = \frac{1}{m_1 + 1}, \quad h_2 = \Delta y = \frac{1}{m_2 + 1}. \tag{3.39}$$

In view of Dirichlet boundary for convenience we use $m_1 + 1$ and $m_2 + 1$ as later we will see it gives us a system of $m_1 \times m_2$ equations.

Now lets discretise our prototype equation by applying the central finite difference discretisation giving us a semi-discrete system. We have only covered the central scheme in the uniform case, (3.26), but the other schemes used can be found in Wilmott's book [22]. Nevertheless, as an example we will just show how to approximate the term with coefficient c_1 in (3.38) by applying (3.26) to get

$$\left(c_1 \frac{\partial u}{\partial x}\right)_{i,j} \approx \frac{c_1}{2\Delta x}(U_{i+1,j}(t) - U_{i-1,j}(t)) \tag{3.40}$$

Continuing with the same technique for the rest of the terms we get the following:

$$\begin{aligned} U'_{i,j}(t) = &\frac{c_1}{2\Delta x}\left(U_{i+1,j}(t) - U_{i-1,j}(t)\right) + \frac{d_{11}}{(\Delta x)^2}\left(U_{i+1,j}(t) - 2U_{ij}(t) + U_{i-1,j}(t)\right) \\ &+ \frac{c_2}{2\Delta y}\left(U_{i,j+1}(t) - U_{i,j-1}(t)\right) + \frac{d_{22}}{(\Delta y)^2}\left(U_{i,j+1}(t) - 2U_{ij}(t) + U_{i,j-1}(t)\right) \\ &+ \frac{d_{12}}{2\Delta x \Delta y}\left(U_{i+1,j+1}(t) + U_{i-1,j-1}(t) - U_{i-1,j+1}(t) - U_{i+1,j-1}(t)\right) \end{aligned} \tag{3.41}$$

for $t \geq 0$ and $1 \leq i \leq m_1,\ 1 \leq j \leq m_2$.

It is nice to notice that the mixed derivative in the last line uses only a four point stencil as opposed to the nine point stencil for the non-uniform case seen in Figure 3.

3.5.2 A different representation of the prototype equation

Our aim is to represent the semi-discrete system with the correct lexicographic ordering that ADI time stepping schemes can solve. At the moment, U_{ij} corresponds to a grid point (x_i, y_j) stored as a matrix of values but this needs to be turned into a vector aligning with the definition of (3.36). With the boundaries we know the function value of $U_{i,j}$ at $t = 0$ for all points and also all function values on the boundaries from the Dirichlet condition. Now we will define some matrices that will aid in turning (3.41) into a vector form.

Denote

$$\mathbf{U}(t) \text{ the matrix with entries } U_{ij}(t) \text{ for } 1 \leq i \leq m_1,\ 1 \leq j \leq m_2.$$

This is the matrix where we will be approximating, so it excludes the area of the known Dirichlet boundary.

$$\widetilde{\mathbf{U}}(t) \text{ the matrix with entries } U_{ij}(t) \text{ for } 0 \leq i \leq m_1 + 1,\ 0 \leq j \leq m_2 + 1.$$

Now this is the notation used for a larger matrix including the Dirichlet boundary, so we have an extra row on the top and bottom and an extra column on the left and right.

To align with the Method of Lines approach we are still keeping the time variable continuous for the moment. Also, making use of our new notation $\widetilde{\mathbf{U}}(t)$ which is a matrix corresponding to the surface of grid points, Heath [27] has shown, in his case for the heat equation, the c_1 term of our example can be written in matrix form as

$$\begin{aligned} &\approx \frac{c_1}{2\Delta x} \begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{pmatrix} \mathbf{U}(t) \\ &= c_1 D_{1,x} \mathbf{U}(t) \end{aligned} \tag{3.42}$$

Where $D_{1,x}$ is defined as a differentiation matrix for a first order derivative in the x direction on a uniform grid. Now we will define the relevant matrices so that we can represent all of (3.41) in this way.

For any arbitrary integer $\nu \geq 2$ and $h = \frac{1}{\nu+1}$ define the matrices firstly for the first derivative

$$\widetilde{D}_{1,\nu} = \frac{1}{2h} \begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{pmatrix} \in \mathbb{R}^{(\nu+2)\times(\nu+2)}, \tag{3.43}$$

and for the second derivative

$$\widetilde{D}_{2,\nu} = \frac{1}{h^2} \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix} \in \mathbb{R}^{(\nu+2)\times(\nu+2)}. \tag{3.44}$$

These correspond to the central finite difference scheme for first and second order derivative. They are defined for the larger matrix including the Dirichlet condition, so the missing values outside the grid on the left and right of the top and bottom rows respectively are not so important.

Now we will introduce a useful operator $\mathcal{R}[\cdot]$ that deletes the top and bottom row and the left and right column of any given matrix.
Applying $\mathcal{R}[\cdot]$ to the two matrices we have

$$D_{1,\nu} = \mathcal{R}[\widetilde{D}_{1,\nu}] = \frac{1}{2h}\begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{pmatrix} \in \mathbb{R}^{\nu\times\nu}, \tag{3.45}$$

and

$$D_{2,\nu} = \mathcal{R}[\widetilde{D}_{2,\nu}] = \frac{1}{h^2}\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix} \in \mathbb{R}^{\nu\times\nu}. \tag{3.46}$$

Considering again our prototype equation, with the above facts, the following holds

$$\begin{aligned} \mathbf{U}'(t) = \mathcal{R}[&(c_1\widetilde{D}_{1,m_1} + d_{11}\widetilde{D}_{2,m_1})\widetilde{\mathbf{U}}(t) \\ &+ \widetilde{\mathbf{U}}(t)(c_2\widetilde{D}^T_{1,m_2} + d_{22}\widetilde{D}^T_{2,m_2}) \\ &+ 2d_{12}\widetilde{D}_{1,m_1}\widetilde{\mathbf{U}}(t)\widetilde{D}^T_{1,m_2}]. \end{aligned} \tag{3.47}$$

In [28], Wyns and Hout show a similar definition but instead for a local volatility model. We can think about expanding out the brackets and applying the $\mathcal{R}$ operator which will give us a representation of our semi-discrete system (3.41) in matrix notation. Let's do this to show we can get our example (3.42) and make a few observations. Multiplying out the top line, ignoring the d_{11} term, gets us $c_1\widetilde{D}_{1,m_1}\widetilde{\mathbf{U}}(t)$. Now applying the row and column deleting operator $\mathcal{R}[c_1\widetilde{D}_{1,m_1}\widetilde{\mathbf{U}}(t)]$ gives the example we presented like so $c_1 D_{1,x}\mathbf{U}(t)$. This is differentiating in the x direction, to differentiate in the y direction, we apply the transpose of the relevant order differentiation matrix on the right side of $\mathbf{U}(t)$ instead of the left side. To give an idea of why this works, lets look at differentiating in the x direction for the c_1 term. The matrix(or dot) product is computed between $\widetilde{D}_{1,x}$ and $\widetilde{\mathbf{U}}(t)$, which takes a row from the first matrix, corresponding to a finite difference

formula, and a column from the second matrix. The column in the second matrix holds values for grid points $x_0, x_1, \cdots, x_{m_1+1}$ with a fixed value for y so this gives an approximation for the x direction. Lastly, for the mixed derivative, the differentiation matrices lie on either side of $\widetilde{U}(t)$ as we need to differentiate in both directions.

Separating boundary information

Now we would like to manipulate our representation above to separate the Dirichlet boundary information from where we are approximating on the interior points. We will do this in the following way.

Let $\widetilde{\Gamma}$ denote the matrix, containing boundary information, with entries $\widetilde{\Gamma}_{ij}(t)$ for $0 \leq i \leq m_1 + 1$, $0 \leq j \leq m_2 + 1$ given by

$$\widetilde{\Gamma}_{ij}(t) = \begin{cases} \gamma(x_i, y_j, t) & \text{if } i = 0 \text{ or } i = m_1 + 1 \text{ or } j = 0 \text{ or } j = m_2 + 1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.48}$$

The only non-zero entries in this matrix are on the boundaries denoted by D in Figure 5. With this, we can consider $\widetilde{\mathbf{U}}(t)$ written like so

$$\widetilde{\mathbf{U}}(t) = \underbrace{\widetilde{\mathbf{U}}(t) - \widetilde{\Gamma}(t)}_{(1)} + \underbrace{\widetilde{\Gamma}(t)}_{(2)} \tag{3.49}$$

Notice that the $\widetilde{\Gamma}(t)$ will take away the elements from the Dirichlet boundary resulting in part (1) having the interior points as non-zero elements surrounded by zeroes matching the definition of $\mathbf{U}(t)$. Part (2) is the exact opposite of that definition with only non-zero elements on the boundary given by the function $\gamma(x, y, t)$ and zeroes inside.

Using this splitting in (3.47), it follows that

$$\begin{aligned} \mathbf{U}'(t) = &\left(c_1 D_{1,m_1} + d_{11} D_{2,m_1}\right) \mathbf{U}(t) \\ &+ \mathbf{U}(t) \left(c_2 D_{1,m_2}^\top + d_{22} D_{2,m_2}^\top\right) \\ &+ 2 d_{12} D_{1,m_1} \mathbf{U}(t) D_{1,m_2}^\top + G(t), \end{aligned} \tag{3.50}$$

with

$$\begin{aligned} G(t) = &\mathcal{R}\left[\left(c_1 \tilde{D}_{1,m_1} + d_{11} \tilde{D}_{2,m_1}\right) \widetilde{\Gamma}(t)\right] \\ &+ \widetilde{\Gamma}(t) \left(c_2 \tilde{D}_{1,m_2}^\top + d_{22} \tilde{D}_{2,m_2}^\top\right) \\ &+ 2 d_{12} \tilde{D}_{1,m_1} \widetilde{\Gamma}(t) \tilde{D}_{1,m_2}^\top. \end{aligned}$$

On the right hand side only $\mathbf{U}(t)$ appears where we will be making our approximations. $G(t)$ represents the additional information of the boundary as a matrix , it is analogous

to (3.50), replacing $\widetilde{U}$ with $\widetilde{\Gamma}$. We can also observe that: $G(t)$ is a linear sum of $\widetilde{\Gamma}(t)$ terms, and importantly at any time, t, we can compute $G(t)$ assuming we have the Dirichlet boundary condition, so it is a known matrix.

Getting to the system of ODEs

We will introduce a few more operators that will get us to a suitable form to then apply time discretisation schemes. Firstly, vec[$\cdot$] will be used to turn matrices into vectors by successively stacking columns below each other. Further, let $\otimes$ be the operator denoting the Kronecker product between two matrices where for two matrices, say S size $m \times n$ and L size $p \times q$, then the Kronecker product $S \otimes L$ is a matrix of size $pm \times qn$. Recall that we need to set up our semi-discrete system as (3.36) with the correct lexicographic ordering of vectors, the following property uses these two operators to make this conversion from matrices as we have been using so far.

Definition 3.1. If X, W, Y are any given matrices of sizes $m_1 \times m_1$, $m_1 \times m_2$ and $m_2 \times m_2$, respectively, then

$$\text{vec}[XWY^\top] = (Y \otimes X)\text{vec}[W]. \tag{3.51}$$

We have many cases where in which we encounter matrix multiplication of the from $XWY^\top$ which are of size $m_1 \times m_2$. Let's outline the three instances where they occur in (3.50), if we introduce identity matrices I_{m_1} and I_{m_2} the correspondence becomes more obvious. Doing this below with underbraces highlighting the positions, we can take a look for example at the first line

$$\underbrace{\Big(c_1 D_{1,m_1} + d_{11} D_{2,m_1}\Big)}_{\text{X}} \underbrace{\mathbf{U}(t)}_{\text{W}} \underbrace{I_{m_1}^\top}_{\text{Y}^\top}, \tag{3.52a}$$

the second line

$$\underbrace{I_{m_2}}_{\text{X}} \underbrace{\mathbf{U}(t)}_{\text{W}} \underbrace{\Big(c_2 D_{1,m_2}^\top + d_{22} D_{2,m_2}^\top\Big)}_{\text{Y}^\top}, \tag{3.52b}$$

and finally the third line

$$\underbrace{2d_{12} D_{1,m_1}}_{\text{X}} \underbrace{\mathbf{U}(t)}_{\text{W}} \underbrace{D_{1,m_2}^\top}_{\text{Y}^\top}. \tag{3.52c}$$

We should note here that the transpose of the identity matrix is the same as the original identity matrix $I_{m_1}^\top = I_{m_1}$. Furthermore we can apply the vec[$\cdot$] operator and use the property in Definition 5.1 with Kronecker products so that the terms look like so

$$\text{vec}[(c_1 D_{1,m_1} + d_{11} D_{2,m_1})\mathbf{U}(t) I_{m_1}] = (I_{m_1} \otimes (c_1 D_{1,m_1} + d_{11} D_{2,m_1}))\text{vec}[\mathbf{U}(t)] \quad (3.53a)$$

$$\text{vec}[I_{m_2}\mathbf{U}(t)(c_2 D_{1,m_2}^\top + d_{22} D_{2,m_2}^\top)] = ((c_2 D_{1,m_2}^\top + d_{22} D_{2,m_2}^\top) \otimes I_{m_2})\text{vec}[\mathbf{U}(t)] \quad (3.53b)$$

$$\text{vec}[2d_{12} D_{1,m_1}\mathbf{U}(t) D_{1,m_2}^\top] = (D_{1,m_2}^\top \otimes 2d_{12} D_{1,m_1})\text{vec}[\mathbf{U}(t)] \quad (3.53c)$$

An important observation is that $\text{vec}[\mathbf{U}(t)]$ occurs on the right of each term. We can factor it out as we know the specific form for the system of ODEs we are trying to aim for, leaving us with a new matrix A below

$$A = A_0 + A_1 + A_2, \quad (3.54a)$$

$$A_1 = I_{m_1} \otimes (c_1 D_{1,m_1} + d_{11} D_{2,m_1}), \quad (3.54b)$$

$$A_2 = (c_2 D_{1,m_2} + d_{22} D_{2,m_2}) \otimes I_{m_2} \quad (3.54c)$$

$$A_0 = D_{1,m_2} \otimes 2d_{12} D_{1,m_1}, \quad (3.54d)$$

Additionally, we have to apply the operator $\text{vec}[\cdot]$ to the following two remaining parts of (3.50), $U'(t) = \text{vec}[\mathbf{U}'(t)]$ and $g(t) = \text{vec}[G(t)]$. Taking into account the factor we took out earlier, $\text{vec}[\mathbf{U}(t)]$, we arrive with the correct lexicographic ordering for the representation of our semi-discrete system shown again below

$$U'(t) = AU(t) + g(t). \quad (3.55)$$

There are also some points to note about the A matrices. The linear algebra based approach allows us to simply compute these matrices with formulas that can be easily implemented computationally. Although, the use of the Kronecker products leads to very sparse matrices, this however is common amongst all definitions of A in the literature [4]. So implementations need to make use of sparse matrix algorithms in programming libraries. Additionally, the two matrices A_1 and A_2 are tri-diagonal and near tri-diagonal respectively while A_0, for the mixed derivative, has four diagonals far away from the main diagonal which results in the full matrix A having a bandwidth equal to m. This structure is important as [30] point out this can lead to expensive operations however we will see how ADI schemes can resolve the issue by making good use of the lower bandwidth A_1 and A_2 matrices.

3.6 The Heston Problem

The above was for the prototype equation, let's now move back to Heston's PDE. We start by defining first and second order differentiation matrices now in terms of the S and v directions, noting that the double subscript is for second order

[4] See [29] for pictures of these matrices, our Kronecker product approach results in the same structure for the Heston matrices in the next section.

$$\widetilde{D}_S, \widetilde{D}_{SS} \in \mathbb{R}^{(m_1+1)\times(m_1+1)} \quad \text{and} \quad \widetilde{D}_v, \widetilde{D}_{vv} \in \mathbb{R}^{(m_2+1)\times(m_2+1)}.$$

Let D_S, D_{SS} be given by deleting the first row and column of $\widetilde{D}_S, \widetilde{D}_{SS}$.
Let D_v, D_{vv} be given by deleting the last row and column of $\widetilde{D}_v, \widetilde{D}_{vv}$.

These differentiation matrices aren't as simple as those described for the prototype convection-diffusion equation for two reasons. The first of which is that we are now back to non-uniform grids, so the elements inside the matrices correspond to the more intricate coefficients α, β, γ and δ. The second reason is that we have a slightly more complicated boundary surrounding our computational grid and as a result there are instances where more than one finite difference scheme is used in one matrix.

3.6.1 Implementation of differentiation matrices

We will take a look at the construction of the differentiation matrix $\widetilde{D}_v$, it serves as a nice illustration because it is the most involved out of the bunch. The reasoning for that is its requirement of the three finite difference schemes (3.16a), (3.16b) and (3.16c) as detailed in Section 3.3.3. To save on computation time and memory we have taken a vectorised approach in the code where speed is investigated later in Section 4.1.2 along with the A matrix. There, we compare our implementation against a non-vectorised example and they also don't make use of the Kronecker product formulas. Hence, we will step through our code here as motivation for our choices.

Listing 1: Computing β's in v direction

```
Delta_v = np.diff(v[:m2_tilde])
Delta_v_i = Delta_v[:-1]
Delta_v_i_1 = Delta_v[1:]

beta_minus_v = -Delta_v_i_1 / (Delta_v_i *      \
    (Delta_v_i + Delta_v_i_1))

beta_plus_v = Delta_v_i / (Delta_v_i_1 *        \
    (Delta_v_i + Delta_v_i_1))

beta_zero_v = -(beta_minus_v + beta_plus_v)
```

We start off by computing a variable `Delta_v` which is a vector of length m_2 containing successive widths to the right in the v direction of our mesh. The variable v holds these mesh points. Now on `Delta_v`, slicing operations can be performed to get Δv_i and Δv_{i+1}, which are stored in `Delta_v_i` and `Delta_v_i_1` respectively. They are inside the coefficients of the central scheme (3.16b). To compute these coefficients, we can now plug in our variables as done in the formulas to get `beta_minus_v` as $\beta_{i,-1}$ and `beta_plus_v` as $\beta_{i,+1}$. For $\beta_{i,0}$, we simply make use of the relation (3.23a) by putting the central term on the left-hand side. The same type of operations are used to compute the coefficients

for the backward scheme, so for brevity, assume we have defined `alpha_minus_2_v` for $\alpha_{i,-2}$, `alpha_minus_v` for $\alpha_{i,-1}$, and `alpha_zero_v` for $\alpha_{i,0}$.

Now we will create two differentiation matrices for the central and backward schemes which will be combined to form the final matrix $\widetilde{D}_v$ along with the forward scheme last.

Listing 2: Central and Backward differentiation matrices

```
inputs = [beta_minus_v, beta_zero_v, beta_plus_v]
D_v_central_tilde = diags(inputs, offsets_central,    \
    shape=(m2_tilde, m2_tilde))

inputs = [alpha_minus_2_v, alpha_minus_v, alpha_zero_v]
D_v_backward_tilde = diags(inputs, offsets_backward,  \
    shape=(m2_tilde, m2_tilde))
```

The first three lines for the central scheme will fill the diagonals of a matrix where the offsets are `[-1, 0, 1]` for the coefficients, resulting in

$$\texttt{D\_v\_central\_tilde} = \begin{pmatrix} \boldsymbol{\beta_{0,1}} & \beta_{1,1} & & & & \\ \beta_{-1,2} & \boldsymbol{\beta_{0,2}} & \beta_{1,2} & & & \\ & \beta_{-1,3} & \boldsymbol{\beta_{0,3}} & \beta_{1,3} & & \\ & & \ddots & \ddots & & \\ & & & \beta_{-1,m_2} & \boldsymbol{\beta_{0,m_2}} & \beta_{1,m_2} \\ & & & & \beta_{-1,m_2+1} & \boldsymbol{\beta_{0,m_2+1}} \end{pmatrix},$$

and the last three lines instead for the backward scheme use offsets `[-2, -1, 0]` to get

$$\texttt{D\_v\_backward\_tilde} = \begin{pmatrix} \boldsymbol{\alpha_{0,1}} & & & & & \\ \alpha_{-1,2} & \boldsymbol{\alpha_{0,2}} & & & & \\ \alpha_{-2,3} & \alpha_{-1,3} & \boldsymbol{\alpha_{0,3}} & & & \\ & \ddots & \ddots & & & \\ & \alpha_{-2,m_2} & \alpha_{-1,m_2} & \boldsymbol{\alpha_{0,m_2}} & & \\ & & \alpha_{-2,m_2+1} & \alpha_{-1,m_2+1} & \boldsymbol{\alpha_{0,m_2+1}} \end{pmatrix}.$$

The leading diagonal of the matrices are highlighted with bold lettering. This serves to show that the schemes on their own aren't appropriate for approximating derivatives as both schemes would need to use points which are outside of the grid.

Now recall from Section 3.3.3 we wanted to use the forward scheme at $v = 0$, the central scheme for $0 < v < 1$ and backward for $v \geq 1$.

Listing 3: Putting everything together for $\widetilde{D}_v$

```
D_v_tilde = D_v_backward_tilde.tocsr()

D_v_central_tilde = D_v_central_tilde.tocsr()


D_v_tilde[:i_1, :] = D_v_central_tilde[:i_1, :]


Delta_v_i_1 = Delta_v[0]
Delta_v_i_2 = Delta_v[1]

D_v_tilde[0, :] = 0
D_v_tilde[0, 0] = ((-2 * Delta_v_i_1) - Delta_v_i_2)  \
    / (Delta_v_i_1 * (Delta_v_i_1 + Delta_v_i_2))

D_v_tilde[0, 1] = (Delta_v_i_1 + Delta_v_i_2)         \
    / (Delta_v_i_1 * Delta_v_i_2)

D_v_tilde[0, 2] = -Delta_v_i_1 / (Delta_v_i_2 *       \
    (Delta_v_i_1 + Delta_v_i_2)
```

This is done by laying the central scheme matrix over the backward scheme matrix , only up until the point where $v < 1$ represented by the index `i_1`. Further, in the last four lines we implement the γ coefficients for the forward scheme at the boundary $v = 0$. This

has now constructed our final matrix $\widetilde{D}_v$ which is used to perform the approximation of the first derivative in the v direction, illustrated below

$$\widetilde{D}_v = \begin{pmatrix} \boldsymbol{\gamma_{0,1}} & \gamma_{1,1} & \gamma_{2,1} & & & & & & \\ \beta_{-1,2} & \boldsymbol{\beta_{0,2}} & \beta_{1,2} & & & & & & \\ & \beta_{-1,3} & \boldsymbol{\beta_{0,3}} & \beta_{1,3} & & & & & \\ & & \ddots & \ddots & & & & & \\ & & & \beta_{-1,k} & \boldsymbol{\beta_{0,k}} & \beta_{1,k} & & & \\ & & & \alpha_{-2,k+1} & \alpha_{-1,k+1} & \boldsymbol{\alpha_{0,k+1}} & & & \\ & & & & \alpha_{-2,k+2} & \alpha_{-1,k+2} & \boldsymbol{\alpha_{0,k+2}} & & \\ & & & & & \ddots & \ddots & & \\ & & & & & \alpha_{-2,m_2+1} & \alpha_{-1,m_2+1} & \boldsymbol{\alpha_{0,m_2+1}} \end{pmatrix} \tag{3.56}$$

where $\widetilde{D}_v \in \mathbb{R}^{(m_2+1)\times(m_2+1)}$ and k is analogous to the index `i_1`.

Looking at the leading, highlighted diagonal we can notice the forward scheme in the first row, indicated by gammas, makes use of two cells to the right. Then switching to the central scheme, the derivative is approximated by having the central coefficient on the leading diagonal and finally the switch is made to the backward scheme using two points to the left-hand side. The rest of the differentiation matrices $\tilde{D}_S, \tilde{D}_{SS}$, and $\tilde{D}_{vv}$ are constructed in the same manner, the only difference being the requirements laid out in Section 3.3.3 and the coeffecients used.

3.6.2 The Heston problem as a system of ODEs

Recall Heston's PDE doesn't have constant coefficients, as S and v can vary, so we define diagonal matrices where the diagonal are the S and v points from the grid. We also consider their restrictions.

$$\widetilde{X} = \operatorname{diag}(S_0, S_1, \ldots, S_{m_1}), \quad X = \operatorname{diag}(S_1, \ldots, S_{m_1-1}),$$
$$\widetilde{Y} = \operatorname{diag}(v_0, v_1, \ldots, v_{m_2}), \quad Y = \operatorname{diag}(v_1, \ldots, v_{m_2-1}).$$

Then for Heston, we can consider A similarly as before but with slight modifications, see Khasi and Rashidinia [31] for a general derivation

$$A = A_0 + A_1 + A_2, \tag{3.57a}$$
$$A_0 = \rho\sigma \left(Y D_v\right) \otimes \left(X D_S\right), \tag{3.57b}$$
$$A_1 = Y \otimes \left(\frac{1}{2} X^2 D_{SS}\right) + I_v \otimes \left(r X D_S\right) - \frac{1}{2} r I_v \otimes I_S, \tag{3.57c}$$
$$A_2 = \left(\frac{1}{2}\sigma^2 Y D_{vv}\right) \otimes I_S + \left(\kappa(\theta I_v - Y) D_v\right) \otimes I_S - \frac{1}{2} r I_v \otimes I_S. \tag{3.57d}$$

Additionally, this representation is identical to Equation (2.1) in Hout and Volder's paper [32], noting the change in variable names and the splitting as later practical stability bounds for convection and diffusion terms are provided separately. Nevertheless we can make some observations. Again, A_0 represents the mixed derivative term while A_1 and A_2 are for the derivatives in the S and v directions respectively. Furthermore, the modifications we needed to consider are for the coefficients; the constant variables can easily be swapped in to the prototype version of A while the diagonal matrices representing the S and v grid points need to be placed on the left side of the differentiation matrices so that the correct elements from X and Y are taken. The last important note is the popular choice, see for example [29], to distribute the reaction term, $-rU$, from the Heston PDE by a half across (3.57c) and (3.57d).

Boundary Information

In a similar manner to before, we need a matrix to represent the boundaries. Here we will instead use two separate matrices to represent both Dirichlet and Neumann boundary conditions.

Let $\widetilde{\Gamma}$ be the matrix, containing all the Dirichlet information, with entries $\widetilde{\Gamma}_{ij}$ for $0 \leq i \leq m_1$, $0 \leq j \leq m_2$ given by

$$\widetilde{\Gamma}_{ij} = \begin{cases} S_i & \text{if } j = m_2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.58}$$

From Section 3.3.3, we only need to insert S_i on the $v = V_{max}$ boundary as due to the other condition, when $S = S_0$, only requires inserting zeroes.

Let $\widetilde{E}$ be the matrix, containing all Neumann information, with entries $\widetilde{E}_{ij}$ for $0 \leq i \leq m_1$, $0 \leq j \leq m_2$ given by

$$\widetilde{E}_{ij} = \begin{cases} 1 & \text{if } i = m_1, \\ 0 & \text{otherwise.} \end{cases} \tag{3.59}$$

For this boundary we now only have 1s inserted at the boundary $S = S_{max}$ for the Neumann condition.

Decomposing the G matrix:
Considering the Dirichlet boundary, we slightly change the definition of the operator $\mathcal{R}[\cdot]$ to only delete the first row and last column of any given matrix. With $\nu = 2/\Delta s_{m_1}$, Hout [6] defines

$$G = G_0 + G_1 + G_2, \tag{3.60a}$$

$$G_0 = \mathcal{R}\left[\rho\sigma\widetilde{X}\widetilde{D}_s\widetilde{\Gamma}\widetilde{D}_v^\top\widetilde{Y}\right], \tag{3.60b}$$

$$G_1 = \mathcal{R}\left[\frac{1}{2}\widetilde{X}^2\left(\widetilde{D}_{ss}\widetilde{\Gamma} + \nu\widetilde{E}\right)\widetilde{Y} + r\widetilde{X}\left(\widetilde{D}_s\widetilde{\Gamma} + \widetilde{E}\right)\right], \tag{3.60c}$$

$$G_2 = \mathcal{R}\left[\frac{1}{2}\sigma^2\widetilde{\Gamma}\widetilde{D}_{vv}\widetilde{Y} + \kappa\widetilde{\Gamma}\widetilde{D}_v(\theta I_v - \widetilde{Y})\right]. \tag{3.60d}$$

These are all known matrices, independent of t, where the decomposition is similar to the A matrix we saw, however Hout notes extra care must be taken to arrive at these due to the extra Neumann boundary. They hold the boundary information used for the mixed derivative, the derivatives in the S direction, and v direction respectively.

This now leads us to the last step, where we can follow our approach to the prototype equation by converting the boundary matrix like so $g(t) = \text{vec}[G]$ and we arrive at the representation of (3.36). We should emphasise that this representation is exactly our semi-discrete system in vector form.

3.7 Discretisation in Time

So far we have kept the time variable continuous; in accordance with the second step of the MOL approach we want to discretise the Heston PDE in time. As we have the starting vector U_0, determined by the initial condition, we can make incremental steps forward in time to reach any state of the solution vector. A similar approach can be taken to spatial discretisation; we will take discrete step sizes in the time direction $\Delta t = T/N$ with given integer $T \geq 1$. This gives us a grid(or temporal grid) in the time direction with grid points $t_n = n\Delta t$ where $n = 1, 2, \ldots, N$. Additionally, as stated previously we can evaluate the boundary vector from (3.36) at any time so we write it as $g_n = g(t_n)$.

3.7.1 θ-method

To do this, we first consider a popular family of methods called θ-methods where the parameter $\theta \in [0, 1]$. It defines an approximation U_n to $U(t_n)$ from the semi-discrete system (3.36) by

$$\frac{U_n - U_{n-1}}{\Delta t} = (1-\theta)[AU_{n-1} + g_{n-1}] + \theta[AU_n + g_n] \iff$$
$$U_n = U_{n-1} + (1-\theta)\Delta t[AU_{n-1} + g_{n-1}] + \theta\Delta t[AU_n + g_n]. \tag{3.61}$$

This is a one step process that takes you from t_{n-1} to t_n where we can start with our initial vector at t_0. In both instances of the above, we see that U_n is on both sides of the equation, if we rearrange as follows

$$U_n - \theta\Delta t A U_n = U_{n-1} + (1-\theta)\Delta t[AU_{n-1} + g_{n-1}] + \theta\Delta t g_n, \tag{3.62}$$

then introduce an identity matrix we get

$$(I - \theta\Delta t A)U_n = (I + (1-\theta)\Delta t A)\, U_{n-1} + (1-\theta)\Delta t g_{n-1} + \theta\Delta t g_n \tag{3.63}$$

Now U_n only appears on the left-hand side of our equation which is what we want to be able to approximate the next step. For the moment we take the simple case of $\theta = 0$ that causes the term $I - \theta\Delta t A$ to vanish meaning we can directly compute the next time step without much trouble, this is an explicit method called the forward Euler method. Nonetheless, it is not suitable for the semidiscrete Heston PDE which is stiff in general. Stiffness meaning that when an explicit method is used to solve a system of ODEs, there is a strict requirement on the time step related to the spatial grid widths to ensure a stable solution. A common stability requirement for a square grid system, for a constant c, is

$$\Delta t \leq c[(\Delta S)^2 + (\Delta v)^2].$$

This means that the time step size Δt needs to be smaller than the sum of the squares of the spatial widths ΔS and Δv. If you want to reasonably increase the resolution on your grid, i.e having smaller spatial widths, your time step must decrease so much that it becomes too inefficient to solve in practice, ruling it out as a possible implementation choice.

Instead for the case when $\theta > 1$, we have $I - \theta\Delta t A$ on the left-hand side which is known and all the terms on the right-hand side are known too. This time to solve for the next time step U_n we need to instead solve a linear system which is of the form $BU_n = C$. Therefore it is known as an implicit method, the current state and future state of the system are solved together.

There are a few common approaches to do this, without requiring expensive matrix inversion, for example Hout and Foulon [17] conclude using LU decomposition of $I - \theta\Delta t A$ is the most efficient. Although due to the structure of the A matrix, the lower triangular and upper triangular matrices result in having many more non-zero elements where the original matrix had zeroes, which is quite inefficient for later calculations. For this reason

we have chosen a different approach in our implementation to avoid this work with the combined A matrix, we move to ADI schemes.

Standard cases for θ-method:

- $\theta = 0$: forward Euler method (Explicit, order 1),
- $\theta = \frac{1}{2}$: Crank–Nicolson method (Implicit, order 2),
- $\theta = 1$: backward Euler method (Implicit, order 1).

3.7.2 ADI method

In the previous section, we concluded that the θ-methods are not appropriate for the Heston problem. We now look to alternating direction implicit (ADI) schemes, which are far more efficient for convection-diffusion-reaction type equations with mixed derivatives while also not having strict restrictions on the time step and grid size. ADI schemes were first introduced by Peaceman and Rachford [33], they make use of both implicit and explicit computations inside each actual time step which allows us to solve a simpler tri-diagonal system of equations. This is possible due to the splitting of the A and g matrices that ADI schemes consider, the following is a standard approach for our equation's type:

$$A = A_0 + A_1 + A_2 \tag{3.64}$$

where

- A_0 corresponds to $\frac{\partial^2 U}{\partial S \partial v}$ term,
- A_1 corresponds to $\frac{\partial U}{\partial S}, \frac{\partial^2 U}{\partial S^2}$ terms,
- A_2 corresponds to $\frac{\partial U}{\partial v}, \frac{\partial^2 U}{\partial v^2}$ terms.

With analogous splitting of $g(t)$

$$g(t) = g_0(t) + g_1(t) + g_2(t). \tag{3.65}$$

We have conveniently defined our matrices to correspond to this splitting in Sections 3.5.2 and 3.6.2. The ADI method breaks down our system of equations into simpler sub problems, that treat one spatial direction at each sub-step (or stage) implicitly. At each stage only one direction is dealt with in an implicit manner while the other is dealt with explicitly, then for the next stage the opposite occurs, the direction dealt with explicitly is now done so implicitly and vice versa for the other direction. This is all done inside each time step t_{n-1} to t_n. The reason this is advantageous is that when one direction is treated implicitly it requires it's corresponding A_1 or A_2 matrix to be

solved in the linear system which is computationally efficient due to their tri-diagonal structure. More importantly, the mixed derivative is only ever treated in the first stage, the explicit forward Euler way. Due to the high bandwidth of the A_0 matrix it saves much computation time not being solved in this linear system.

3.7.2.1 Idea behind the Douglas scheme

We will first present the simplest ADI scheme that we consider, developed by Douglas [34] in collaboration with Peaceman and Rachford. As stated above it solves each time step split into stages, where at each stage we store the intermediate solution as Y_i where $i = 1, ..., n$.

Considering our system of ODEs (3.36) we discretise with respect to t:

$$\frac{U_n - U_{n-1}}{\Delta t} = AU_{n-1} + g(t). \tag{3.66}$$

As before, isolating U_n on the left side

$$U_n = U_{n-1} + \Delta t A U_{n-1} + \Delta t g(t). \tag{3.67}$$

This is the first intermediate solution, also known as the predictor stage, a fully explicit step involving the combined A matrix. It is still a valid approximation to the system but it is very crude. Therefore we store it as Y_0 shown below

$$Y_0 = U_{n-1} + \Delta t A U_{n-1} + \Delta t g(t) \tag{3.68}$$

We now want to increase the accuracy in the S direction so we treat the first so-called corrector stage Y_1 in the following implicit style

$$\begin{aligned} Y_1 = Y_0 + \theta\Delta t(A_1 Y_1 + g_1(t_n) - A_1 U_{n-1} + g_1(t_{n-1})) \implies \\ (I - \theta\Delta t A_1)Y_1 = Y_0 - \theta\Delta t(g_1(t_n) - A_1 U_{n-1} + g_1(t_{n-1})). \end{aligned} \tag{3.69}$$

So far, only the S direction is corrected implicitly and the other only treated explicitly in (3.69). Hence, we correct the v direction implicitly below

$$\begin{aligned} Y_2 = Y_1 + \theta\Delta t(A_2 Y_2 + g_2(t_n) - A_1 U_{n-1} + g_2(t_{n-1})) \implies \\ (I - \theta\Delta t A_2)Y_2 = Y_1 - \theta\Delta t(g_2(t_n) - A_2 U_{n-1} + g_2(t_{n-1})). \end{aligned} \tag{3.70}$$

In both cases we need to solve for Y_1 and Y_2. Moving on, since we have been updating our Y_j variable to hold the predictor followed by two correction stages, our approximation to the next step is that of the last correction.

$$U_n = Y_2. \tag{3.71}$$

All later defined schemes begin the same way, however introducing more intermediate steps.

Implementation to solve linear systems efficiently
To summarise, the A_0 term is only ever treated explicitly while the A_1 and A_2 terms are corrected implicitly. As a result, per actual time step the Douglas scheme requires us to solve two linear systems, not involving the awkward A_0 matrix but instead $(I - \theta\Delta t A_1)$ and $(I - \theta\Delta t A_2)$ in (3.69) and (3.70) respectively. As mentioned before, LU decomposition is an efficient method to solve them, so later on we will investigate against an implementation that instead chooses matrix inversion.

Another way to save computation time specific to ADI schemes is to notice that those two matrices are independent of time, the Δt width remains constant as well as the other terms. So before stepping into the ADI loops, from $t = 1, ..., N$, we compute their LU factorisation once and use it at every step. We present the following code:

Listing 4: LU Decomposition

```
LHS_1 = sp.eye(I.shape[0]) - (theta * dt) * A_1
LHS_2 = sp.eye(I.shape[0]) - (theta * dt) * A_2

LU1 = spla.splu(LHS_1)
LU2 = spla.splu(LHS_2)

del A_1, A_2
```

This step is crucially performed before the ADI loop. We simply use SciPy's sparse `linalg` library to perform an LU decomposition and further delete the A_1 and A_2 matrix as they are now no longer needed. This is the precursor to solving the linear system by LU Decomposition.

Listing 5: ADI loop

```
for n in range(1, N + 1):
    gr = g
    gr0 = g_0
    gr1 = g_1
    gr2 = g_2
    u = Do(u, dt, LU1, LU2, A, g, g_1, gr1, g_2, gr2, theta)
    g = gr
    g_0 = gr0
    g_1 = gr1
    g_2 = gr2
```

Now stepping into the ADI loop where we also have to store the previous and current matrices for $g(t)$. Additionally, we take in the decomposed matrices as `LU1` and `LU2`, where we treat them in the following manner inside the Douglas functon

Listing 6: Douglas function

```
def Do(u, dt, LU1, LU2, A, prev, prev_1, next_1, prev_2,   \
    next_2, theta):

    # Explicit Predictor stage
    z0 = dt * (A @ u + prev)

    # Two Implicit Corrector stages
    z = LU1.solve(z0 + theta * dt * (next_1 - prev_1))
    z = LU2.solve(z + theta * dt * (next_2 - prev_2))

    u = u + z

    return u
```

Now the final important step is solving the linear system by making use of the decomposed matrices. The `solve` function makes use of a popular approach using forward and backward substitution outlined here [35] by West. Essentially, you first solve with the lower triangular matrix in the forward step, and use the result of that to solve in the backward step with the upper triangular matrix.

3.7.2.2 Douglas scheme

To make the schemes easier to read we define the following functions, for $0 \leq t \leq T$ and $w \in \mathbb{R}^m$, let

$$F(t, w) = Aw + g(t), \tag{3.72}$$

with necessary splittings as

$$F_j(t, w) = A_j w + g_j(t), \quad (j = 0, 1, 2). \tag{3.73}$$

All together combining the intermediate steps from Section 3.7.2.1, the Douglas scheme is as follows

$$\begin{aligned} Y_0 &= U_{n-1} + \Delta t F(t_{n-1}, U_{n-1}), \\ Y_j &= Y_{j-1} + \theta \Delta t \left(F_j(t_n, Y_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2), \\ U_n &= Y_2. \end{aligned} \tag{3.74}$$

It is said to have an order of 2 for $\theta = \frac{1}{2}$, however order 1 when A_0 is non-zero[5]. Also see [29] for a proof that the Douglas scheme approximates the Crank-Nicolson method, even though it is more efficient for mixed derivative problems.

[5]This is common in practice, $A_0 = 0$ only when the correlation between the two processes $\rho = 0$.

3.7.2.3 Craig-Sneyd scheme

Here a new explicit step is introduced on the third line, followed by two more implicit corrections.

$$\begin{aligned}
Y_0 &= U_{n-1} + \Delta t F(t_{n-1}, U_{n-1}),\\
Y_j &= Y_{j-1} + \theta\Delta t \left(F_j(t_n, Y_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2),\\
\widetilde{Y}_0 &= Y_0 + \frac{1}{2}\Delta t \left(F_0(t_n, Y_2) - F_0(t_{n-1}, U_{n-1})\right),\\
\widetilde{Y}_j &= \widetilde{Y}_{j-1} + \theta\Delta t \left(F_j(t_n, \widetilde{Y}_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2),\\
U_n &= \widetilde{Y}_2.
\end{aligned} \tag{3.75}$$

Order 2 if $\theta = \frac{1}{2}$

3.7.2.4 Modified Craig-Sneyd scheme

The modification of Craig-Sneyd's scheme improves accuracy to second order for all θ values by introducing a correction term in the explicit step.

$$\begin{aligned}
Y_0 &= U_{n-1} + \Delta t F(t_{n-1}, U_{n-1}),\\
Y_j &= Y_{j-1} + \theta\Delta t \left(F_j(t_n, Y_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2),\\
\widetilde{Y}_0 &= Y_0 + \theta\Delta t \left(F_0(t_n, Y_2) - F_0(t_{n-1}, U_{n-1})\right)\\
&\quad + \left(\frac{1}{2} - \theta\right)\Delta t \left(F(t_n, Y_2) - F(t_{n-1}, U_{n-1})\right),\\
\widetilde{Y}_j &= \widetilde{Y}_{j-1} + \theta\Delta t \left(F_j(t_n, \widetilde{Y}_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2),\\
U_n &= \widetilde{Y}_2.
\end{aligned} \tag{3.76}$$

Order 2 for all θ.

3.7.2.5 Hundsdorfer-Verwer (HV) scheme

A further modification is made for the weightings in the second explicit step.

$$
\begin{aligned}
Y_0 &= U_{n-1} + \Delta t F(t_{n-1}, U_{n-1}), \\
Y_j &= Y_{j-1} + \theta \Delta t \left(F_j(t_n, Y_j) - F_j(t_{n-1}, U_{n-1})\right) \quad (j = 1, 2), \\
\widetilde{Y}_0 &= Y_0 + \frac{1}{2}\Delta t \left(F(t_n, Y_2) - F(t_{n-1}, U_{n-1})\right), \\
\widetilde{Y}_j &= \widetilde{Y}_{j-1} + \theta \Delta t \left(F_j(t_n, \widetilde{Y}_j) - F_j(t_n, Y_2)\right) \quad (j = 1, 2), \\
U_n &= \widetilde{Y}_2.
\end{aligned}
\tag{3.77}
$$

Order 2 for all θ.

3.8 The Greeks

The Greeks measure the sensitivity of an options price to changes in certain parameters. They play an extensive role in financial risk management, such as hedging tools, or can help traders with their intuition. The finite difference approximation we have performed in this section is an incredibly useful way to calculate the Greeks, in essence we have already computed them to solve the PDE. Nevertheless, we produce option prices for a large selection of values for the underlying and volatility so we can directly apply our finite difference schemes to approximate the partial derivative of U with respect to certain parameters. We shall consider these

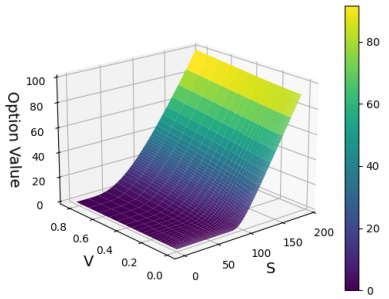
$$\text{Delta} = \frac{\partial u}{\partial s}, \text{Gamma} = \frac{\partial^2 u}{\partial s^2}, \text{Vega} = \frac{\partial u}{\partial v}. \tag{3.78}$$

Notice Vega uses the variance v, in the Black-Scholes case it would be with respect to the volatility σ, but a choice of variance is commonly made for stochastic volatility models.

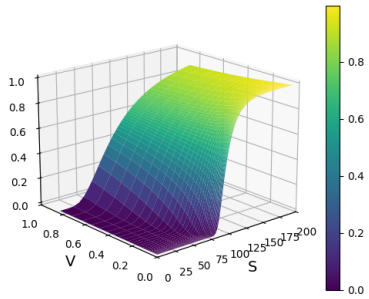
We convert our price vector into a matrix, $\widetilde{\mathbf{U}}_n$ where the i and j-th entry approximates $u(s_i, v_j, t_n)$ for $0 \leq i \leq m_1, 0 \leq j \leq m_2$. Further, we define the differentiation matrices as in Section 3.6: $\widetilde{D}_s, \widetilde{D}_{ss}, \widetilde{D}_v$. To compute the Greeks we just perform simple matrix multiplication as shown below

$$\widetilde{D}_s \widetilde{U}_n, \quad \widetilde{D}_{ss} \widetilde{U}_n, \quad \widetilde{U}_n \widetilde{D}_v^\top,$$

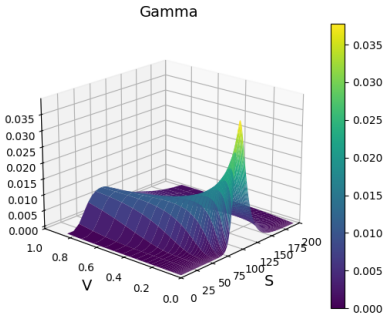
respectively to approximate Delta, Gamma and Vega. See their surface plots in Figure 6.



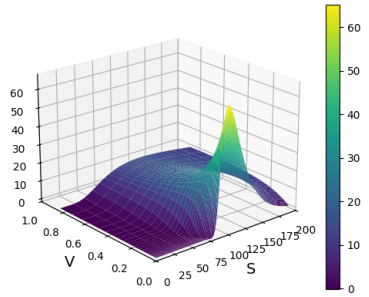
(a) Option Price Surface



(b) Delta Surface



(c) Gamma Surface



(d) Vega Surface

Figure 6: Surface plots computed via MCS scheme

4 Results and Analysis

4.1 A comparison of approaches

When considering numerical approximation approaches for pricing options, speed is often a large factor influencing the choice of method. In our implementation we have followed a vectorised approach, as outlined in Chapter 5. A key area which has been vectorised, is our handling of the A matrix utilising Kronecker products. This approach is not as common in the literature so we will firstly have to make sure that our prices are good approximations by comparing to a more standard implementation [36], following this we can compare implementation specific choices.

4.1.1 Accuracy

To confirm accuracy, the relative absolute error will be measured between our implementation and that of [36], against their Monte-Carlo simulation. This price is accurate

as it gets a near exact match to option values calculated from closed-form solution in Section 2.6 using QuantLib's Python library. The European call option price we get with strike price $K = 100$, initial asset price $S_0 = 120$, initial variance $v_0 = 0.4$ and the following Heston parameters

$$\kappa = 1.5, \theta = 0.04, \sigma = 0.3, \rho = -0.9, r = 0.025, T = 1$$

is

True value	33.773423

The finite difference approach produces a price surface, so we use interpolation to approach the desired grid point (S_0, v_0) to find our price approximation. For a fair comparison between the two finite difference approaches, we need to keep the sizes of the grids consistent. Choosing the number of points in the S-direction as $m_1 = 100$ and in the v-direction as $m_2 = 50$, $\theta = 0.8$ and for the time step $N = 100$ are used. The following results have been obtained:

$$\text{Error} = \frac{\text{price}_{\text{FD method}} - \text{price}_{\text{Analytical}}}{\text{price}_{\text{Analytical}}} \times 100$$

My Implementation		
Method	Price	% error
DO	33.741413	0.0948%
CS	33.740924	0.0962%
MCS	33.759496	0.0412%
HV	33.769839	0.0106%

Other Implementation		
Method	Price	% error
DO	33.747852	0.0757%
CS	33.747250	0.0775%
MCS	33.765745	0.0227%
HV	33.765777	0.0226%

Table 1: Relative absolute errors for different methods

From Table 1 we can see that both implementations result in values that are quite close to the True value and as a result have low errors. Furthermore, this confirms that our approach is working correctly. We do however see the first three methods are slightly more accurate for [36] while HV is more accurate for us. The accuracy may vary from grid point to grid point so later we check global errors which take measurements from a large section of the grid. Nevertheless, we move to computational speed for now.

4.1.2 Computation speed

In this section there will be a comparison of the computation speed between design choices made in both implementations. Firstly considering the A matrix. In our implementation we take a vectorised approach, shown in Section 3.6.1, to construct the differentiation matrices which are subsequently plugged into the formulas for the A matrix, from Section 3.6.2, along with the Heston parameters. However, the implementation

[36] uses nested for loops to construct the A matrix that still follow the same requirements. To measure the time taken to construct the matrices, in both cases the `datetime` Python package is used, we should see speed improvements in our case. The size of the grids used is given by $m_1 = 2m_2$ and we test using the Modified Craig-Sneyd scheme, other schemes provide similar results. The tests are run on a machine with a 3.5Ghz Ryzen 5 CPU with 6 cores and 16GB of memory.

My Implementation	
Grid size	Time (seconds)
$m_1 = 50$	0.007001
$m_1 = 75$	0.009002
$m_1 = 100$	0.010003
$m_1 = 125$	0.012002
$m_1 = 150$	0.015004
$m_1 = 175$	0.018004

Other Implementation	
Grid size	Time (seconds)
$m_1 = 50$	0.056013
$m_1 = 75$	0.241053
$m_1 = 100$	0.695162
$m_1 = 125$	1.586706
$m_1 = 150$	3.180159
$m_1 = 175$	5.577352

Table 2: Comparison of Time (in seconds) for computing A matrices

Another design choice we made was solving the linear system that arises in the ADI loops using LU decomposition techniques outlined in Section 3.7.2.1. The implementation of [36] instead solves it by inversion of sparse matrices using SciPy. SciPy's documentation doesn't state the algorithms employed however in general sparse matrix inversion is quite costly.

My Implementation	
Grid size	Time (seconds)
$m_1 = 50$	0.021005
$m_1 = 75$	0.039008
$m_1 = 100$	0.060017
$m_1 = 125$	0.080018
$m_1 = 150$	0.116023
$m_1 = 175$	0.152035

Other Implementation	
Grid size	Time (seconds)
$m_1 = 50$	0.440102
$m_1 = 75$	1.338717
$m_1 = 100$	3.226246
$m_1 = 125$	6.673401
$m_1 = 150$	12.610076
$m_1 = 175$	23.723955

Table 3: Comparison of Time (in seconds) for finishing the ADI loop

In Table 2 and 3 we see much greater speed improvements especially as the grid size gets larger, our design choices have led to speeds up to four times faster for computing the ADI loop. This is especially useful as larger grids have more accurate approximations which is investigated later. This justifies some of the critical ideas that we laid out.

4.2 Global Errors

An important and popular way to measure convergence for finite difference schemes is to consider global errors, see Magali [37] or Hout and Wyns [38]. This is more robust as previously we only considered one point on the surface by interpolation. The idea is to plot the maximum absolute error over a region of our grid, usually a region of interest, that is compared to the semi-analytical solution of the Heston PDE in Section 2.6. To avoid using interpolation the exact grid points in our mesh are considered. For the remainder of this section we consider the following cases:

	Case 1	Case 2	Case 3	Case 4
κ	1.50	0.30	0.38	0.30
θ	0.04	0.06	0.09	0.06
σ	0.3	0.15	1.26	0.15
ρ	-0.9	0.78	-0.55	0.78
r	0.025	0.01	0.01	0.01
T	1	2	4	5

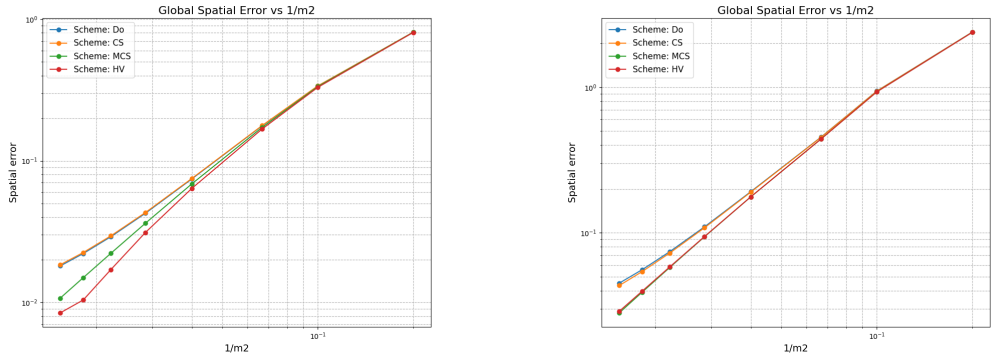
Table 4: Parameter values for different cases.

4.2.1 Global Spatial Discretisation Error

We first investigate the spatial error for different sizes of the mesh, noting that we keep the ratio of grid points in the S direction double that of the v direction $m_1 = 2m_2$. This is an efficient heuristic choice considered and works well [17]. Formally, we define *global spatial discretisation error* at $t = T$ as

$$e(m_1, m_2) = \max\left\{|u(S_i, v_j, T) - U_k(T)| : \frac{1}{2}K < S_i \leq \frac{3}{2}K,\, 0 < v_j < 1\right\}. \tag{4.1}$$

where u is the exact European call option price, Section 2.6, over an unbounded domain while U is our finite difference approximation both considered for a grid point (S_i, v_j) over a predefined region of interest related to the strike price and initial variance. We aim to keep the influence of temporal error low by having the number of time steps as $T = 500$. We choose $\theta = 0.8$ for all schemes with a strike $K = 100$ to obtain the following results



(a) Case 1 (b) Case 2

Figure 7: Global Spatial Discretisation Error

The results in Figure 7 are pleasant as we see for each scheme a more fine grid results in more accurate approximations. We also notice that the Hunsdorfer-Verwer and Modified Craig-Sneyd schemes have a faster rate of convergence, for $\theta = 0.8$ they exhibit second order accuracy while the Douglas and Craig-Sneyd schemes do not[6], however temporal errors are preferred for this analysis. Furthermore, the accuracy results align with the literature to show a monotonic decay behaviour, which again confirms our implementation is working correctly.

4.2.2 Global Temporal Discretisation Error & Damping

In this section we would like to start off by investigating the errors related only due to the size of the time step which is used in the ADI schemes. This will give us an insight into the stability and convergence behaviour and whether any procedures can improve them. As [17], we define *global temporal discretisation error* at maturity as

$$\hat{e}(N; m_1, m_2) = \max\left\{ |U_k(T) - U_{N,k}| : \frac{1}{2}K < s_i < \frac{3}{2}K,\ 0 < v_j < 1 \right\}. \tag{4.2}$$

where $U_k(T)$ instead will be the 'true' value of the semi-discrete system which we obtain, motivated by our experiment in 4.2.1, using the Hundsdorfer-Verwer scheme with number of time steps $N = 2000$ and $\theta = \frac{1}{2} + \frac{1}{6}\sqrt{3}$. Then $U_{N,k}$ are the schemes we will be comparing for time discretisation errors. Notice that we are not comparing against the closed-form Heston PDE solution.

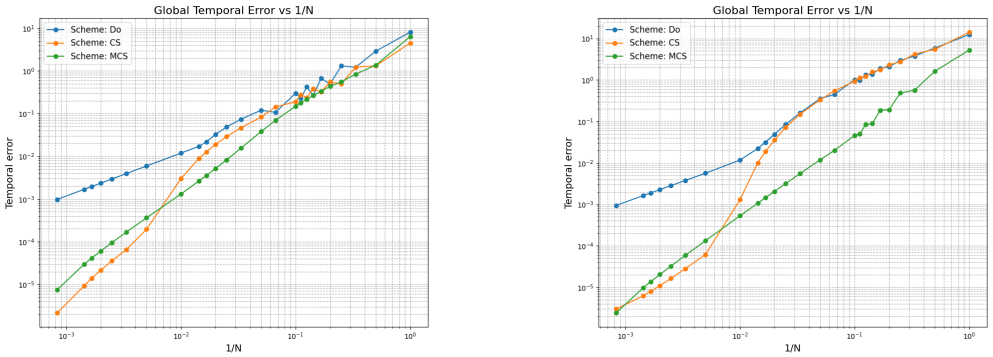
For stability the following values based on literature, where for example values of $\theta < \frac{1}{2}$ break down for Douglas, are chosen.

[6]There are theoretical proofs for their convergence behaviour, see Strikwerda [39].

- (Do) Douglas with $\theta = \frac{1}{2}$
- (CS) Craig-Sneyd with $\theta = \frac{1}{2}$
- (MCS) Modified Craig-Sneyd with $\theta = \frac{1}{3}$

We will now consider Case 3 and 4, with the same grid ratio $m_1 = 2m_2$ and a strike price $K = 100$. Starting with a grid size of $m_1 = 100$ in Figure 8, we see a monotonic decay for each scheme which is analogous to that in the literature. This behaviour is favourable as we can increase the number of time steps to get a lower error. Furthermore, the errors are small, even for only one time step, $\frac{1}{N} = 10^0$, the error is reasonable.

For the moment, lets look at the portion of the graphs left of the point where $\frac{1}{N} = 10^{-2}$, there is a noticeable difference in the slopes between the Douglas scheme and the two other schemes. The gradient of the Douglas scheme is around 1, as the number of time steps increases by a factor 10, from $\frac{1}{N} = 10^{-2}$ to 10^{-3} the error decreases by a factor 10 as well. However, the Modified Craig-Sneyd and Craig-Sneyd schemes have a faster rate of convergence with their gradient being around 2. This is related to their order of accuracy, it has been shown theoretically, see specifically [40] or again [39], with the θ parameters selected that the Douglas scheme is of order 1 while Modified Craig-Sneyd and Craig-Sneyd are of order 2.



(a) Case 3 with $m_1 = 100$

(b) Case 4 with $m_1 = 100$

Figure 8: Global Temporal Discretisation Error

If we instead look to the portion of the graph right at point $\frac{1}{N} = 10^{-2}$, with a smaller number of time steps the Douglas and Craig-Sneyd schemes experience strange behaviour. Their errors are larger relative to their asymptotic behaviour on the other side. We investigate this further and examine the effects of increasing the grid size to $m_1 = 200$ and $m_2 = 100$ to see in Figure 9 this behaviour is exacerbated for Douglas and Craig-Sneyd while Modified Craig-Sneyd remains largely unaffected.

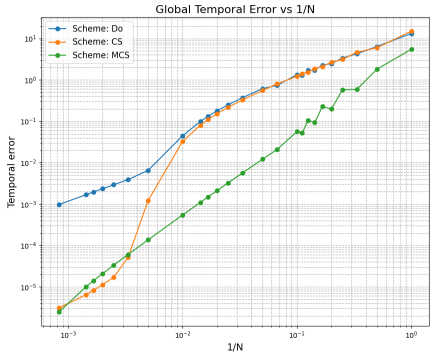
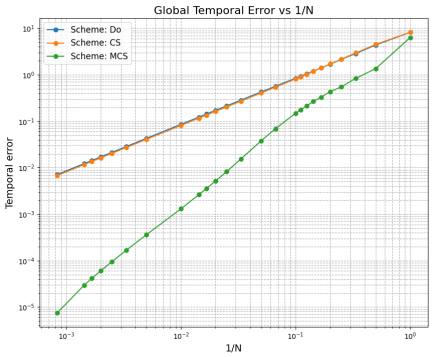


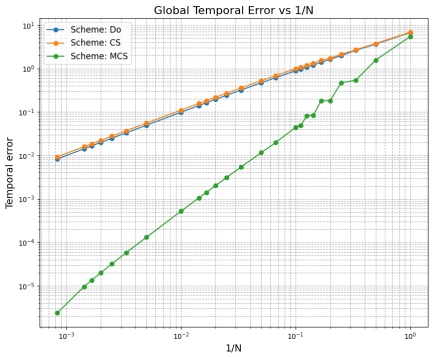
Figure 9: Case 4 with $m_1 = 200$

Hout and Foulon [17] mention these errors are due to the non-smooth initial function where Douglas and Craig-Sneyd don't have any built in damping procedures allowing errors introduced to persist for longer. In figure 9 with a larger grid, not only are the errors amplified, but they also occur for a larger number of time steps. However on a positive note, these experiments show that the three schemes show unconditionally stable behaviour, increasing the number of grid points hasn't caused any monumental errors that are sometimes observed with other time stepping schemes applied to Heston type PDEs, see Runge-Kutta-Chebyshev methods in [41].

Now to combat these convergence issues seen with a small amount of time steps, [42] recommend a popular choice to use Rannacher time stepping as a damping procedure. This works by employing two backward Euler steps at $t = 0$ with time step size $\Delta\frac{t}{2}$. We do this for the Douglas and Craig-Sneyd schemes to obtain the following results in Figure 10. There is a partial success, the Douglas scheme has successfully been damped and still retains its order of 1, although the Craig-Sneyd scheme has this same order and behaviour. It has been damped however its order should be consistent with that of the Modified Craig-Sneyd, this failure needs to be investigated further.



(a) Case 3, $m_1 = 100$

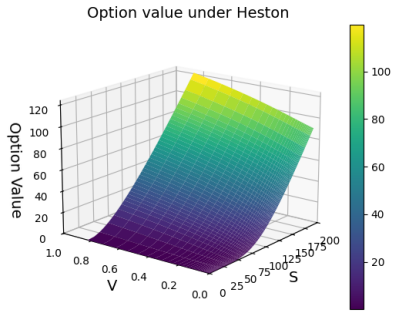


(b) Case 4 with $m_1 = 100$

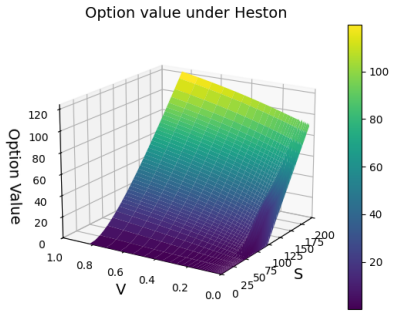
Figure 10: Global Temporal Discretisation Error with Damping Applied

4.3 Upwinding

In this section we will show the importance of applying appropriate upwinding(or one-sided difference) schemes for approximating derivatives on our grid. Previously, in Section 3.1, we mentioned that the Heston PDE becomes convection dominated in the v direction if σ is small. Using the central scheme for the first order v derivatives can lead to instabilities so we make use of the forward scheme at $v = 0$.



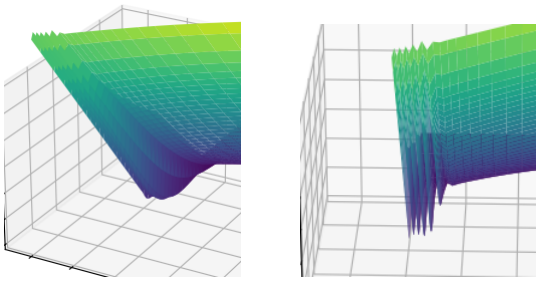
(a) Price surface with correct upwinding schemes applied



(b) Price surface with central scheme applied at $v = 0$

Figure 11: Oscillations

In Figure 11 we have in both cases used the parameter set Case 3 but with $\sigma = 0.0005$. Note this phenomenon occurs even for larger, more realistic, values of σ (e.g., $\sigma = 0.05$). In Figure 11 (b) the central scheme has been applied at $v = 0$ while in (a) the forward scheme, as suggested by [19], is used. In the close up from Figure 12 the instabilities are



(a) Price surface with central scheme applied at $v = 0$

(b) Price surface with central scheme applied at $v = 0$

Figure 12: Oscillations zoomed in

more clear. With a more coarse grid the oscillations persist further and affect regions deeper in the grid, so it is clear that Kluge's recommendation for the forward scheme is essential, especially when σ is small.

4.4 Feller Condition

As mentioned before, when the Feller condition is not satisfied the variance process is able to yield negative values. We want to check whether this influences our global spatial error as for Monte-Carlo simulations it can reduce accuracy. Below are the results using Case 3, noticing that Feller is not satisfied $\frac{2\kappa\theta}{\sigma^2} = 0.029 \not> 1$:

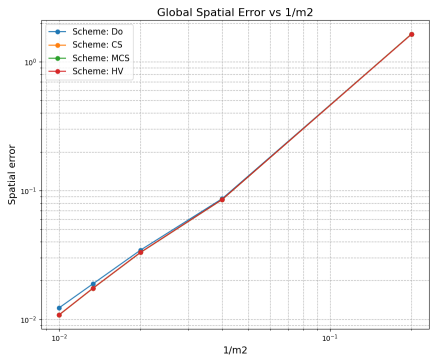
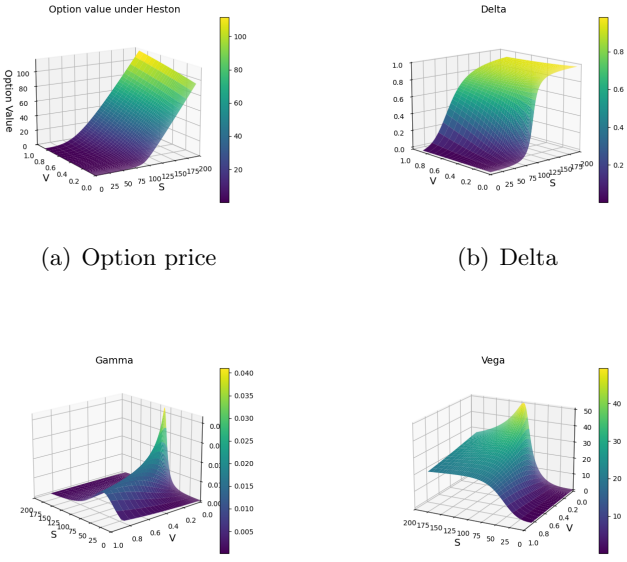


Figure 13: Global Spatial Error for Feller condition

In this case the Craig-Sneyd scheme instead performs the best, just under Hunsdorfer-Verwer although we see the same behaviour and accuracy as in Section 4.2.1. There is likely only some small influence of the Feller condition not being satisfied. The surface plots with $m_1 = 150$ of this experiment are shown in Figure 14.



(a) Option price

(b) Delta

(c) Gamma

(d) Vega

Figure 14: Surface plots from Case 3

5 Legal, Social, Ethical and Professional Issues

In this work, the Code of Conduct, published by the British Computer Society (BCS) [43] is to the best of my ability strictly adhered to. In this section we will shortly outline how we have followed their four key principles:

5.1 "You make IT for everyone"

The key theme of this project has been to give implementation specific insights for finite difference methods, without which they could become rather complicated. This drastically increases the accessibility of this work, especially with the provided code. Due regard has been kept for legitimate rights of third parties, where relevant sources have been acknowledged appropriately. Furthermore, the program makes use of open-source libraries so our work can be simply reproduced in a non-commercial environment. These tie into legal issues where we have acted in accordance with relevant laws and regulations.

5.2 "Show what you know, learn what you don't"

I initially planned for a rather broad scoping project, however with advice from my supervisor, Dr Riaz Ahmad, the choice was made to focus on a specific subset of finite difference methods for a single stochastic volatility model. This could have otherwise led to a project not being completed within the given time frame. This is a professional

issue that became resolved, now allowing for collaboration with other researchers in the field.

5.3 "Respect the organisation or individual you work for"

This principle isn't as relevant for us, there have been no cases for cause of conflict with others as this is an individual endeavour. However, ethical standards have been kept in terms of citing other researchers literature.

5.4 Keep IT real. Keep IT professional. Pass IT on.

Professional standards have been followed in the creation of this project and its subsequent program, which allows for the IT to be passed on as mentioned before. A result of this is that our work, to some extent, could have an impact on financial markets as well as their participants and regulators. This social issue could encourage others to become more informed about markets and introduce efficiency.

6 Conclusion

In this dissertation, we have applied the finite difference method to approximate derivatives in the Heston PDE, allowing us to solve for the case of European options. The focus has been on efficiency, where we made choices along the way based on previous literature. The most important of which being ADI schemes, which combine the best of both implicit and explicit schemes.

We first began with a short motivation on stochastic volatility and then jumped into our background theories on the Heston model, introducing the PDE as well as the closed form solution.

For the finite difference approximation, we introduce many standard concepts but then show how the, not so common, Kronecker product can be used for construction of the A matrix, this proved to be far more efficient as shown in our results section. Along with that, we showed why LU decomposition techniques, as opposed to matrix inversion, are the way to go for solving linear systems.

To get better ideas about accuracy we introduced global spatial errors, the results here aligned with the literature. Furthermore, when checking global temporal errors, we were able to see the orders of each scheme by measuring their slope, their orders aligned with the theoretical ones based on their θ values. We attempted a damping procedure which saw positive results for the Douglas scheme, however for Craig—Sneyd the order seemed to be damaged, in the future we should try different types of damping schemes.

Finally, we presented experiments on the Feller condition, and the importance of upwinding schemes. Additionally, the Greeks have been introduced, which are simple to compute via finite difference techniques.

This dissertation has solely focused on the European call option, it can be fairly easily extended to Barrier options as the payoff function is similar however it could be more interesting for Asian options. In that case the PDE would have an extra auxiliary variable for the running average of the options price. Another area of interest is exploring different types of multi—dimensional PDEs. Models such as the 3/2 model seems as a natural extension due to its similarity to Heston.

References

[1] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[2] S. L. Heston, "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options," *The Review of Financial Studies*, vol. 6, pp. 327–343, 04 2015.

[3] S. S. Mikhailov and U. Nögel, "Heston's Stochastic Volatility Model Implementation, Calibration and Some," *Wilmott Magazine*, 2003.

[4] International Swaps and Derivatives Association (ISDA), "Key Trends in the Size and Composition of OTC Derivatives Markets in the Second Half of 2023," tech. rep., International Swaps and Derivatives Association (ISDA), 2023.

[5] C. F. Loan, "The ubiquitous Kronecker product," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1, pp. 85–100, 2000. Numerical Analysis 2000. Vol. III: Linear Algebra.

[6] K. in't Hout, "Interest Rate Modelling: ADI Schemes for Pricing Options under the Heston model." Workshop presented at Quants Hub, 2018. Accessed: 2024-06-10.

[7] J. Anderson, *Computational Fluid Dynamics*. Computational Fluid Dynamics: The Basics with Applications, McGraw-Hill Education, 1995.

[8] D. Tavella and C. Randall, *Pricing Financial Instruments: The Finite Difference Method*. Wiley series in financial engineering, John Wiley & Sons, 2000.

[9] J. C. Cox, J. E. Ingersoll, and S. A. Ross, "A Theory of the Term Structure of Interest Rates," *Econometrica*, vol. 53, no. 2, pp. 385–407, 1985.

[10] F. D. Rouah, *The Heston Model and Its Extensions in Matlab and C#*. John Wiley & Sons, Ltd, 2013.

[11] W. Bernard and C. Heyde, "On changes of measure in stochastic volatility models," *Journal of Applied Mathematics and Stochastic Analysis*, 2006.

[12] J. Gil-Pelaez, "Note on the inversion theorem," *Biometrika*, vol. 38, pp. 481–482, 12 1951.

[13] J. Emerick, "Heston Model Calibration to Option Prices." `https://quantpy.com.au/stochastic-volatility-models/heston-model-calibration-to-option-prices/`, 2023. Accessed: 2024-07-15.

[14] L. Ballabio, *Implementing QuantLib: Quantitative Finance in C++: an Inside Look at the Architecture of the QuantLib Library*. Amazon Digital Services LLC - KDP Print US, 2020.

[15] T. Haentjens, "Efficient and stable numerical solution of the Heston–Cox–Ingersoll–Ross partial differential equation by alternating direction implicit finite difference schemes," *International Journal of Computer Mathematics*, vol. 90, no. 11, pp. 2409–2430, 2013.

[16] W. Hundsdorfer and J. Verwer, *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*, vol. 33 of *Springer Series in Computational Mathematics*. Berlin, Heidelberg: Springer, 2003.

[17] K. J. in 't Hout and S. Foulon, "ADI finite difference schemes for option pricing in the Heston model with correlation," 2011.

[18] T. Haentjens and K. J. in 't Hout, "ADI Schemes for Pricing American Options under the Heston Model," *Applied Mathematical Finance*, vol. 22, p. 207–237, Feb. 2015.

[19] T. Kluge, "Pricing derivatives in stochastic volatility models using the finite difference method," Master's thesis, Chemnitz University of Technology, Faculty of Mathematics, 2002. Accessed: 2024-07-05.

[20] A. Clevenhaus, M. Ehrhardt, and M. Günther, "An ADI Sparse Grid Method for Pricing Efficiently American Options under the Heston Model," *Advances in Applied Mathematics and Mechanics*, 200x. Accessed: 2024-08-15.

[21] E. Ngounda, K. Patidar, E. Pindza, "A Robust Spectral Method for Solving Heston's Model," *Journal of Optimization Theory and Applications*, vol. 161, 04 2014.

[22] P. Wilmott, *Paul Wilmott on Quantitative Finance*. John Wiley & Sons, 2010.

[23] G. G. Lazareva, I. P. Oksogoeva, and A. V. Sudnikov, "Mathematical model of matter transfer in a helical magnetic field using boundary conditions at infinity," *Contemporary Mathematics. Fundamental Directions*, vol. 69, pp. 418–429, 10 2023.

[24] E. Ekström and J. Tysk, "The Black–Scholes equation in stochastic volatility models," *Journal of Mathematical Analysis and Applications*, vol. 368, no. 2, pp. 498–507, 2010.

[25] A. Hirsa, *Computational methods in finance*. Boca Raton, FL: CRC Press, 2013.

[26] C. O'Sullivan and S. O'Sullivan, "Pricing European and American Options in the Heston Model with Accelerated Explicit Finite Difference Methods," *International Journal of Theoretical and Applied Finance*, vol. 16, no. 03, 2013.

[27] M. T. Heath, *Scientific Computing: An Introductory Survey*, ch. Chapter 11: Partial Differential Equations, pp. 446–493. Soceity for Industrial and Applied Mathematics, 1996.

[28] M. Wyns and K. in 't Hout, "An adjoint method for the exact calibration of Stochastic Local Volatility models," 2016.

[29] C. S. L. de Graaf, "Finite Difference Methods in Derivatives Pricing under Stochastic Volatility Models," Master's thesis, Mathematisch Instituut, Universiteit Leiden, September 2012. Master's thesis, Specialisation: Applied Mathematics, Thesis advisor: B. Koren.

[30] L. Calatroni, C. Estatico, N. Garibaldi, and S. Parisotto, "Alternating Direction Implicit (ADI) schemes for a PDE-based image osmosis model," *Journal of Physics: Conference Series*, vol. 904, p. 012014, Oct. 2017.

[31] M. Khasi and J. Rashidinia, "A Bilinear Pseudo-spectral Method for Solving Two-asset European and American Pricing Options," *Computational Economics*, vol. 63, no. 2, pp. 893–918, 2024.

[32] K. J. in 't Hout and K. Volders, "Stability of central finite difference schemes for the Heston PDE," *Numerical Algorithms*, vol. 60, no. 1, pp. 115–133, 2012.

[33] D. W. Peaceman and H. H. Rachford, "The Numerical Solution of Parabolic and Elliptic Differential Equations," *Journal of the Society for Industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, 1955.

[34] J. Douglas and H. H. Rachford, "On the Numerical Solution of Heat Conduction Problems in Two and Three Space Variables," *Transactions of the American Mathematical Society*, vol. 82, pp. 421–439, 1956.

[35] M. West, "Lu decomposition for solving linear equations." `https://courses.physics.illinois.edu/cs357/sp2020/notes/ref-9-linsys.html`, 2018. Accessed: 2024-07-3.

[36] redbzi, "NM-Heston." Github Repository. Available at: `https://github.com/redbzi/NM-Heston`, 2017. Accessed: 2024-07-4.

[37] M. Calabressi, "Alternating Direction Implicit Methods for Solutions of the Heston Stochastic Volatility Model," Master's thesis, University College London, September 2014.

[38] K. in 't Hout and M. Wyns, "Convergence of the Modified Craig-Sneyd scheme for two-dimensional convection-diffusion equations with mixed derivative term," 2014.

[39] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations, Second Edition*. Soceity for Industrial and Applied Mathematics, 2004.

[40] K. in 't Hout and B. Welfert, "Unconditional stability of second-order ADI schemes applied to multi-dimensional diffusion equations with mixed derivative terms," *Applied Numerical Mathematics*, vol. 59, no. 3, pp. 677–692, 2009. Selected Papers from NUMDIFF-11.

[41] F. L. Floc'h, "Instabilities of Super-Time-Stepping Methods on the Heston Stochastic Volatility Model," 2023.

[42] D. M. Pooley, K. R. Vetzal, and P. A. Forsyth, "Convergence remedies for non-smooth payoffs in option pricing," *Journal of Computational Finance*, vol. 6, pp. 25–40, 2003.

[43] BCS, The Chartered Institute for IT, "BCS Code of Conduct." Available at: `https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/`, 2024. Accessed: 2024-08-4.

A Appendix

A.1 Making the non-uniform mesh

Listing 7: Code for Non-Uniform grids

```
import math
import numpy as np
import scipy.sparse as sp


def make_grid(T, K, S_max, c, M1, V_max, M2):


    negative_discount = math.exp(-0.1 * T)
    positive_discount = math.exp(0.1 * T)

    S_lower_uniform = max(1/2, negative_discount) * K

    S_upper_uniform = max(1/2, positive_discount) * K


    xi_min = np.arcsinh(-S_lower_uniform / c)


    xi_interval = (S_upper_uniform - S_lower_uniform) / c


    xi_max = xi_interval + np.arcsinh((S_max - S_upper_uniform) / c)


    xi = np.linspace(xi_min, xi_max, M1)


    # xi in segments
    xi_first = [val for val in xi if val <= 0]
    xi_second = [val for val in xi if 0 < val < xi_interval]
    xi_third = [val for val in xi if val >= xi_interval]

    xi_first = np.array(xi_first)
    xi_second = np.array(xi_second)
    xi_third = np.array(xi_third)


    # s segments
    s_first = S_lower_uniform + c * np.sinh(xi_first)

    s_second = S_lower_uniform + c * xi_second
```

```
s_third = S_upper_uniform + c * np.sinh(xi_third - xi_interval)


# Combine segments
s = np.concatenate([s_first, s_second, s_third])


# v - direction

d = (V_max / 500)

psi_max = np.arcsinh(V_max / d)

psi = np.linspace(0, psi_max, M2)

v = d * np.sinh(psi)
v[v < 0] = 0



# Where s less than 2K
index_left_plot = np.where(s < 2 * K)[0][-1]

# Where v less than 1
index_v_l1 = np.where(v < 1)[0][-1]

index_s_boundary = np.arange(1, M1)
index_v_boundary = np.arange(M2 - 1)

index_s_boundary_size = index_s_boundary.size
index_v_boundary_size = index_v_boundary.size

# Create sparse diagonal matrices
X_tilde = sp.diags(s, 0, shape=(M1, M1))
Y_tilde = sp.diags(v, 0, shape=(M2, M2))

# Extract submatrices by converting to dense format
X_dense = X_tilde.toarray()
Y_dense = Y_tilde.toarray()

X = X_dense[index_s_boundary, :][:, index_s_boundary]
Y = Y_dense[index_v_boundary, :][:, index_v_boundary]

# Create grid for S and V
S, V = np.meshgrid(s, v, indexing='ij')

return index_left_plot, index_v_l1, index_s_boundary,   \
```

```
        index_v_boundary, index_s_boundary_size,            \
        index_v_boundary_size, X_tilde, Y_tilde, X, Y
```

A.2 The ADI Functions

Listing 8: Hundsdorfer Verwer

```
def MCS(u, dt, LU1, LU2, A, A_0, prev,                \
    prev_0, prev_1, prev_2, next, next_0, next_1, next_2, theta):

    # Predictor step
    z0 = dt * (A @ u + prev)

    # Two correction steps
    z = LU1.solve(z0 + (theta * dt) * (next_1 - prev_1))
    z = LU2.solve(z + (theta * dt) * (next_2 - prev_2))

    # Predictor step
    z = z0 + (theta * dt) * (A_0 @ z + next_0 - prev_0)     \
        + ((1 / 2 - theta) * dt) * (A @ z + next - prev)

    # Two correction steps
    z = LU1.solve(z + (theta * dt) * (next_1 - prev_1))
    z = LU2.solve(z + (theta * dt) * (next_2 - prev_2))

    u = u + z

    return u

def CS(u, dt, LU1, LU2, A, A_0, prev,                 \
    prev_0, prev_1, prev_2, next, next_0, next_1, next_2, theta):

    # Predictor step
    z0 = dt * (A @ u + prev)  # Y0 = Un-1 + dt * F(tn-1, Un-1)

    # Two corrector steps
    z = LU1.solve(z0 + (theta * dt) * (next_1 - prev_1))# Y1
    z = LU2.solve(z + (theta * dt) * (next_2 - prev_2))# Y2

    # Correction for mixed derivative term
    z_hat = z0 + 0.5 * dt * (A_0 @ z + next_0 - prev_0)

    # Two corrector steps
    z_tilde = LU1.solve(z_hat + (theta * dt) * (next_1 - prev_1))
    z_tilde = LU2.solve(z_tilde + (theta * dt) * (next_2 - prev_2))

    u = u + z_tilde
```

```
    return u


def Do(u, dt, LU1, LU2, A, prev,                \
    prev_1, next_1, prev_2, next_2, theta):

    # Explicit Predictor stage
    z0 = dt * (A @ u + prev)

    # Two Implicit Corrector stages
    z = LU1.solve(z0 + theta * dt * (next_1 - prev_1))
    z = LU2.solve(z + theta * dt * (next_2 - prev_2))

    u = u + z

    return u


def HV(u, dt, LU1, LU2, A, A_0, prev,           \
    prev_0, prev_1, prev_2, next, next_0, next_1, next_2, theta):

    # Step 1: Initial explicit step
    Y0 = u + dt * (A @ u + prev)

    # Step 2: Solving stages for Yj
    Z1 = Y0 - u
    Y1 = Z1 + theta * dt * (next_1 - prev_1)
    Y1 = LU1.solve(Y1)
    Y1 = LU2.solve(Y1)
    Y1 = Y1 + u

    Z2 = Y1 - u
    Y2 = Z2 + theta * dt * (next_2 - prev_2)
    Y2 = LU1.solve(Y2)
    Y2 = LU2.solve(Y2)
    Y2 = Y2 + u

    # Step 3: Corrector step
    Y0_hat = Y0 + (0.5 * dt) * (next_0 - prev_0)

    # Step 4: Solving stages for hat{Y}_j
    Z1_hat = Y0_hat - u
    hat_Y1 = Z1_hat + theta * dt * (next_1 - prev_1)
    hat_Y1 = LU1.solve(hat_Y1)
    hat_Y1 = LU2.solve(hat_Y1)
    hat_Y1 = hat_Y1 + u
```

```
    Z2_hat = hat_Y1 - Y2
    hat_Y2 = Z2_hat + theta * dt * (next_2 - prev_2)
    hat_Y2 = LU1.solve(hat_Y2)
    hat_Y2 = LU2.solve(hat_Y2)
    hat_Y2 = hat_Y2 + Y2

    # Step 5: Final update
    u = hat_Y2

    return u
```

A.3 Heston Closed-Form Solution Function

Listing 9: Heston's closed form solution via QuantLib

```
import QuantLib as ql

def heston_price(spot, strike, maturity, risk_free_rate,  \
 dividend_yield, v0, kappa, theta, sigma, rho):
    """ Heston price via QuantLib """


    spot_handle = ql.QuoteHandle(ql.SimpleQuote(spot))
    maturity_date = ql.Date().todaysDate()          \
    + ql.Period(int(maturity * 365), ql.Days)
    risk_free_rate_handle = ql.YieldTermStructureHandle(
        ql.FlatForward(0, ql.NullCalendar(),
        ql.QuoteHandle(ql.SimpleQuote(risk_free_rate)),
        ql.Actual365Fixed()))
    dividend_yield_handle = ql.YieldTermStructureHandle(
        ql.FlatForward(0, ql.NullCalendar(),
        ql.QuoteHandle(ql.SimpleQuote(dividend_yield)),
        ql.Actual365Fixed()))


    heston_process = ql.HestonProcess(risk_free_rate_handle, \
        dividend_yield_handle, spot_handle, v0, kappa, theta,
                                      sigma, rho)


    heston_model = ql.HestonModel(heston_process)


    engine = ql.AnalyticHestonEngine(heston_model)


    european_option = ql.EuropeanOption(ql.PlainVanillaPayoff  \
        (ql.Option.Call, strike),                              \
```

```
        ql.EuropeanExercise(maturity_date))
    european_option.setPricingEngine(engine)


    heston_price = european_option.NPV()

    return heston_price
```

A.4 Global Discretisation Error Functions

Listing 10: Global errors functions

```
def compute_global_spatial_error(m1_values, kappa, eta, sigma, rho
    , rd, rf, T, K, scheme):
    errors = []

    for m1 in m1_values:



        U, S, V = FDMHeston(kappa=kappa, eta=eta, sigma=sigma, rho
            =rho, rd=rd, rf=rf, T=T, K=K, m1=m1, scheme=scheme, N
            =500, theta_exp=0.0)
        max_error = 0

        for i in range(U.shape[0]):
            for j in range(U.shape[1]):
                if 0 < V[i, j] < 1 and (0.5 * K) < S[i, j] < (1.5
                    * K):
                    analytical_price = heston_price(S[i, j], K, T,
                        rd, rf, V[i, j], kappa, eta, sigma, rho)
                    max_error = max(max_error, abs(U[i, j] -
                        analytical_price))
                    print("MAX␣ERROR␣:", max_error, "At␣Point␣S=",
                        S[i, j], "At␣Point␣V=", V[i, j], "Price␣
                        FDM␣␣=␣", U[i,j], "Price␣Analytical␣=",
                        analytical_price)


        errors.append(max_error)

    return errors



# Params
```

```
#case 3
kappa = 0.38
eta = 0.09
sigma = 1.26
rho = -0.55
rd = 0.01
rf = 0.0
T = 4
K = 100

#case 4
#kappa = 0.3
#eta = 0.06
#sigma = 0.15
#rho = 0.78
#rd = 0.01
#rf = 0.0
#T = 5
#K = 100

#Case 2
#kappa = 0.3
#eta = 0.06
#sigma = 0.15
#rho = 0.78
#rd = 0.01
#rf = 0.0
#T = 2
#K = 100



m1_values = [10, 50, 100, 150, 200]


schemes = ['Do', 'CS', 'MCS', 'HV']

errors_dict = {}


plot_spatial = False

if plot_spatial:

    for scheme in schemes:
        errors = compute_global_spatial_error(m1_values, kappa,
           eta, sigma, rho, rd, rf, T, K, scheme)
        errors_dict[scheme] = errors
```

```
    m2_values = [m1 / 2 for m1 in m1_values]

    # Plotting
    plt.figure(figsize=(10, 8))

    for scheme in schemes:
        plt.loglog([1 / m2 for m2 in m2_values], errors_dict[
            scheme], marker='o', label=f'Scheme: {scheme}')

    plt.xlabel('1/m2', fontsize=14)
    plt.ylabel('Spatial error', fontsize=14)
    plt.title('Global Spatial Error vs 1/m2', fontsize=16)
    plt.grid(True, which="both", ls="--")
    plt.legend(fontsize=12)
    plt.show()


"""
# Spatial error plotting - Single scheme

errors = compute_global_spatial_error(m1_values, kappa, eta, sigma
    , rho, rd, rf, T, K)
m2_values = [m1 / 2 for m1 in m1_values]

# Plotting
plt.figure(figsize=(8, 6))
plt.loglog([1/m2 for m2 in m2_values], errors, 'bo-')
plt.xlabel('1/m2', fontsize=14)
plt.ylabel('Spatial error', fontsize=14)
plt.title('Global Spatial Error vs 1/m2', fontsize=16)
plt.grid(True, which="both", ls="--")
plt.show()
"""



# Temporal error
def compute_global_temporal_error(N_values, kappa, eta, sigma, rho
    , rd, rf, T, K, scheme):
    errors = []

    #U_acc, S_acc, V_acc = FDMHeston(kappa=kappa, eta=eta, sigma=
        sigma, rho=rho, rd=rd, rf=rf, T=T, K=K, m1=200 , N=5000,
        scheme='Do')

    U_acc, S_acc, V_acc = FDMHeston(kappa=kappa, eta=eta, sigma=
        sigma, rho=rho, rd=rd, rf=rf, T=T, K=K, N=2000, scheme='MCS
        ')
```

```
    for N in N_values:

        U, S, V = FDMHeston(kappa=kappa, eta=eta, sigma=sigma, rho
            =rho, rd=rd, rf=rf, T=T, K=K, N=N, scheme=scheme)
        max_error = 0




        for i in range(U.shape[0]):
            for j in range(U.shape[1]):
                if 0.1 < V[i, j] < 1.0 and (0.5 * K) < S[i, j] <
                    (1.5 * K):

                    #acc_price = heston_price(S[i, j], K, T, rd,
                        rf, V[i, j], kappa, eta, sigma, rho)

                    acc_price = U_acc[i, j]



                    max_error = max(max_error, abs(acc_price - U[i
                        , j] ))
                    print("MAX␣ERROR␣:", max_error, "At␣Point␣S=",
                         S[i, j], "At␣Point␣V=", V[i, j], "Price␣
                        FDM␣␣=␣", U[i,j], "Price␣Analytical␣=",
                        acc_price)
                    print("N:␣", N)



        errors.append(max_error)

    return errors


plot_temporal = False

if plot_temporal:

    schemes = ['Do', 'CS', 'MCS']
    errors_dict = {}


    N_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50,
        60, 70 ,100, 200, 300, 400, 500, 600, 700, 1200]
```

```
    for scheme in schemes:
        print("Scheme␣(First␣loop␣computing):", scheme)

        errors = compute_global_temporal_error(N_values, kappa,
            eta, sigma, rho, rd, rf, T, K, scheme)
        errors_dict[scheme] = errors

    oneOverN = [1 / N for N in N_values]

    # Plotting
    plt.figure(figsize=(10, 8))

    for scheme in schemes:
        print("Scheme␣(Second␣loop␣plotting):", scheme)

        plt.loglog(oneOverN, errors_dict[scheme], marker='o',
            label=f'Scheme:␣{scheme}')

    plt.xlabel('1/N', fontsize=14)
    plt.ylabel('Temporal␣error', fontsize=14)
    plt.title('Global␣Temporal␣Error␣vs␣1/N', fontsize=16)
    plt.grid(True, which="both", ls="--")
    plt.legend(fontsize=12)
    plt.show()
```

A.5 Putting Everything Together

Listing 11: Computing the Matrices and Running the code all together

```
def FDMHeston(kappa = 0.9, eta = 0.04, sigma = 0.3, rho = 0.4, rd
   = 0.025, rf = 0.0, T = 1, K = 100, m1=150, N=500,
              scheme = 'HV', theta_exp=0.0, S_max_multiplier = 30,
                  V_max_multiplier = 15):

    S_max = K * S_max_multiplier
    V_max = 1 * V_max_multiplier

    m2 = round(m1 * 0.5)

    S, V, s, v, index_left_plot, index_v_l1, index_s_boundary,
        index_v_boundary, index_s_boundary_size, \
    index_v_boundary_size, X_tilde, Y_tilde, X_diags, Y_diags =
        grid.make_grid(T, K, S_max, c, m1_tilde, V_max, m2_tilde)

    # Find  where s > K
    index_sK = np.argmax(s > K)
```

```
right = s[index_sK] - K
left = s[index_sK - 1] - K

index_sK -= np.abs(right) > np.abs(left)

# s_left and s_right
s_left = (s[index_sK - 1] + s[index_sK]) / 2
s_right = (s[index_sK] + s[index_sK + 1]) / 2

# Update u at the index index_sK
u[index_sK, :] = (1/2) * pow(s_right - K, 2) / (s_right -
    s_left)

# Extract the submatrix)
u_sub = u[np.ix_(index_s_boundary, index_v_boundary)]

# Vectorise
u_flat = u_sub.ravel(order='F')

u = u_flat

# Gamma_tilde matrix
Gamma_tilde = np.full((m1_tilde, m2_tilde), 0)

Gamma_tilde[:, - 1] = (s)

start2 = datetime.now()

# Discretisation Matrices - Starting with Coefficients
offsets_central = [-1, 0, 1]
offsets_backward = [-2, -1, 0]
# Successive mesh widths in s-direction, and change in mesh
    widths
Delta_s = np.diff(s[:m1_tilde])
Delta_s_i = Delta_s[:-1]
Delta_s_i_1 = Delta_s[1:]

# Betas in s-direction
beta_minus_s = -Delta_s_i_1 / (Delta_s_i * (Delta_s_i +
    Delta_s_i_1))
beta_plus_s = Delta_s_i / (Delta_s_i_1 * (Delta_s_i +
    Delta_s_i_1))
beta_zero_s = -(beta_minus_s + beta_plus_s)

# detlas in s direction (2nd derivative)
delta_minus_s = 2 / (Delta_s_i * (Delta_s_i + Delta_s_i_1))
delta_plus_s = 2 / (Delta_s_i_1 * (Delta_s_i + Delta_s_i_1))
delta_zero_s = -(delta_minus_s + delta_plus_s)
```

```
# Pad the arrays to work in sparse matrix format
beta_minus_s = np.append(beta_minus_s, [0, 0])
beta_zero_s = np.concatenate(([0], beta_zero_s, [0]))
beta_plus_s = np.concatenate(([0], beta_plus_s))

delta_minus_s = np.append(delta_minus_s, [0, 0])
delta_zero_s = np.concatenate(([0], delta_zero_s, [0]))
delta_plus_s = np.concatenate(([0], delta_plus_s))


# Now same thing but in v direction
# Successive mesh widths in v-direction
Delta_v = np.diff(v[:m2_tilde])
Delta_v_i = Delta_v[:-1]
Delta_v_i_1 = Delta_v[1:]

# Beta's for v direction
beta_minus_v = -Delta_v_i_1 / (Delta_v_i * (Delta_v_i +
   Delta_v_i_1))
beta_plus_v = Delta_v_i / (Delta_v_i_1 * (Delta_v_i +
   Delta_v_i_1))
beta_zero_v = -(beta_minus_v + beta_plus_v)

# Alpha's for v direction
alpha_minus_2_v = -(beta_minus_v)
alpha_minus_v = -(Delta_v_i + Delta_v_i_1) / (Delta_v_i *
   Delta_v_i_1)
alpha_zero_v = -(alpha_minus_2_v + alpha_minus_v)

# delta's for v direction (2nd derivative)
delta_minus_v = 2 / (Delta_v_i * (Delta_v_i + Delta_v_i_1))
delta_plus_v = 2 / (Delta_v_i_1 * (Delta_v_i + Delta_v_i_1))
delta_zero_v = -(delta_minus_v + delta_plus_v)


# Pad the arrays to work in sparse matrix format using np.pad
alpha_minus_2_v = np.pad(alpha_minus_2_v, (0, 2), mode='
   constant')
alpha_minus_v = np.pad(alpha_minus_v, (1, 1), mode='constant')
alpha_zero_v = np.pad(alpha_zero_v, (2, 0), mode='constant')

beta_minus_v = np.pad(beta_minus_v, (0, 2), mode='constant')
beta_zero_v = np.pad(beta_zero_v, (1, 1), mode='constant')
beta_plus_v = np.pad(beta_plus_v, (1, 0), mode='constant')

delta_minus_v = np.pad(delta_minus_v, (0, 2), mode='constant')
delta_zero_v = np.pad(delta_zero_v, (1, 1), mode='constant')
delta_plus_v = np.pad(delta_plus_v, (1, 0), mode='constant')
```

```
# Discretisation matrices

# e.g D_s_tilde is \widetilde{D}_S  - First derivative in S-
    direction - This follows central scheme
inputs = [beta_minus_s, beta_zero_s, beta_plus_s]
D_s_tilde = diags(inputs, offsets_central, shape=(m1_tilde,
    m1_tilde), format='lil')

# Central scheme isn't followed at the upper boundary, This is
     implementing the Neumann boundary
D_s_tilde[m1_tilde - 1, :] = 0


#  For \widetilde{D}_v - First derivative in v direction -
    Here we have 3 schemes(Forward scheme at v=0,
#                          Central scheme at between 0 and 1,
    Backward scheme for v larger than 1)

# Central scheme so using Beta's on diagonals
inputs = [beta_minus_v, beta_zero_v, beta_plus_v]
D_v_central_tilde = diags(inputs, offsets_central, shape=(
    m2_tilde, m2_tilde))

# Backward scheme with Alpha's on main diagonal and the two
    diagonals below
inputs = [alpha_minus_2_v, alpha_minus_v, alpha_zero_v]
D_v_backward_tilde = diags(inputs, offsets_backward, shape=(
    m2_tilde, m2_tilde))


# \widetilde{D}_v  (almost all together)
D_v_tilde = D_v_backward_tilde.tocsr()

D_v_central_tilde = D_v_central_tilde.tocsr()

# Now combine D_v_tilde (less than index_v_l1 being central
    scheme) and for bigger than index_v_l1 we have backward
    scheme
D_v_tilde[:index_v_l1, :] = D_v_central_tilde[:index_v_l1, :]


# Manually implementing forward scheme at the lower boundary,
    v=0. See section 5.3.3
Delta_v_i_1 = Delta_v[0]
Delta_v_i_2 = Delta_v[1]

D_v_tilde[0, :] = 0
D_v_tilde[0, 0] = ((-2 * Delta_v_i_1) - Delta_v_i_2) / (
```

```
    Delta_v_i_1 * (Delta_v_i_1 + Delta_v_i_2))   # gamma_0
D_v_tilde[0, 1] = (Delta_v_i_1 + Delta_v_i_2) / (Delta_v_i_1 *
    Delta_v_i_2)                              # gamma 1
D_v_tilde[0, 2] = -Delta_v_i_1 / (Delta_v_i_2 * (Delta_v_i_1 +
    Delta_v_i_2))                             # gamma 2


# D_ss_tilde  Second derivative in S-direction
inputs = [delta_minus_s, delta_zero_s, delta_plus_s]

D_ss_tilde = diags(inputs, offsets_central, shape=(m1_tilde,
    m1_tilde), format='lil')


# Neumann condition when i = m1_tilde
D_ss_tilde[-1, :] = 0

Delta_s_i = Delta_s[m1 - 1]
Delta_s_i_1 = 0                              # From Neumman = 0

D_ss_tilde[-1, -2] = 2 / (Delta_s_i * (Delta_s_i +
    Delta_s_i_1))      # delta -1

D_ss_tilde[-1, -1] = -2 / (Delta_s_i * (Delta_s_i +
    Delta_s_i_1))     # delta 0


# D_vv_tilde
inputs = [delta_minus_v, delta_zero_v, delta_plus_v]

D_vv_tilde = diags(inputs, offsets_central, shape=(m2_tilde,
    m2_tilde))


# D_s_mixed_tilde
inputs = [beta_minus_s, beta_zero_s, beta_plus_s]

D_s_mixed_tilde = diags(inputs, offsets_central, shape=(
    m1_tilde, m1_tilde))


# D_v_mixed_tilde
inputs = [beta_minus_v, beta_zero_v, beta_plus_v]

D_v_mixed_tilde = diags(inputs, offsets_central, shape=(
    m2_tilde, m2_tilde))


# Set the last row to 0 for D_s_mixed_tilde (see 5.3.3)
```

```
D_s_mixed_tilde = D_s_mixed_tilde.tocsc()
D_s_mixed_tilde[- 1, :] = 0

D_vv_tilde = D_vv_tilde.tocsc()
D_v_mixed_tilde = D_v_mixed_tilde.tocsc()



# From tilde to no tilde
D_s = D_s_tilde[index_s_boundary, :][:, index_s_boundary]

D_v = D_v_tilde[index_v_boundary, :][:, index_v_boundary]

D_ss = D_ss_tilde[index_s_boundary, :][:, index_s_boundary]

D_vv = D_vv_tilde[index_v_boundary, :][:, index_v_boundary]

D_s_mixed = D_s_mixed_tilde[index_s_boundary, :][:,
    index_s_boundary]

D_v_mixed = D_v_mixed_tilde[index_v_boundary, :][:,
    index_v_boundary]

# E_tilde matrix                    - Neumann condition, last
    row has 1s
E_tilde = csr_matrix((m1_tilde, m2_tilde))

# Set the last row to 1s
E_tilde[-2 + 1, :] = np.ones((m2_tilde, 1))


# Sparse identity matrix for S
I_s = identity(index_s_boundary_size)
# Sparse identity matrix for v
I_v = identity(index_v_boundary_size)
# Sparse identity matrix for v (extended)
I_v_tilde = identity(m2_tilde)



# A_0
comp_1 = (rho * sigma) * Y_diags @ D_v_mixed
comp_2 = X_diags @ D_s_mixed
A_0 = kron(comp_1, comp_2)
# plot_sparse_matrix(A_0, 'A_0')    nice plots (matches de
    Graaf)

# A_1
comp_1 = Y_diags
```

```
comp_2 = 0.5 * X_diags @ X_diags @ D_ss
comp_3 = I_v
comp_4 = (rd - rf) * X_diags @ D_s
comp_5 = 0.5 * rd
comp_6 = I_v
comp_7 = I_s
A_1 = kron(comp_1, comp_2) + kron(comp_3, comp_4) - comp_5 *
   kron(comp_6, comp_7)
# plot_sparse_matrix(A_1, "A_1")

# A_2
comp_1 = (0.5 * sigma ** 2) * Y_diags @ D_vv
comp_2 = kappa * (eta * I_v - Y_diags) @ D_v
kron_comp = kron(comp_1 + comp_2, I_s)
comp_3 = (0.5 * rd) * kron(I_v, I_s)
A_2 = kron_comp - comp_3
# plot_sparse_matrix(A_2, "A_2")

# Finally A (all together)
A = A_0 + A_1 + A_2

#plot_sparse_matrix(A, "A")

end2 = datetime.now()
time2 = (end2 - start2).total_seconds()

#print("Computation Time 2 (A matrix):", time2)



# G matrix (boundaries)

# g_0
comp_1 = D_s_mixed_tilde @ Gamma_tilde
comp_2 = X_tilde @ comp_1
comp_3 = comp_2 @ D_v_mixed_tilde.transpose()
comp_4 = comp_3 @ Y_tilde
g_0 = (rho * sigma) * comp_4


# g_1
comp_1_1 = D_ss_tilde @ Gamma_tilde
comp_1_2 = comp_1_1 + (2 / Delta_s[m1 - 1]) * E_tilde
comp_1_3 = X_tilde @ X_tilde
comp_1_4 = comp_1_3 @ comp_1_2
comp_1_5 = comp_1_4 @ Y_tilde
first_term = 0.5 * comp_1_5

comp_2_1 = D_s_tilde @ Gamma_tilde
```

```
comp_2_2 = comp_2_1 + E_tilde
comp_2_3 = X_tilde @ comp_2_2
second_term = (rd - rf) * comp_2_3

g_1 = first_term + second_term
g_1 = np.array(g_1)


# g_2
comp_1 = (0.5 * pow(sigma, 2))
first_term = comp_1 * Gamma_tilde @ D_vv_tilde.transpose() @
    Y_tilde

comp_2 = kappa * Gamma_tilde
second_term = comp_2 @ D_v_tilde.transpose() @ (eta *
    I_v_tilde - Y_tilde)

g_2 = first_term + second_term


g_0_sliced = g_0[np.ix_(index_s_boundary, index_v_boundary)]
g_0_vector = g_0_sliced.flatten(order='F')


g_1_sliced = g_1[np.ix_(index_s_boundary, index_v_boundary)]
g_1_vector = g_1_sliced.flatten(order='F')

g_2_sliced = g_2[np.ix_(index_s_boundary, index_v_boundary)]
g_2_vector = g_2_sliced.flatten(order='F')

# Ensure all vectors have the same shape
g_0 = g_0_vector
g_1 = g_1_vector
g_2 = g_2_vector


g = g_0 + g_1 + g_2


if scheme == 'Do':

    theta = 0.5

if scheme == 'CS':

    theta = 0.5

if scheme == 'MCS':
```

```
        theta = 1/3

    if scheme == 'HV':

        theta = 1/2 + ((1/6) * math.sqrt(3))

    if theta_exp != 0.0:

        theta = theta_exp

    #start3 = datetime.now()




    start = datetime.now()  # Start timing ADI loop from LU decomp

    # LU Decomposition
    I = kron(I_v, I_s)

    LHS_2 = sp.eye(I.shape[0]) - (theta * dt) * A_2

    LHS_1 = sp.eye(I.shape[0]) - (theta * dt) * A_1

    # Decompose A_2 part
    LU1 = spla.splu(LHS_1)

    # Decompose A_1 part
    LU2 = spla.splu(LHS_2)



    if scheme == 'Do':

        damping_Do = False

        if damping_Do:
            print("Damping")

            half_point = 0.5 * dt
            I = sp.identity(A.shape[0])  # Assuming A is square

            # Two Backward Euler steps involving A     --
               Decompose the matrix A
            matrix_sparse = sp.eye(I.shape[0]) - half_point * A
            LU = spla.splu(matrix_sparse)
```

```
            # First solve step
            u = LU.solve(u + half_point *  g)

            # Second solve step)
            u = LU.solve(u + half_point * g)


        for n in range(1, N + 1):
            gr = g
            gr0 = g_0
            gr1 = g_1
            gr2 = g_2
            u = Do(u, dt, LU1, LU2, A, g, g_1, gr1, g_2, gr2,
                theta)
            g = gr
            g_0 = gr0
            g_1 = gr1
            g_2 = gr2


    if scheme == 'CS':

        damping_CS = False

        if damping_CS:
            print("Damping")

            half_point = 0.5 * dt
            I = sp.identity(A.shape[0])

            # Two Backward Euler steps involving A      --
                Decompose the matrix A
            matrix_sparse = sp.eye(I.shape[0]) - half_point * A
            LU = spla.splu(matrix_sparse)

            # First solve step
            u = LU.solve(u + half_point * g)

            # Second solve step
            u = LU.solve(u + half_point * g)


            ###### Attempting Douglas as a damp, works slightly
                better but still looks same order as Douglas for CS
                .
            """
            # Predictor step
            Y0 = half_point * (A @ u + g)
```

```
            # Two corrector steps
            Y = LU1.solve(z0 + theta * half_point * (g_1 - g_1))
            Y = LU2.solve(z + theta * half_point * (g_2 - g_2))

            # Final update
            u = u + Y
            """

        for n in range(1, N + 1):
            gr = g
            gr0 = g_0
            gr1 = g_1
            gr2 = g_2
            u = CS(u, dt, LU1, LU2, A, A_0, g, g_0, g_1, g_2, gr,
                gr0, gr1, gr2, theta)
            g = gr
            g_0 = gr0
            g_1 = gr1
            g_2 = gr2

    if scheme == 'MCS':
        for n in range(1, N + 1):
            gr = g
            gr0 = g_0
            gr1 = g_1
            gr2 = g_2
            u = MCS(u, dt, LU1, LU2, A, A_0, g, g_0, g_1, g_2, gr,
                gr0, gr1, gr2, theta)
            g = gr
            g_0 = gr0
            g_1 = gr1
            g_2 = gr2

    if scheme == 'HV':
        for n in range(1, N + 1):
            gr = g
            gr0 = g_0
            gr1 = g_1
            gr2 = g_2
            u = HV(u, dt, LU1, LU2, A, A_0, g, g_0, g_1, g_2, gr,
                gr0, gr1, gr2, theta)
            g = gr
            g_0 = gr0
            g_1 = gr1
            g_2 = gr2

    end = datetime.now()
    time = (end - start).total_seconds()
```

```
#print("Computation Time (ADI loop):", time)

# Reshape for plotting + Greeks calculations   -  First is
    forward then central after
temp_vec = u

U = np.zeros((m1_tilde, m2_tilde))

U[np.ix_(index_s_boundary, index_v_boundary)] = temp_vec.
    reshape((index_s_boundary_size, index_v_boundary_size),
    order='F')

# Ensure boundaries for Greeks work at S_0

D_s_tilde[0, :] = 0
Delta_s_i_1 = Delta_s[0]
Delta_s_i_2 = Delta_s[1]

D_s_tilde[0, 0] = (- 2 * Delta_s_i_1 - Delta_s_i_2) / (
    Delta_s_i_1 * (Delta_s_i_1 + Delta_s_i_2))

D_s_tilde[0, 1] = (Delta_s_i_1 + Delta_s_i_2) / (Delta_s_i_1 *
     Delta_s_i_2)

D_s_tilde[0, 2] = -Delta_s_i_1 / (Delta_s_i_2 * (Delta_s_i_1 +
    Delta_s_i_2))


D_ss_tilde[0, :] = 0

D_ss_tilde[0, 0] = 2 / (Delta_s_i_1 * (Delta_s_i_1 +
    Delta_s_i_2))

D_ss_tilde[0, 1] = - 2 / (Delta_s_i_1 * Delta_s_i_2)

D_ss_tilde[0, 2] = 2 / (Delta_s_i_2 * (Delta_s_i_1 +
    Delta_s_i_2))

D_s_tilde = D_s_tilde.tocsr()
D_ss_tilde = D_ss_tilde.tocsr()
D_v_tilde = D_v_tilde.tocsr()


# the Greeks
Delta = D_s_tilde @ U
Gamma = D_ss_tilde @ U
Vega = U @ D_v_tilde.transpose()

def truncate_for_plot(arr, cutoff_left, cutoff_right):
```

```
    return arr[:cutoff_left, :cutoff_right]

S = truncate_for_plot(S, index_left_plot, index_v_l1)
V = truncate_for_plot(V, index_left_plot, index_v_l1)
U = truncate_for_plot(U, index_left_plot, index_v_l1)
Delta = truncate_for_plot(Delta, index_left_plot, index_v_l1)
Gamma = truncate_for_plot(Gamma, index_left_plot, index_v_l1)
Vega = truncate_for_plot(Vega, index_left_plot, index_v_l1)

# Plots (set to True or False to plot only price surface, or
   all, or none)
plotPS = False

if plotPS:
    # Calculate maxU
    maxU = np.max(U)

    # Create the plot
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # Plot the surface
    surf = ax.plot_surface(S, V, U, cmap='viridis')

    # Set limits
    ax.set_xlim(0, 2 * K)
    ax.set_ylim(0, 1)
    ax.set_zlim(0, maxU)


    # Set labels and title
    ax.set_xlabel('S', fontsize=14)
    ax.set_ylabel('V', fontsize=14)
    ax.set_zlabel('Option␣Value', fontsize=14)
    ax.set_title('Option␣value␣under␣Heston', fontsize=14)

    # Add color bar which maps values to colors
    fig.colorbar(surf)

    # Show the plot
    plt.show()



plot_all = True # Plotting option value under heston

if plot_all:
    # Plotting option value under heston
```

```
# Calculate maxU
maxU = np.max(U)

# Create the plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(S, V, U, cmap='viridis')



# Set labels and title
ax.set_xlabel('S', fontsize=14)
ax.set_ylabel('V', fontsize=14)
ax.set_zlabel('Option␣Value', fontsize=14)
ax.set_title('Option␣value␣under␣Heston', fontsize=14)

# Add color bar which maps values to colors
fig.colorbar(surf)

# Show the plot
plt.show()

# Plot Delta
fig1 = plt.figure()
ax1 = fig1.add_subplot(111, projection='3d')
surf1 = ax1.plot_surface(S, V, Delta, cmap='viridis')
ax1.set_xlim([0, 2 * K])
ax1.set_ylim([0, 1])
ax1.set_xlabel('S', fontsize=14)
ax1.set_ylabel('V', fontsize=14)
ax1.set_title('Delta', fontsize=14)
fig1.colorbar(surf1)
plt.show()

# Plot Gamma
fig2 = plt.figure()
ax2 = fig2.add_subplot(111, projection='3d')
surf2 = ax2.plot_surface(S, V, Gamma, cmap='viridis')
ax2.set_xlim([0, 2 * K])
ax2.set_ylim([0, 1])
ax2.set_xlabel('S', fontsize=14)
ax2.set_ylabel('V', fontsize=14)
ax2.set_title('Gamma', fontsize=14)
fig2.colorbar(surf2)
plt.show()

# Plot Vega
```

```
    fig3 = plt.figure()
    ax3 = fig3.add_subplot(111, projection='3d')
    surf3 = ax3.plot_surface(S, V, Vega, cmap='viridis')
    ax3.set_xlim([0, 2 * K])
    ax3.set_ylim([0, 1])
    ax3.set_xlabel('S', fontsize=14)
    ax3.set_ylabel('V', fontsize=14)
    ax3.set_title('Vega', fontsize=14)
    fig3.colorbar(surf3)
    plt.show()



#   Getting specific Option Values - First interpolator is
    just nearest grid point

# Create the interpolator using NearestNDInterpolator
points = np.array([(S[i, j], V[i, j]) for i in range(S.shape
    [0]) for j in range(S.shape[1])])
values = U.flatten()
nearest_interpolator = NearestNDInterpolator(points, values)

# Specify the current values of S and V
S_current = 120.0  # Replace with the current value of S
V_current = 0.4  # Replace with the current value of V

# Interpolate to get the option price at the current S and V
option_price = nearest_interpolator((S_current, V_current))

true_price = heston_price(S_current, K, T, rd, rf, V_current,
    kappa, eta, sigma, rho)

#true_price = 45.63379764919681

print("TRUE␣PRICE:␣", true_price)

#print("(Nearest grid point (bad)) Option price at S =",
    S_current, "and V =", V_current, "is", option_price)

#print("(Nearest grid point (bad)) Relative error:", abs((
    option_price - true_price) / true_price))

# Better interpolator
points = np.array([(S[i, j], V[i, j]) for i in range(S.shape
    [0]) for j in range(S.shape[1])])
values = U.flatten()

linear_interpolator = LinearNDInterpolator(points, values)
```

```
    option_price_linear = linear_interpolator(S_current, V_current
        )

    print("(LinearNDInterpolator) Option price at S =", S_current,
         "and V =", V_current, "using LinearNDInterpolator is",
        option_price_linear)

    print("(LinearNDInterpolator) Relative error :", abs((
        option_price_linear - true_price) / true_price))

    # Return for global spatial/temporal functions
    return U, S, V


start4 = datetime.now()

FDMHeston()

end4 = datetime.now()
time4 = (end4 - start4).total_seconds()

print("Computation time (All): ", time4)
```