Option pricing in the Heston Model using Neural Networks

Neil Drew-Lopez

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Financial Mathematics

University of Liverpool
June 24, 2025

Contents

Statement of Originality

This dissertation was written by me, in my own words, except for quotations from published and unpublished sources which are clearly indicated and acknowledged as such. I am conscious that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism, according to the University Academic Integrity Policy. The source of any picture, map or other illustration is also indicated, as is the source, published or unpublished, of any material not resulting from my own research.

Chapter 1

Introduction

In the constantly evolving field of Financial Mathematics, option pricing remains one of the most important areas of focus. Accurate pricing is essential for traders and financial institutions, impacting investment decisions and risk management strategies. Traditional models like the Black-Scholes-Merton model have been widely applied due to their simplicity and analytical solutions. However, they fall short in capturing the true nature of market volatility. This limitation has led to the development of new option pricing models that treat volatility as a random process.

One of the most first and most notable of these is the Heston model, a stochastic volatility model that accurately reflects market behaviors and offers a closed form solution. This thesis concentrates on the Heston model with a focus on pricing European call options. In the first part, I will derive the characteristic functions, explore numerical methods for solving them, and analyze the advantages and disadvantages of using the Heston model compared to the Black-Scholes Model.

To illustrate the usefulness of models like Black-Scholes and Heston, consider the analogy of representing the Earth as a sphere. While we know the Earth is not a perfect sphere, this simplification allows us to use latitude and longitude for navigation, making complex calculations more manageable. Similarly, the Black-Scholes model provides a simplified framework that makes it easier to perform calculations related to option pricing, even though it doesn't accurately model volatility. Recognizing the model's limitations, we can then consider deviations—like stochastic volatility—to gain more accurate and useful information.

In the second part of the thesis, I focus on the advancements in technology and the growing influence of artificial intelligence in Finance. Since ancient times, humans have been fascinated by the idea of machines that can think, as seen in Greek mythology with figures like Hephaestus, who created mechanical servants, and Pandora (Mayor 2018). In the modern age machine learning techniques are being applied to every sector from agriculture to medicine and especially finance. I apply deep learning techniques to large datasets of Heston model parameters and their corresponding European call prices. I will examine the architecture of artificial neural networks and implement Python code to demonstrate their ability to learn and replicate the Heston model.

By combining traditional financial theories with modern machine learning methods, this thesis aims to contribute to a deeper understanding of option pricing models and explore how artificial intelligence can enhance financial modeling.

Chapter 2

Literature Review

To summarize the relevant literature for this thesis one must begin by mentioning earlier models that formed the foundation for derivative pricing. The first option pricing model to provide a closed form solution is known as the Black-Scholes-Merton model developed in 1973 originally published in “The Pricing of Options and Corporate Liabilities” by Fischer Black and Myron Scholes (Black & Scholes 1973) and revised in “Theory of Rational Option Pricing” by Robert C. Merton (Merton 1973). The model is still used to this day for its simplicity and relative accuracy despite its assumption that volatility is constant and difficulty coping in times of financial turbulence. This flaw was highlighted in the 1987 financial crash which led to the development of early stochastic volatility models seen in “Option Pricing When the Variance Changes Randomly: Theory, Estimation and an Application” by Louis O. Scott (Scott 1987) and “The Pricing of Options on Assets with Stochastic Volatilities” by John Hull and Alan White (Hull & White 1987). During this time important concepts such as the mean-reverting square-root process were used in the evolution of interest rate models such as “A theory of the term structure of interest rates” by John Cox, Jonathan Ingersoll and Stephen Ross (Cox, Ingersoll & Ross 1985).

This research laid the groundwork for Steven L. Heston’s landmark paper “A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options” (Heston 1993) in an effort to address the limitations of the Black-Scholes model. Following this paper several important texts emerged that solidified the model’s place in financial literature. John Hull’s “Options, Futures, and Other Derivatives” (Hull 1997) and Steven Shreve’s “Stochastic Calculus for Finance II” (Shreve 2004) are popular examples. Numerical methods for solving for the option price were focused on such as the Fourier method in “Option valuation using the fast Fourier transform” by Carr and Madan (Carr & Madan 1999). Studies also emerged comparing the Heston model to the Black-Scholes model and other stochastic volatility models as in Gurdip Bakshi and Chales Cao and Zhiwu Chen’s “Empirical Performance of Alternative Option Pricing Models” (Bakshi, Cao & Chen 1997). In more recent years Jim Gatheral’s “The Volatility surface: A practitioner’s Guide (Gatheral 2006), Fabrice Rouah’s “The Heston model and its extensions in Matlab and C#” (Rouah 2013) and “The Volatility smile” by Emanuel Derman and Michael B. Miller (Derman & Miller 2016) have offered further applications and extensions to the model. “The Little Heston Trap” (Albrecher, Mayer, Schoutens & Tistaert 2007) is a notable article which discusses a slight idiosyncrasy of the model.

I will now discuss the evolution of literature pertaining to Artificial Neural Networks, the theory of which dates back to as early as 1943 in "A logical calculus of the ideas immanent in nervous ativity" by Warren S. McCulloch and Walter Pitts (McCulloch & Pitts 1943) which proposed the first mathematical model of a human neuron. This led to Frank Rosenblatt's work "The perceptron: a probabilistic model for information storage and organisation in the brain" (Rosenblatt 1958) which provided one of the first algorithms for supervised learning. However Marvin Minsky and Seymour Papert's book "Perceptrons: An Introduction to Computational Geometry" (Minsky & Papert 1969) highlights some of the limitations of the perceptron.

Interest in Neural Networks was revived in the 1980's due in part to the development of the back-propagation algorithm in "Learning Representations by Back-Propagating Errors" (Rumelhart, Hinton & Williams 1986), a paper by David Rumenhalt, Geoffrey Hilton and Ronald Williams. Rumenhalt also published a book the same year with James McClelland "Parallel Distributed Processing: Explorations in the Microstructure of Cognition" (Rumelhart & McClelland 1986) which provided a comprehensive introduction to the capabilities of neural networks.

The 2000s saw the introduction of Deep Learning in which deeper architectures, trained with large data sets and combined with greater computational efficiency allowed for superior performance in tasks such as image recognition. "Deep Learning" by Ian Goodfellow, Yoshua Bengio and Aaron Courville (Goodfellow, Bengio & Courville 2016) provides a comprehensive overview of the field and combines theoretical background with practical applications.

In Finance the application of Artificial Neural Networks has grown significantly in the last decade. "Deep Learning with Long Short-Term Memory Networks for Financial Market Predictions" by Thomas Fischer and Christopher Krauss (Fischer & Krauss 2018) discusses the prediction of stock price movements while "Artificial Intelligence in Finance: A Python-Based Guide" by Yves Hilpisch (Hilpisch 2020) provides insights into the application of machine learning techniques to financial markets.

However papers concerning option pricing in the Heston model using Artificial Neural Networks are sparse. "Neural Networks for Option Pricing and Hedging: A literature review" by Johannes Ruf and Weiguan Wang (Ruf & Wang 2020) categorizes around 150 papers relevant to the topic though few focus specifically on the Heston model. Research such as "Efficient Simulation of the Heston Stochastic Volatility Model" by Leif Anderson (Andersen 2007) and "Option Pricing Using Neural Networks" by Norbert Forgasi (Fogarasi 2004) discuss implementing the Heston model and machine learning strategies for option pricing respectively. Notable works in recent years include "Deep Learning and the Heston Model: Calibration and Hedging" by Oliver Klinberg Malmer and Victor Tisel (Klingberg Malmer & Tisell 2020) and "A neural network approach to pricing of a European Call Option with the Heston model" by Sandra Patricia Guerrero Torres (Guerrero Torres 2019) which explore the applications of Deep Learning to Heston model.

This thesis endeavours to expand on the work that has been done previously and provide an overview of the mathematics that govern the Heston model, an analysis of its numerical solving methods and a comparison to the classic Black-Scholes model. I discuss the theory behind the architecture of Artificial Neural Networks and modern optimisation techniques. Finally I give a comprehensive guide to creating a model in Python that reliably predicts the price of European Call options using the Heston model and discuss the results and recommendations for future research.

Chapter 3

Mathematical Theory

3.1 Background to financial derivatives

This section explains financial derivatives, why they are important and how they can be defined. The concepts are discussed at length in (Hull 2017), see also for further reading on the topic.

In Finance, derivatives play a key role and the derivatives market is massive, with options and futures being traded in fast paced environments. A derivative is a financial instrument that gets its value from the value of other, simpler, underlying variables such as the prices of traded assets. The simplest form of financial derivative is a forward contract, which is an agreement between buyer and seller to buy or sell an asset at a future time for a certain price. Another form of financial derivative is the option. In this thesis I will be looking at European vanilla options which give the holder the right but not the obligation to buy or sell at a predetermined date and price. The date in the contract is time to maturity T and the price as exercise price or strike price K. American options differ slightly in that they can be exercised at any point up to the expiration date; this makes them more complex. Vanilla options can be two types, call and put. A call option gives the buyer the right to buy the underlying asset and a put option gives the seller the right to sell. The payoff structures of these options are quite simple.

$$\text{Payoff}_{\text{call}} = \max(S_T - K, 0)$$

$$\text{Payoff}_{\text{put}} = \max(K - S_T, 0)$$

where S_T denotes the price of the asset at terminal time T. Here we can introduce the concept of moneyness using $S_T/K = M$. A call option is ITM (in-the-money) in $M > 1$ and OTM (out-the-money) if $M < 1$. So for the Call option if $K < S_T$, the call option has positive pay off and is ITM and for the put option the payoff is zero and it is OTM. The pay-off stucture can be seen in Figure 3.1.

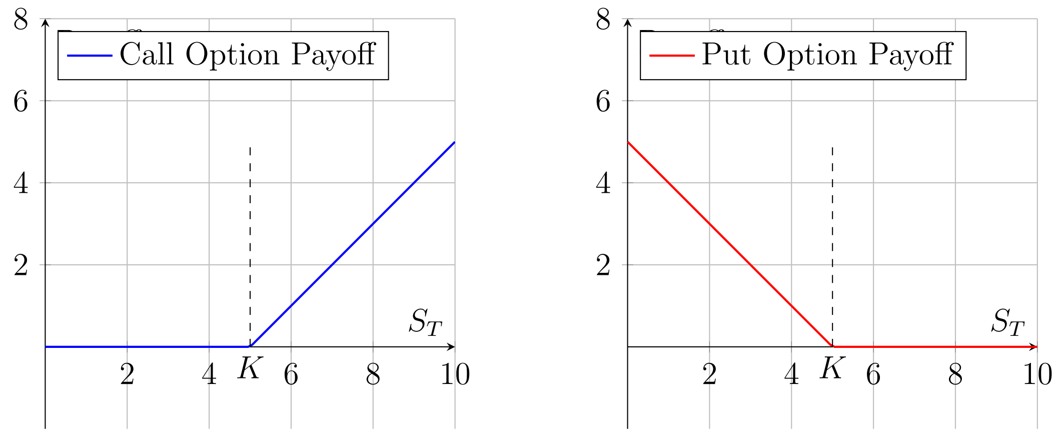


Figure 3.1: Call and Put option Payoff

Source: created using TikZ in LaTeX

3.2 Probability Background

This section discusses the basics of probability theory required to understand some of the ideas found in later sections and define key concepts such as Geometric Brownian motion. For definitions and further reading, refer to Durret (2019).

A probability space forms the framework for describing random events and their probabilities, and is the foundation upon which stochastic modelling is built. It is comprised of three components known as a probability triple $(\Omega, \mathcal{F}, \mathbb{P})$

1. **Sample Space** (Ω): The set of all possible outcomes

$$\Omega = \{\omega_1, \omega_2, \dots, \omega_n\}$$

2. **Sigma Algebra** $(\mathcal{F})$: A collection of subsets of Ω that satisfies the following axioms:
   (a) $\Omega \in \mathcal{F}$
   (b) If $A \in \mathcal{F}$, then $A^c \in \mathcal{F}$
   (c) If $\{A_i\}_{i=1}^{\infty} \subseteq \mathcal{F}$, then $\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$

3. **Probability Measure** (P): A function P: $F \to [0, 1]$ that assigns a probability to each event in, satisfying the following axioms:
   (a) $P(\Omega) = 1$
   (b) For any countable collection of disjoint sets $\{A_i\}_{i=1}^{\infty} \subseteq \mathcal{F}$,

$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

We may also must introduce the concept of a filtration. A filtration $F := (F_t)_{t \in T}$ is defined as an increasing collection of σ-algebras which in this case models the evolution over time. We now have a filtered probability space $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$

We are now able to model stochastic processes which are families of stochastic variables defined on the probability space. The simplest form of this is known as the Geometric Brownian Motion or Wiener Process and can be defined as follows (Schilling 2021):

The Wiener process $(W_t)_{t\geq 0}$ is a stochastic process with the following properties:

1. $W_0 = 0$. The initial point of the process is zero.
2. Increments in W are independent.
3. W_t is almost surely continuous.
4. $W_t \sim N(0,t)$. All increments of W are normally distributed with $E[W_t - W_s] = 0$ and $\mathrm{Var}(W_t - W_s) = t - s$ for $t \geq s$.

Itô's lemma Let $f(t,x)$ be a smooth function of two variables, and let X_t be a stochastic process satisfying $dX_t = \mu dt + \sigma dB_t$, for a Brownian motion B_t. Then

$$df(t,X_t) = \left(\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2}\right)dt + \frac{\partial f}{\partial x}dB_t. \tag{3.1}$$

We now have the suitable tools to represent stochastic processes in their differential form, known as SDE's (stochastic differential equations). To solve these equations one must apply stochastic calculus making use of Ito's formula. We are now ready to begin pricing options.

3.3 The Black-Scholes Model

In 1973, Fischer Black and Myron Scholes transformed the field of option pricing by introducing their revolutionary option pricing model in (Black & Scholes 1973). Robert C. Merton expanded the model in (Merton 1973). The model redefined option pricing by allowing for closed form solutions; it forms the base for the majority of derivative pricing. In this section I discuss the model and its limitations.

I begin first with the Black-Scholes differential equation:

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 f}{\partial S^2} = rf \tag{3.2}$$

which is derived under the assumption that the stock price process is as follows with μ and σ constant:

$$dS = \mu S\, dt + \sigma S\, dz \tag{3.3}$$

From this one can derive the Black-Scholes-Merton pricing formulas for European Call and Put options respectively.

$$c = S_0 N(d_1) - Ke^{-rT}N(d_2) \tag{3.4}$$

$$p = Ke^{-rT}N(-d_2) - S_0 N(-d_1) \tag{3.5}$$

where

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}} \tag{3.6}$$

$$d_2 = \frac{\ln(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T} \tag{3.7}$$

where c and p are the prices of European call and put options respectively, S_0 is the price of the stock at time zero, K is the strike price, r is the continuously compounded risk-free rate, $N(x)$ as a function is the cdf of the standard normal distribution, σ is the volatility and time to maturity is denoted $T - t$ (Black & Scholes 1973).

As mentioned before there are some limitations to the Black-Scholes model. First the SDE oversimplifies the evolution of stock prices over time by assuming that returns are log-normally distributed with known mean and variance. This can distort the probability that the contract expires ITM, which in turn affects the options price. This model also assumes that volatility is constant but market data shows that volatility changes with different strikes and maturities (Ané & Labidi 2001). The model also assumes continuous time when trading and hedging occur in discrete time.

These limitations led to the development of new models to counteract them, focusing primarily on Volatility and modelling it stochastically.

Chapter 4

The Heston Model

This section introduces the concept of Stochastic volatility models and goes on to define the Heston model and to derive its corresponding formulae.

4.1 Stochastic Volatility

On October 19th 1987 the stock market experienced a significant crash now dubbed “Black Monday”, a devastating global financial crisis that ranks second in the Dow Jones Industrial Average “DJIA” history for its daily decrease of 22.61% (Marotta 2018).

Prior to this the financial market exclusively used the Black-Scholes Model, but the crash exposed some of the previously discussed weaknesses of the model and its inability to cope in times of such financial turbulence. Emperical studies since the crash have found that the assumption of continuously compounded stock returns being normally distributed with constant volatility does not hold in equity markets (Nunno, Kubilius, Mishura & Yurchenko-Tytarenko 2023). Returns show skewness and kurtosis - fat tails - which normality cannot account for. Volatility tends to be somewhat inversely proportional to price, and is not constant. Researchers began attempting to develop models that could account for this changing volatility. Stochastic volatility models interpret volatility as its own random process that changes randomly over time.

Original attempts by Scott (1987) and Hull and White (1987) attempted to re-calibrate the Black-Scholes model to allow for stochastic volatility; the problem with this was the lack of a closed-form solution which led to very complex partial differential equations that required numerical techniques to solve. New models were created to avoid some of these issues and produce closed-form solutions.

Stochastic Volatility Models are joint models of price and variance that allow prices to be dependent on the variance process, where variance and therefore volatility is modelled as a latent stochastic process. This allows the mean reverting process to be captured and makes use of two different but correlated Brownian motions there are many different stochastic volatility models but in this thesis I will primarily focus on one of the first and most important, the Heston Model.

4.2 The Heston Model

The Heston model, as the name suggests, was first introduced in 1993 by Steven L. Heston; the original being titled "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". It presents a closed form solution for a stochastic volatility model. In this section I follow and take inspiration from the derivation presented in Rouah (2013).

The Heston model is characterised by the following set of stochastic differential equations where S_t is the price of the asset at time t and follows a Black-Scholes type process. V_t is the instantaneous variance of the asset price at time t and follows a Cox, Ingersoll and Ross process (Cox et al. 1985):

$$\begin{aligned} dS_t &= \mu S_t \, dt + \sqrt{V_t} S_t \, dW_t^S, \\ dV_t &= \kappa(\theta - V_t) \, dt + \sigma \sqrt{V_t} \, dW_t^V, \\ \rho \, dt &= dW_t^S \, dW_t^V, \end{aligned} \tag{4.1}$$

where:

- μ is the drift rate of the asset price,
- $\kappa > 0$ is the rate of mean reversion for the variance,
- $\theta > 0$ is the mean reversion level for the variance,
- $\sigma > 0$ is the volatility of the variance,
- W_t^S and W_t^V are two correlated Wiener processes (Brownian motions) with correlation $\rho \in [-1, 1]$.

The parameters within the model have notable characteristics. Naturally a higher long term mean θ means higher prices of options, and high values of κ mean a higher rate of reversion of the process to θ. Parameter ρ affects skewness of the returns distribution which in turn affects the skewness of the volatility smile. Parameter ρ can be positive or negative which stretches the right and left tails of the distribution respectively. Parameter σ changes the kurtosis of the returns distribution and therefore steepness of the implied volatility smile.

It is important to note that the volatility $\sqrt{v_t}$ is not modeled directly in the Heston model but instead though the variance v_t. The variance process comes from the Orstein-Uhlenbeck process for the volatility $h_t = \sqrt{v_t}$ given by:

$$dh_t = -\beta h_t dt + \delta dW_{t_1} \tag{4.2}$$

Applying Itô's lemma, $v_t = h_t^2$ follows the process

$$dv_t = (\beta^2 - 2\beta v_t)dt + 2\delta\sqrt{v_t}dW_{t_2} \tag{4.3}$$

if we define $\kappa = 2\beta$, $\theta = \delta^2/(2\beta)$ and $\sigma = 2\delta$ then I have dv_t from equation (4.1) as (4.3).

In order to begin pricing I need to represent the model under the risk neutral measure $\mathbb{Q}$, this can be done using the Girsanov theorem. The risk-neutral process is:

$$\begin{aligned} dS &= rSdt + \sqrt{v}Sd\hat{W}_{1,t}, \\ dv &= \kappa(\theta - v)dt + \sigma\sqrt{v}d\hat{W}_{2,t} \end{aligned} \tag{4.4}$$

where

$$\begin{aligned} E^{\mathbb{Q}}[dW_t^1\, dW_t^2] &= \rho dt \\ \kappa^* = \kappa + \lambda, &\quad \theta^* = \frac{\kappa\theta}{\kappa + \lambda} \\ \tilde{W}_{1,t} &= \left(W_{1,t} + \frac{\mu - r}{\sqrt{v_t}}t\right) \\ \tilde{W}_{2,t} &= \left(W_{2,t} + \frac{\lambda(S_t, v_t)}{\sigma\sqrt{v_t}}t\right) \end{aligned}$$

When $\lambda = 0$, $\kappa = \kappa^*$ and $\theta = \theta^*$ the physical and risk-neutral versions of these are the same. Throughout this thesis one can assume $\lambda = 0$. When it is obvious we will be using the risk-neutral process I will remove the asterix from the parameters and the tilde from the Brownian Motion.

4.2.1 The Partial Differential Equation

In this section, I outline the process of deriving the partial differential equation (PDE) for the Heston model. The approach is similar to the hedging argument used for deriving the Black-Scholes PDE, where a portfolio consisting of the underlying stock and a single derivative is formed to hedge the stock and make the portfolio riskless. However, the Heston model requires an additional derivative in the portfolio to hedge against volatility. Thus, I create a portfolio with an option $V = V(S, v, t)$, Δ_1 units of the stock, and Δ_2 units of another option $U(S, v, t)$ for the volatility hedge. The portfolio value is given by:

$$\Pi = V + \Delta_1 S + \Delta_2 U \tag{4.5}$$

For simplicity, I omit the t subscripts. Assuming the portfolio is self-financing, the change in its value is:

$$d\Pi = dV + \Delta_1 dS + \Delta_2 dU \tag{4.6}$$

The method here is similar to that used for the Black-Scholes case. I apply Itô's lemma to determine the processes for U and V, enabling the finding of the process for Π. The values of Δ_1 and Δ_2 that make the portfolio riskless are then derived, and this result is used to derive the Heston PDE.

To create the hedging portfolio, I first apply Itô's lemma to the value of the first derivative, $V(S,t)$. This gives:

$$dV = \frac{\partial V}{\partial t}dt + \frac{\partial V}{\partial S}dS + \frac{\partial V}{\partial v}dv + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}dS^2 + \frac{1}{2}\frac{\partial^2 V}{\partial v^2}dv^2 + \frac{\partial^2 V}{\partial S \partial v}dSdv \tag{4.7}$$

Here, $dS^2 = S^2\sigma^2 dt$, $dSdv = \Delta_2 S\sigma v dt$, and $dv^2 = v^2 dt$. Hence

$$dV = \frac{\partial V}{\partial t}dt + \frac{\partial V}{\partial S}dS + \frac{\partial V}{\partial v}dv + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2}dt + \frac{1}{2}\sigma^2 v\frac{\partial^2 V}{\partial v^2}dt + \sigma\rho v S\frac{\partial^2 V}{\partial S \partial v}dt \tag{4.8}$$

Applying Itô's lemma to the second derivative $U(S,v,t)$ results in the same expression in terms of U. Substituting these into (4.6) yields:

$$\begin{aligned} d\Pi &= dV + \Delta_1 dS + \Delta_2 dU \\ &= \left[\frac{\partial V}{\partial t} + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2} + \rho\sigma v S\frac{\partial^2 V}{\partial S\partial v} + \frac{1}{2}\sigma^2 v\frac{\partial^2 V}{\partial v^2}\right]dt \\ &\quad + \Delta_2\left[\frac{\partial U}{\partial t} + \frac{1}{2}S^2 v\frac{\partial^2 U}{\partial S^2} + \sigma\rho v S\frac{\partial^2 U}{\partial S\partial v} + \frac{1}{2}\sigma^2 v\frac{\partial^2 U}{\partial v^2}\right]dt \\ &\quad + \left[\frac{\partial V}{\partial S} + \Delta_2\frac{\partial U}{\partial S} + \Delta_1\right]dS \\ &\quad + \left[\frac{\partial V}{\partial v} + \Delta_2\frac{\partial U}{\partial v}\right]dv. \end{aligned} \tag{4.9}$$

For the portfolio to be riskless against movements in both stock and volatility, the last two terms must be zero, giving:

$$\Delta_2 = -\frac{\frac{\partial V}{\partial v}}{\frac{\partial U}{\partial v}}, \quad \Delta_1 = -\Delta_2\frac{\partial U}{\partial S} - \frac{\partial V}{\partial S} \tag{4.10}$$

Substituting these values results in:

$$\begin{aligned} d\Pi = &\left[\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 V}{\partial v^2} + \sigma\rho\nu S\frac{\partial^2 V}{\partial S\partial v}\right]dt \\ &+ \Delta_2\left[\frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 U}{\partial S^2} + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial v^2} + \sigma\rho\nu S\frac{\partial^2 U}{\partial S\partial v}\right]dt \end{aligned} \tag{4.11}$$

The condition that the portfolio earns the risk-free rate r implies that $d\Pi = r\Pi dt$ which can be written as $d\Pi = (A + \Delta_2 B)dt$, a simplification presented in Rouah (2010). Hence one has

$$A + \Delta_2 B = r(V + \Delta_1 S + \Delta_2 U). \tag{4.12}$$

Substituting for Δ_2 and re-arranging, produces the equality

$$\frac{A - rV + rS\frac{\partial V}{\partial S}}{\frac{\partial V}{\partial v}} = \frac{B - rU + rS\frac{\partial U}{\partial S}}{\frac{\partial U}{\partial v}} \tag{4.13}$$

Here I show the PDE in terms of the price. The left-hand side of Equation (4.13) is a function of V only, and the right-hand side is a function of U only. This implies that both sides can be written as a function $f(S, v, t)$ of S, v, and t. Following Heston, specify this function as $f(S, v, t) = -\kappa(\theta - v) + \lambda(S, v, t)$, where $\lambda(S, v, t)$ is the price of volatility risk. Substitute $f(S, v, t)$ into the left-hand side of Equation (4.13), substitute for B, and rearrange to produce the Heston PDE expressed in terms of the price S, This is Equation (6) of Heston (1933):

$$\begin{aligned} \frac{\partial U}{\partial t} + \frac{1}{2}\nu S^2 \frac{\partial^2 U}{\partial S^2} + \rho\sigma S\nu \frac{\partial^2 U}{\partial S \partial v} + \frac{1}{2}\sigma^2\nu \frac{\partial^2 U}{\partial v^2} \\ -rU + rS\frac{\partial U}{\partial S} + [\kappa(\theta - v) - \lambda(S, v, t)]\frac{\partial U}{\partial v} = 0. \end{aligned} \tag{4.14}$$

Next I show the PDE in terms of the Log Price. Let $x = \ln S$ and express the PDE in terms of x, t and v instead of S, t, and v. This leads to a simpler form of the PDE. We need the following derivatives, which are straightforward to derive:

$$\frac{\partial U}{\partial S}, \quad \frac{\partial^2 U}{\partial v \partial S}, \quad \frac{\partial^2 U}{\partial S^2}$$

Plug this into the Heston PDE Equation (4.14). All the S terms cancel and I obtain the Heston PDE in terms of the log price $x = \ln S$:

$$\begin{aligned} \frac{\partial U}{\partial t} + \frac{1}{2}\left(\frac{\partial^2 U}{\partial x^2}\right) + \left(r - \frac{1}{2}\nu\right)\frac{\partial U}{\partial x} + \rho\nu\frac{\partial^2 U}{\partial v \partial x} \\ + \frac{1}{2}\sigma^2\nu\frac{\partial^2 U}{\partial v^2} - rU + [\kappa(\theta - v) - \lambda]\frac{\partial U}{\partial v} = 0. \end{aligned} \tag{4.15}$$

where, just as in Heston (1993), I have written the market price of risk to be a linear function of the volatility, so that $\lambda(S, v, t) = \lambda v$.

4.2.2 European Call Price

In this section, I demonstrate how the call price in the Heston model can be expressed in a manner similar to the call price in the Black-Scholes model. Which sometimes gets dubbed “Black-Scholes-Like” or “a la Black-Scholes”. The time-t price of a European call on a non-dividend paying stock with spot price S_t, strike price K, and time to maturity $\tau = T - t$, is the discounted expected value of the payoff under the risk-neutral measure.

$$C(K) = e^{-r\tau}\mathbb{E}^Q\left[(S_T - K)^+\right] \tag{4.16}$$

This can be further expanded as:

$$C(K) = e^{-r\tau}\left[\mathbb{E}^Q[S_T \cdot 1_{\{S_T>K\}}] - K\mathbb{E}^Q[1_{\{S_T>K\}}]\right] \tag{4.17}$$

where 1 is the indicator function and Q is the risk-neutral measure.

To derive the expression for the first term in the last line, I need to transform the measure Q to another measure Q^S. Considering the Radon-Nikodym derivative:

$$\frac{dQ}{dQ^S} = \frac{B_T/B_t}{S_T/S_t} = \frac{\mathbb{E}^Q[e^{\int_0^t r du}]}{e^{rt}} \tag{4.18}$$

where B_t represents the money market account, satisfying
$B_t = \exp\left(\int_0^t r\, du\right) = e^{rt}$.

In (4.18), I denote $S_t e^{-(T-t)} = S_t e^{-rt}$, since under Q, assets grow at the risk-free rate r. The first expectation in equation (23) can be rewritten using Q^S:

$$\mathbb{E}^Q[S_T \cdot 1_{\{S_T>K\}}] = S_t \mathbb{E}^{Q^S}\left[\frac{S_T/S_t}{B_T/B_t} \cdot 1_{\{S_T>K\}}\right] \tag{4.19}$$

This simplifies to:

$$\mathbb{E}^Q[S_T \cdot 1_{\{S_T>K\}}] = S_t \mathbb{E}^{Q^S}[1_{\{S_T>K\}}] = S_t Q^S(S_T > K) \tag{4.20}$$

The second expectation in equation (4.17) is written as:

$$\mathbb{E}^Q[1_{\{S_T>K\}}] = Q(S_T > K) \tag{4.21}$$

Thus, the European call price in equation (4.17) can be expressed in terms of both measures as:

$$C(K) = S_t Q^S(S_T > K) - K e^{-r\tau} Q(S_T > K) \tag{4.22}$$

I denote $P_1 = Q^S(S_T > K)$ and $P_2 = Q(S_T > K)$, where the measure Q uses the bond B_t as the numeraire, while the measure Q^S uses the stock price S_t. The last line in the equation represents the "Black-Scholes-like" call price formula, with P_1 replacing $\Phi(d_1)$ and P_2 replacing $\Phi(d_2)$ in the Black-Scholes call price. The quantities P_1 and P_2 represent the probabilities of the call expiring in-the-money, conditioned on the value $S_t = e^{rt}$ of the stock and the value ν_t of the volatility at time t.

4.2.3 The PDE for P_1 and P_2

The call price C in equation (4.22) follows the PDE presented in Equation (4.15), but here I express it using the time derivative with respect to τ instead of t:

$$\begin{aligned}\frac{\partial C}{\partial \tau} + \frac{1}{2}\sigma^2\frac{\partial^2 C}{\partial x^2} + \left(r - \frac{1}{2}v\right)\frac{\partial C}{\partial x} + \rho v\frac{\partial^2 C}{\partial x \partial \nu} + \frac{1}{2}v^2\frac{\partial^2 C}{\partial \nu^2} \\ - rC + \left[\kappa(\theta - v) - \lambda v\right]\frac{\partial C}{\partial \nu} = 0.\end{aligned} \tag{4.23}$$

The derivatives of C from Equation (4.22) will be expressed in terms of P_1 and P_2. By substituting these derivatives into the PDE (4.23) and regrouping the common terms related to P_1, I set $K = 0$ and $S = 1$ to derive the PDE for P_1. Similarly, by regrouping terms related to P_2 and setting $S = 0, K = -1$, and $r = 0$, I obtain the PDE for P_2. For notational simplicity, I combine the PDEs for P_1 and P_2 into a single expression:

$$\begin{aligned}-\frac{\partial P_j}{\partial \tau} + \rho v\frac{\partial^2 P_j}{\partial x \partial \nu} + \frac{1}{2}\sigma^2\frac{\partial^2 P_j}{\partial x^2} + \frac{1}{2}v^2\frac{\partial^2 P_j}{\partial \nu^2} \\ + (r + u_j)\frac{\partial P_j}{\partial x} + (a - b_j v)\frac{\partial P_j}{\partial \nu} = 0\end{aligned} \tag{4.24}$$

where $j = 1, 2$ and the parameters are defined as $u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, a = \kappa\theta, b_1 = \kappa + \lambda - \rho\sigma$, and $b_2 = \kappa + \lambda$. This corresponds to Equation (12) in Heston (1993), but it is expressed in terms of τ rather than t. This adjustment accounts for the minus sign in the first term of Equation (4.23) above.

4.2.4 The Characteristic Functions

Heston assumes that the characteristic functions for the logarithm of the terminal stock price, $x = \ln S_T$, take the form

$$f_j(\phi, x, v) = \exp(C_j(\tau, \phi) + D_j(\tau, \phi)v + i\phi x) \tag{4.25}$$

where C_j and D_j are coefficients and $\tau = T - t$ is the time to maturity and $i = \sqrt{-1}$ is the imaginary unit. The characteristic functions f_j will follow the PDE in Equation (4.24). This follows from the Feynman-Kac theorem. Hence the PDE for the characteristic function is, from Equation (4.24),

$$\begin{aligned}-\frac{\partial f_j}{\partial \tau} + \rho v\frac{\partial^2 f_j}{\partial x \partial \nu} + \frac{1}{2}\sigma^2\frac{\partial^2 f_j}{\partial x^2} + \frac{1}{2}v^2\frac{\partial^2 f_j}{\partial \nu^2} \\ + (r + u_j)\frac{\partial f_j}{\partial x} + (a - b_j v)\frac{\partial f_j}{\partial \nu} = 0.\end{aligned} \tag{4.26}$$

To evaluate this PDE for the characteristic function, I need the following derivatives, which are straightforward to derive:

$$\begin{aligned}
\frac{\partial f_i}{\partial \tau} &= \left(\frac{\partial C}{\partial \tau} + \frac{\partial D_i}{\partial \tau} v\right) f_i, \\
\frac{\partial f_i}{\partial x} &= i\phi f_i, \\
\frac{\partial f_i}{\partial v} &= D_i f_i, \\
\frac{\partial^2 f_i}{\partial x^2} &= -\phi^2 f_i, \\
\frac{\partial^2 f_i}{\partial x \partial v} &= i\phi D_i f_i, \\
\frac{\partial^2 f_i}{\partial v^2} &= D_i^2 f_i.
\end{aligned}$$

Substitute these derivatives into Equation (4.26), drop the f_j terms, and rearrange to obtain two differential equations:

$$\frac{\partial D_j}{\partial \tau} = \rho\sigma i\phi D_j - \frac{1}{2}\sigma^2\phi^2 + \frac{1}{2}D_j^2 + u_j i\phi - b_j D_j \tag{4.27}$$

$$\frac{\partial C_j}{\partial \tau} = ri\phi + aD_j \tag{4.28}$$

These are Equations (A7) in Heston (Heston 1993). Heston specifies the initial conditions $D_j(0,\phi) = 0$ and $C_j(0,\phi) = 0$. The first equation in (4.28) is a Riccati equation in D_j, while the second is an ordinary differential equation (ODE) for C_j that can be solved using straightforward integration once D_j is obtained.

4.2.5 The Heston Riccati Equation

First I introduce the concept of a Riccati equation and show how to solve it and then show how to solve the Heston equation specifically.

The Riccati equation for $y(t)$ with coefficients $P(t)$, $Q(t)$, and $R(t)$ is defined as

$$\frac{dy(t)}{dt} = P(t) + Q(t)y(t) + R(t)y(t)^2. \tag{4.29}$$

The equation can be solved by considering the following second-order ODE for $w(t)$

$$w'' - \left[\frac{P'}{P} + Q\right] w' + PRw = 0 \tag{4.30}$$

which can be written as $w'' + bw' + cw = 0$. The solution to Equation (4.29) is then

$$y(t) = -\frac{w'(t)}{w(t)}\frac{1}{R(t)}. \tag{4.31}$$

The ODE in (4.30) can be solved via the auxiliary equation $r^2 + br + c = 0$, which has two solutions α and β given by

$$\alpha = \frac{-b + \sqrt{b^2 - 4c}}{2}, \quad \beta = \frac{-b - \sqrt{b^2 - 4c}}{2}. \tag{4.32}$$

The solution to the second-order ODE in (4.30) is

$$w(t) = Me^{\alpha t} + Ne^{\beta t} \tag{4.33}$$

where M and N are constants. The solution to the Riccati equation is therefore

$$y(t) = -\frac{M\alpha e^{\alpha t} + N\beta e^{\beta t}}{Me^{\alpha t} + Ne^{\beta t}}\frac{1}{R(t)}. \tag{4.34}$$

I will now show how to apply this to the Heston model.

From Equations (4.27) and (4.28) the Heston Riccati equation is

$$\frac{\partial D_j}{\partial \tau} = P_j - Q_j D_j + RD_j^2 \tag{4.35}$$

The related second-order ODE is

$$w'' + Q_j w' + P_j R = 0 \tag{4.36}$$

The solution to the Heston Riccati equation (40) is therefore

$$D_j = -\frac{1}{R}\left(\frac{K\alpha e^{\alpha\tau} + \beta e^{\beta\tau}}{Ke^{\alpha\tau} + e^{\beta\tau}}\right) \tag{4.37}$$

Using the initial condition $D_j(0, \phi) = 0$ results in the solution for D_j

$$D_j = \frac{b_j - \rho\sigma i\phi + d_j}{\sigma^2}\left(\frac{1 - e^{d_j\tau}}{1 - g_j e^{d_j\tau}}\right) \tag{4.38}$$

where

$$d_j = \sqrt{(\rho\sigma i\phi - b_j)^2 - \sigma^2(2u_j i\phi - \phi^2)} \tag{4.39}$$

$$g_j = \frac{b_j - \rho\sigma i\phi + d_j}{b_j - \rho\sigma i\phi - d_j} \tag{4.40}$$

The solution for C_j is obtained by integrating (4.28)). Thus

$$C_j = \int_0^\tau ri\phi d\tau + a(\frac{Q_j + d_j}{2R}) \int_0^\tau \left(\frac{1 - e^{d_j y}}{1 - g_j e^{d_j y}} \right) dy + K_1 \tag{4.41}$$

where K_1 is a constant. Integrating and applying the initial condition $C_j(0, \phi) = 0$, and substituting for d_j, Q_j, and g_j, I obtain the solution for C_j

$$C_j = ri\phi\tau + \frac{a}{\sigma^2}\left[(b_j - \rho\sigma i\phi + d_j)\tau - 2\ln\left(\frac{1 - g_j e^{d_j\tau}}{1 - g_j}\right)\right] \tag{4.42}$$

where $a = \kappa\theta$.

This ends the derivation of the Heston model and its characteristic equation, later in the thesis I will explain numerical methods for solving for the option price.

4.3 Heston versus Black-Scholes

In this section I will examine the advantages and disadvantages of the Heston model and compare it to the Black-Scholes model.

Many empirical studies have been conducted comparing the accuracy of the two models, such as in Wu (2019), which analyses over 3,000 S&P 500 Index European call options and concludes that the Heston Model for option pricing is more accurate than the Black-Scholes Model. Specifically that the Heston Model has a lower pricing error and thus improves its accuracy as the maturity gets longer so is better for predicting options for mid-term or long-term maturity, while Black-Scholes is better for short-term.

Similar results can be observed from Chakrabarti and Santra (2019) which has in-sample data of over 25,000 call options from the S&P Index. It concludes that for short term ITM, DITM and ATM options the Black-Scholes model outperforms the Heston model which struggles to capture the high-implied volatility, but increasing time-to-maturity leads to more accurate estimates. The Heston model performs significantly better in OTM and DOTM options since Black-Scholes gives significantly high error values.

Finally, Hao and Yin (2022) comes to similar conclusions. The Black-Scholes model is praised for its computational simplicity and accuracy for short-term options, while the Heston model again proved more accurate for long term options.

From the results studies and the mathematical composition of the two models the advantages and disadvantages of Black-Scholes and Heston in pricing European options become clear. Black-Scholes is historically significant and is the foundation for modern derivative pricing theory; it is essential as the majority of financial theories and models are built on the back of it. It is renowned for its simplicity to implement and its closed-form analytical solution of European call options is fast and computationally efficient; this is important in real-time trading environments. It has very few parameters and they are straightforward to estimate from market data. However, one of the model's major limitations is its assumptions of constant volatility which leads to reduced pricing accuracy as time-to-maturity increases. It also assumes log-normal distribution which fails to account for kurtosis. It is also unable to handle the volatility smile Derman and Miller (2016).

The Heston model's incorporation of stochastic volatility leads to a more realistic model that is able to capture the observed volatility smile in options markets. It is much more accurate for long term options and has more flexibility when it comes to modelling more complex markets. Parameter estimation is difficult and its accuracy directly affects pricing accuracy. It is at a disadvantage due to its complexity and computational intensity.

The choice between the two models depends on the needs of the user and the characteristics of the market. In later sections of this thesis I will be attempting to counteract some of the disadvantages of the Heston model.

Chapter 5

Pricing Methods

In Finance, when working with complex models such as the Heston model, it is sometimes impossible or impractical to obtain option prices analytically. In this case, we must use numerical methods to price derivatives. This is how practitioners get approximate solutions when closed-form expressions are unavailable.

This section will discuss three of the most fundamental numerical methods in the Heston model: Monte Carlo Simulation, Fourier Transformation and the Finite Difference Method. The Finite Difference Method will be discussed at length in this section while Monte Carlo and Fourier Transformation will be presented as an overview of key concepts. Each method has its strengths and weaknesses and are better suited to different application cases; comparisons between them—in particular between the FDM and Fourier methods—will therefore be offered.

5.1 Monte Carlo Simulation

The Monte Carlo Simulation method has gone under many names such as simply "Statistical Sampling". The name Monte Carlo was popularised by physicist Stanislaw Ulam as a reference to the famous casino in Monaco where his uncle used to gamble. The method was famously used by Enrico Fermi to calculate the properties of the neutron using random sampling (Fermi 1935). The method was used in the development of nuclear weapons to simulate the behaviour of particles. The majority of the information in this section is presented in (Oosterlee & Grzelak 2019). For more detailed information on the financial applications see (Boyle, Broadie & Glasserman 1997) and (Broadie & Kaya 2006). In this section I will briefly provide an explanation of the theory behind the Monte Carlo Simulation and show how it can be used in the Heston model.

Monte Carlo methods are based on concepts in probability theory such as the Central Limit Theorem and the Law of Large Numbers. It involves a Monte Carlo integration which takes the area of the space and multiplies it by the fraction of randomly generated points under the curve to compute the integral. This method of integration is used in Finance for stochastic integrals, allowing the discretisation of the Brownian motion to find a convergence. This convergence strongly depends on the smoothness of the function.

We must take advantage of discretisation techniques. The most common of which are known as the Euler scheme and Milstein scheme. Since in this context we are discussing the Heston model I must discresitise both the asset price and the variance processes. The variance is modelled by a Cox-Ingersol-Ross process meaning we face challenges with these standard discretisation schemes so must employ advanced techniques to avoid this. The problem we face is one of non-negativity which can be dealt with by using either a Truncated Euler scheme or a Reflecting Euler scheme which use max or absolute functions respectively to avoid negative values.

A different approach is to use an exact simulation of the CIR model making use of the non-central chi-squared distribution. This allows the creation of an almost exact discretisation of the Heston Model which will be presented here.

The Heston model we know as:

$$\begin{cases} dS_t = rS_t dt + S_t\sqrt{V_t}dW_t^S, \\ dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t}dW_t^V, \end{cases} \tag{5.1}$$

Which for $X(t) := -\log S(t)$, becomes:

$$\begin{cases} dX_t = \left(r - \frac{1}{2}V_t\right)dt + \sqrt{V_t}dW_t^S, \\ dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t}dW_t^V, \end{cases} \tag{5.2}$$

which I present in terms in independent Brownian motions:

$$\begin{cases} dX_t = \left(r - \frac{1}{2}V_t\right)dt + \sqrt{V_t}\left[\rho_{x,v}d\widetilde{W}_t^V + \sqrt{1-\rho_{x,v}^2}d\widetilde{W}_t^x\right], \\ dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t}d\widetilde{W}_t^V, \end{cases} \tag{5.3}$$

After integration of both processes in a certain time interval $[t_i, t_{i+1}]$ I get the following discretisation scheme:

$$\begin{aligned} x_{i+1} = x_i &+ \int_{t_i}^{t_{i+1}} \left(r - \frac{1}{2}v_t\right)dt + \rho_{x,v}\int_{t_i}^{t_{i+1}} \sqrt{v_t}d\widetilde{W}_t^V \\ &+ \sqrt{1-\rho_{x,v}^2}\int_{t_i}^{t_{i+1}} \sqrt{v_t}d\widetilde{W}_t^x, \end{aligned} \tag{5.4}$$

and

$$v_{i+1} = v_i + \kappa\int_{t_i}^{t_{i+1}} (\theta - v_t)\,dt + \sigma\int_{t_i}^{t_{i+1}} \sqrt{v_t}d\widetilde{W}_t^V. \tag{5.5}$$

Notice that the two integrals with $\widetilde{W}_t^V$ in the SDEs above are the same, and in terms of the variance realisations they are given by:

$$\int_{t_i}^{t_{i+1}} \sqrt{v_t} d\widetilde{W}_t^V = \frac{1}{\sigma}\left(v_{i+1} - v_i - \kappa \int_{t_i}^{t_{i+1}} (\theta - v_t) dt\right). \tag{5.6}$$

The variance v_{i+1} is then simulated, for v_i, using the CIR process. The discretisation for x_{i+1} is given by:

$$\begin{aligned} x_{i+1} = x_i &+ \int_{t_i}^{t_{i+1}} \left(r - \frac{1}{2}v_t\right) dt \\ &+ \frac{\rho_{x,v}}{\sigma}\left(v_{i+1} - v_i - \kappa \int_{t_i}^{t_{i+1}} (\theta - v_t) dt\right) \\ &+ \sqrt{1 - \rho_{x,v}^2} \int_{t_i}^{t_{i+1}} \sqrt{v_t} d\widetilde{W}_t^x. \end{aligned} \tag{5.7}$$

I approximate all integrals appearing in the expression above by their left integration boundary values of the integrand, as in the Euler discretisation scheme:

$$\begin{aligned} x_{i+1} \approx x_i &+ \int_{t_i}^{t_{i+1}} \left(r - \frac{1}{2}v_i\right) dt \\ &+ \frac{\rho_{x,v}}{\sigma}\left(v_{i+1} - v_i - \kappa \int_{t_i}^{t_{i+1}} (\theta - v_i) dt\right) \\ &+ \sqrt{1 - \rho_{x,v}^2} \int_{t_i}^{t_{i+1}} \sqrt{v_i} d\widetilde{W}_t^x. \end{aligned} \tag{5.8}$$

The calculation of the integrals results in:

$$\begin{aligned} x_{i+1} \approx x_i &+ \left(r - \frac{1}{2}v_i\right) \Delta t \\ &+ \frac{\rho_{x,v}}{\sigma}\left(v_{i+1} - v_i - \kappa(\theta - v_i)\Delta t\right) \\ &+ \sqrt{1 - \rho_{x,v}^2}\sqrt{v_i}\left(\widetilde{W}_{t_{i+1}}^x - \widetilde{W}_{t_i}^x\right). \end{aligned} \tag{5.9}$$

Finally, for $\widetilde{W}_{t_{i+1}}^x - \widetilde{W}_{t_i}^x = \sqrt{\Delta t} Z_x$, with $Z_x \sim \mathcal{N}(0,1)$, we find:

$$\begin{cases} x_{i+1} \approx x_i + k_0 + k_1 v_i + k_2 v_{i+1} + \sqrt{k_3 v_i} Z_x, \\ v_{i+1} = \bar{c}(t_{i+1}, t_i)\chi^2(\delta, \bar{\kappa}(t_{i+1}, t_i)), \end{cases} \tag{5.10}$$

with the variance process simulated as follows:

$$
\begin{aligned}
\bar{c}(t_{i+1}, t_i) &= \frac{\sigma^2}{4\kappa}\left(1 - e^{-\kappa(t_{i+1}-t_i)}\right), \\
\delta &= \frac{4\kappa\theta}{\sigma^2}, \\
\bar{\kappa}(t_{i+1}, t_i) &= \frac{4\kappa e^{-\kappa\Delta t}}{\sigma^2\left(1 - e^{-\kappa\Delta t}\right)} v_i,
\end{aligned}
\tag{5.11}
$$

and $\chi^2(\delta, \bar{\kappa}(.,.)$ the noncentral chi-squared distribution with δ degrees of freedom and noncentrality parameter $\bar{\kappa}(t_{i+1}, t_i)$.

$$
\begin{aligned}
k_0 &= \left(r - \frac{\rho_{x,v}}{\sigma}\kappa\theta\right)\Delta t, \\
k_1 &= \left(\frac{\rho_{x,v}}{\sigma}\kappa - \frac{1}{2}\right)\Delta t, \\
k_2 &= \frac{\rho_{x,v}}{\sigma}, \\
k_3 &= (1 - \rho_{x,v}^2)\Delta t.
\end{aligned}
\tag{5.12}
$$

To begin the simulation one must choose parameters κ, σ, θ, r and $\rho_{x,v}$. Simulation numbers often run into the hundreds of thousands to achieve accurate results. The advantages of the Monte Carlo method are its flexibility to handle the stochastic volatility and relative simplicity. However it can be very computationally intensive and therefore expensive. For these reasons I chose to omit data generation using the Monte Carlo Simulation in favour of more efficient methods.

5.2 Fourier Transformation

In this section I summarise how you can solve the Heston model using a Fourier transformation. This was popularised by Carr and Madan (1999), whose methodology I will be following, where a full explanation of the model can be found. I will show a reminder of the Heston characteristic function derived earlier and show how the Fourier transformation can serve to derive the option price.

The characteristic function is defined according to equation 4.25. For $x = \ln S_T$, the Heston characteristic function $\phi(u)$ is defined as:

$$
\phi(u) = \mathbb{E}\left[e^{iux}\right] = \exp\left(C(u,\tau) + D(u,\tau)v_0 + iu\ln S_0\right),
\tag{5.13}
$$

with C and D defined as before. Note that the earlier simplification of $a = k\theta$ has been revoked. This characteristic function serves as the basis for Fourier transformation.

5.2.1 Fundamentals of Fourier Transformation

The Fourier transform is a mathematical operation that decomposes a function into its constituent frequencies, for more information see Bracewell (1999). For a given function $f(x)$, its Fourier transform $\mathcal{F}\{f\}(k)$ is defined as:

$$\mathcal{F}\{f\}(k) = \int_{-\infty}^{\infty} f(x)e^{-ikx}\, dx, \tag{5.14}$$

where k represents the frequency variable. Conversely, the inverse Fourier transform, which reconstructs the original function from its frequency-domain representation, is given by:

$$\mathcal{F}^{-1}\{\mathcal{F}\{f\}\}(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathcal{F}\{f\}(k)e^{ikx}\, dk. \tag{5.15}$$

Several key properties of the Fourier transform make it useful in option pricing. Linearity holds for the Fourier transform of a combination of functions. Shifting a function in the time domain corresponds to a modulation in the frequency domain. The Fourier transform of a convolution of two functions is the point-wise product of their Fourier transforms, which follows from the convolution theorem (Bracewell 1999).

These properties allow for the manipulation and combination of transformed functions in ways that simplify complex integrals and differential equations, making the Fourier transform a valuable tool in finance.

5.2.2 Applications to the Heston model

In the context of the Heston model, the Fourier transform serves as a bridge between the characteristic function and the probability distribution of the underlying asset price. The inversion of the characteristic function via the Fourier transform allows the retrieval of option prices through integral representations.

We need to express the probability density function $f(x)$ of the logarithm of the terminal stock price x in terms of its characteristic function $\phi(u)$:

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-iux}\phi(u)\, du. \tag{5.16}$$

This is making use of the inversion formula above.

To price a European call option with strike price K and maturity T, the risk-neutral valuation formula requires the computation of the expectation $\mathbb{E}^{\mathbb{Q}}[(S_T - K)^+]$. Utilizing the Fourier transform, this expectation can be expressed in terms of integrals involving the characteristic function.

The option price C is given by:

$$C = e^{-r\tau}\mathbb{E}^{\mathbb{Q}}[(S_T - K)^+] = e^{-r\tau}\left(S_0 P_1 - K P_2\right), \tag{5.17}$$

where P_1 and P_2 are probabilities corresponding to different measures under the risk-neutral framework that are defined as before. These probabilities are retrieved via the Fourier inversion of the characteristic function:

$$P_j = \frac{1}{2} + \frac{1}{\pi}\int_0^\infty \text{Re}\left[\frac{e^{-iu\ln K}\phi_j(u)}{iu}\right] du, \quad j = 1, 2. \tag{5.18}$$

Here, $\phi_j(u)$ represents the characteristic functions associated with P_j, incorporating specific damping factors to ensure integrability and convergence of the integral as can be seen in Carr and Madan (1999).

Substituting P_1 and P_2 into the option price formula consolidates the expression into a single integral:

$$C = \frac{1}{2}\left(S_0 - Ke^{-r\tau}\right) + \frac{1}{\pi}\int_0^\infty \text{Re}\left[\frac{e^{-iu\ln K}\left(S_0\phi_1(u) - Ke^{-r\tau}\phi_2(u)\right)}{iu}\right] du. \tag{5.19}$$

This integral formulation allows for efficient computation of option prices within the Heston model framework.

5.3 Finite Difference Method

In this section, I focus on solving the Heston PDE numerically using the Finite Difference Method (FDM). The Heston model leads to a two-dimensional PDE with respect to the asset price and variance. To numerically approximate the price of a European option, I discretize the Heston PDE on a spatial and temporal grid. We follow the explanation given by Guerrero Torres (2019).

The Heston PDE was outlined in equation 4.15. I present it here for the value $V(S, v, t)$ of an option with asset price S, variance v, and time to maturity $\tau = T - t$, is given by:

$$-\frac{\partial V}{\partial \tau} + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2} + \rho\sigma vS\frac{\partial^2 V}{\partial S\partial v} + \frac{1}{2}\sigma^2 v\frac{\partial^2 V}{\partial v^2} + rS\frac{\partial V}{\partial S} + \kappa(\theta - v)\frac{\partial V}{\partial v} - rV = 0, \tag{5.20}$$

where:

- S is the asset price,

- v is the variance,
- r is the risk-free interest rate,
- σ is the volatility of volatility,
- ρ is the correlation between the asset price and variance processes,
- κ is the rate of mean reversion, and
- θ is the long-term mean variance.

Using $V(t) = V(S, v, t)$ as compact notation we can express the PDE (4.1) as:

$$\frac{\partial V}{\partial t} = LV(t) \tag{5.21}$$

where the operator L is defined as:

$$L = \frac{1}{2}v\frac{\partial^2}{\partial x^2} + \left(r - \frac{v}{2}\right)\frac{\partial}{\partial x} + \frac{1}{2}\sigma^2 v\frac{\partial^2}{\partial v^2} + \rho\sigma v\frac{\partial^2}{\partial v \partial x} - r + \kappa(\theta - v)\frac{\partial}{\partial v} \tag{5.22}$$

The goal is to develop a stable and accurate numerical method to approximate the solution to this PDE. To achieve this, I employ a finite difference discretisation in both the asset price and variance dimensions, along with the Crank-Nicolson time-stepping scheme to solve the PDE over a grid of points.

In the following sections, I outline the grid construction and present the finite difference approximations for the spatial derivatives in the Heston PDE.

5.3.1 Non-uniform grids

Numerical methods for PDE's, such as the FDM, require the domain of the PDE to be discretised into a grid. A uniform grid divides the domain into equally spaced intervals, but for some problems, such as option pricing using the Heston model, a non-uniform grid is often preferable. This allows for more grid points to be concentrated in areas where higher accuracy is required, such as near the strike price or at lower volatility levels.

In the case of option pricing, accuracy is particularly important near the strike price, where the option's value is most sensitive to changes in the underlying asset price, and at low volatility levels. To achieve this, a non-uniform grid is used, where grid points are more concentrated in these critical areas. This is achieved by mapping a uniform grid onto a non-uniform one using a transformation function, $g(x)$, that determines how the grid points are distributed.

The transformation function $g(x)$ maps the uniform grid points x_i to non-uniform grid points y_i. Given a uniform grid $x_i = \frac{i}{n}$ for $i = 0, 1, \dots, n$, the non-uniform grid points y_i are obtained by applying the transformation $g(x)$, such that:

$$y_i = cg(x_i) + d, \quad i = 1, 2, \ldots, n \tag{5.23}$$

where c and d are scaling and translation parameters, respectively. The transformation function $g(x)$ must be continuously differentiable, bijective, and strictly monotonic.

The distance between two adjacent points in the non-uniform grid is determined by the ratio of the distance between the corresponding points in the uniform grid. This ratio is characterised by the distance ratio function $r(y)$, defined as:

$$r(y) = \frac{dg(g^{-1}(y))}{dx} \tag{5.24}$$

The transformation function $g(x)$ must satisfy the ordinary differential equation (ODE):

$$g'(x) = r(g(x)) \tag{5.25}$$

This ODE can be solved by the separation of variables approach:

$$\int_0^{g(x)} \frac{dg(z)}{r(g(z))} dz = \int_0^x dz \tag{5.26}$$

To create a non-uniform grid, I can specify a distance ratio function $r(y)$ that concentrates points around a certain region of interest. One such example is:

$$r(y) = \sqrt{c^2 + p^2(y - y^*)^2} \tag{5.27}$$

where y^* is the center of the grid concentration and c controls the intensity of the concentration. For large values of y, the function becomes almost linear. The parameter p is chosen to ensure the density function's property is satisfied, and the transformation function $g(x)$ can be found by solving the ODE.

The solution to the ODE for $g(x)$ is obtained by integrating the following equation:

$$\int_0^{g(x)} \frac{dy}{\sqrt{c^2 + p^2(y - y^*)^2}} = x \tag{5.28}$$

This is solved via differentiation and making use of the hyperbolic trigonometric identities. For a full derivation see Guerrero Torres (2019).

$$g(x) = \frac{c}{p} \sinh\left[px + \sinh^{-1}\left(\frac{-py^*}{c}\right)\right] + y^* \tag{5.29}$$

5.3.2 Space Discretisation

I denote the upper bounds for the asset price S, variance v, and time t as $S_{\max}$, $v_{\max}$, and $t_{\max} = \tau$ (the maturity), and the lower bounds as $S_{\min}$, $v_{\min}$, and $t_{\min} = 0$. I denote by $V_{i,j}^n = V(S_i, v_j, t_n)$ the value of a European call at time t_n when the stock price is S_i and the volatility is v_j. I employ $N_S + 1$ grid points for the asset price, $N_v + 1$ grid points for the variance, and $N_T + 1$ grid points for the time to maturity. For simplicity, I may sometimes write $V(S_i, v_j)$ instead of $V_{i,j}^n$.

Taking the lower bounds $S_{\min} = v_{\min} = 0$ as reference and following the approach of Kluge (2003), I implement a non-uniform grid that concentrates more grid points around the strike price K and near the initial volatility $v_0 = 0$. The grid for the asset price S of size $N_S + 1$ is constructed as:

$$S_i = c \sinh\left(i\Delta\zeta + \sinh^{-1}\left(\frac{-K}{c}\right)\right) + K, \quad i = 0, 1, \ldots, N_S \tag{5.30}$$

with

$$\Delta\zeta = \frac{1}{N_S}\left(\sinh^{-1}\left(\frac{S_{\max} - K}{c}\right) - \sinh^{-1}\left(\frac{-K}{c}\right)\right) \tag{5.31}$$

Equation (5.20) follows from Kulge (2003), where $y^* = p = K$, with the parameter c governing the concentration of grid points in the vicinity of the strike price K. I set $c = \frac{K}{5}$, following the experiment in Hout and Foulon (2010).

I construct a non-uniform mesh in the v-direction in a similar fashion to that of the S-direction. Let $d > 0$ be a constant. The grid for the volatility, consisting of $N_v + 1$ points, is given by:

$$v_j = d \sinh(j\Delta\eta), \quad j = 0, 1, \ldots, N_v, \tag{5.32}$$

where

$$\Delta\eta = \frac{1}{N_v} \sinh^{-1}\left(\frac{v_{\max}}{d}\right). \tag{5.33}$$

In Hout and Foulon (2010), the parameter d is chosen as $d = \frac{v_{\max}}{500}$, and a uniform grid is used for the time variable t. Figure 4.1 illustrates the non-uniform grid generated using these settings, where $N_S = 50$, $N_v = 25$, $S_{\max} = 8$, $v_{\max} = 5$, and $K = 1$. In this figure, the grid for the stock price is represented by the horizontal points, while the grid for the volatility is represented by the vertical points. In the asset price dimension, the grid is finest around the strike price K, whereas in the volatility dimension, the grid becomes finer as the volatility approaches zero.

5.3.3 Derivatives

The next step in the method is to approximate the derivatives in equation 5.14, I follow the methodology presented in Kluge (2003) in the non-uniform case, where a full derivation

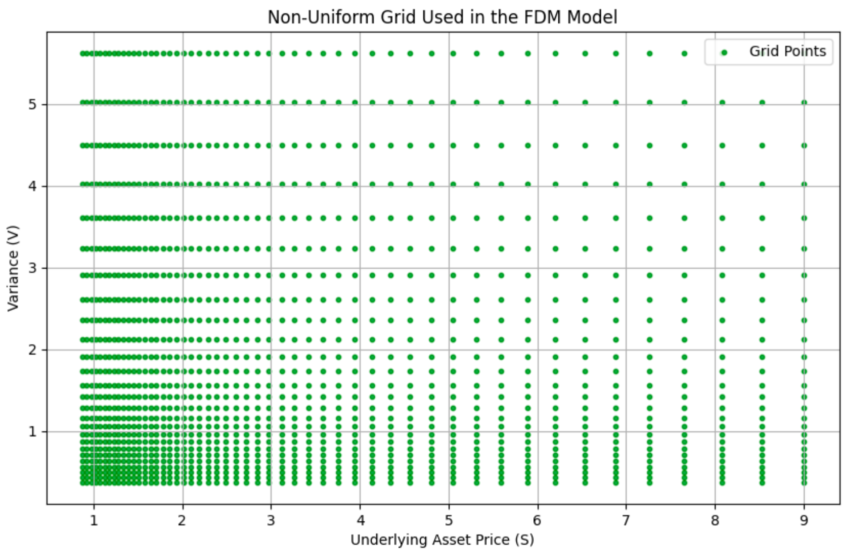


Figure 5.1: Non-uniform grid

Source: generated in Python

of the formulae is available. I will provide a summary of the results.

The results follow from approximating the derivatives in Operator L using the following equation, known as the central difference scheme:

$$\frac{\partial V}{\partial S}(S_i) \approx \sum_{k=-1}^{1} a_k V(S_{i+k}) \tag{5.34}$$

Applying Taylor series expansion around grid point S_i to form backward and forward differences, yields a series of linear equations. Which through careful substitution can be solved, the results are summarised in the following table:

a_k	V	V'	V''
a_{-1}	0	$\frac{-\Delta S_{i+1}}{\Delta S_i(\Delta S_i+\Delta S_{i+1})}$	$\frac{2}{\Delta S_i(\Delta S_i+\Delta S_{i+1})}$
a_0	1	$\frac{\Delta S_{i+1}-\Delta S_i}{\Delta S_i \Delta S_{i+1}}$	$\frac{-2}{\Delta S_i \Delta S_{i+1}}$
a_1	0	$\frac{\Delta S_i}{\Delta S_{i+1}(\Delta S_i+\Delta S_{i+1})}$	$\frac{2}{\Delta S_{i+1}(\Delta S_i+\Delta S_{i+1})}$

Table 5.1: Central difference scheme

This does not work on boundaries, so I must use left and right side approximation schemes. The right hand side approximation makes use of the following equation:

$$\frac{\partial V}{\partial S}(S_i) \approx \sum_{k=0}^{2} a_k V(S_{i+k}) \tag{5.35}$$

and for the left hand side:

$$\frac{\partial V}{\partial S}(S_i) \approx \sum_{k=-2}^{0} a_k V(S_{i+k}) \tag{5.36}$$

Using the same method as before I get systems of equations for each which once again I summarize in the tables:

a_k	V	V'	V''
a_0	1	$\frac{-\Delta S_{i+1}}{\Delta S_i(\Delta S_i+\Delta S_{i+1}+\Delta S_{i+2})}$	$\frac{2}{\Delta S_i(\Delta S_i+\Delta S_{i+1}+\Delta S_{i+2})}$
a_1	0	$\frac{\Delta S_{i+1}-\Delta S_i}{\Delta S_i\Delta S_{i+1}}$	$\frac{-2}{\Delta S_i\Delta S_{i+1}}$
a_2	0	$\frac{\Delta S_i}{\Delta S_{i+1}(\Delta S_i+\Delta S_{i+1}+\Delta S_{i+2})}$	$\frac{2}{\Delta S_{i+1}(\Delta S_i+\Delta S_{i+1}+\Delta S_{i+2})}$

Table 5.2: Right approximation of derivatives

a_k	V	V'	V''
a_{-2}	0	$\frac{\Delta S_i}{\Delta S_{i-1}(\Delta S_{i-1}+\Delta S_i)}$	$\frac{2}{\Delta S_{i-1}(\Delta S_{i-1}+\Delta S_i)}$
a_{-1}	0	$\frac{-\Delta S_i}{\Delta S_{i-1}(\Delta S_{i-1}+\Delta S_i)}$	$\frac{-2}{\Delta S_{i-1}\Delta S_i}$
a_0	1	$\frac{-\Delta S_{i-1}}{2\Delta S_i\Delta S_{i-1}}$	$\frac{2}{\Delta S_i(\Delta S_{i-1}+\Delta S_i)}$

Table 5.3: Left approximation of derivatives

5.3.4 Crank-Nicolson Scheme

We need to choose an effective numerical time-discretisation scheme. Many are outlined in Hout and Foulon (2010) such as the Douglas, Craig-Sneyd or Hundesdorfer-Verwer scheme. I choose the most common which is known as the Crank-Nicolson scheme (Crank & Nicolson 1947). This is also known as the weighted method, setting $\theta = 0.5$ in the following method gives the Crank-Nicolson scheme.

In this method I constuct a vector V^n of size N where N is the size of the grid I made $N = (N_S + 1)(N_V + 1)$. Our vector is $V^n = \left(\nu_0^T, \nu_1^T, \ldots, \nu_{N_v}^T\right)^T$ whose entries correspond to the points in the grid.

The method is defined defined via the relationship:

$$\frac{V^{n+1} - V^n}{\Delta t} = L\left(\theta V^{n+1} + (1-\theta)V^n\right) \tag{5.37}$$

where L is a matrix of dimension $N \times N$ based on the operator defined in equation (5.14). Therefore I can work from V^0, the value of the call at expiry, and use L to obtain V^1, V^2, until I reach V^{N^t}. This is achieved by solving the system:

$$(I - \Delta t\theta L)V^{n+1} = (I + (1-\theta)\Delta t L)\, V^n \tag{4.13}$$

where I is the identity matrix of size N. The system can be solved by inverting the left hand side matrix so that:

$$V^{n+1} = (I - \Delta t \theta L)^{-1} (I + (1-\theta)\Delta t L) V^n$$

Here I set $\theta = 0.5$ as discussed earlier and form $L = A_0 + A_1 + A_2$, with each A being a matrix of size N x N. A_0 contains all entries of L corresponding to the mixed derivative $\frac{\partial^2 V}{\partial S \partial \nu}$, A_1 contains all entries corresponding to $\frac{\partial V}{\partial S}$ and $\frac{\partial^2 V}{\partial S^2}$, and A_2 contains all entries corresponding to $\frac{\partial V}{\partial \nu}$ and $\frac{\partial^2 V}{\partial \nu^2}$. The matrices are constucted as:

$$A_0 = \sigma \rho \nu S \left(\frac{\partial^2 V}{\partial S \partial \nu} \right)_{N \times N}, \tag{5.38}$$

$$A_1 = rS \left(\frac{\partial V}{\partial S} \right)_{N \times N} + \frac{1}{2} \nu S^2 \left(\frac{\partial^2 V}{\partial S^2} \right)_{N \times N} - \frac{1}{2} r V(N)_{N \times N}, \tag{5.39}$$

$$A_2 = \kappa(\theta - \nu) \left(\frac{\partial V}{\partial \nu} \right)_{N \times N} + \frac{1}{2} \sigma^2 \nu \left(\frac{\partial^2 V}{\partial \nu^2} \right)_{N \times N} - \frac{1}{2} r V(N)_{N \times N}. \tag{5.40}$$

This concludes the construction of the finite difference method the implementation of which can be found in *Appendix A*. The method involves the construction of a non-uniform grid for S_0 and v_0, spacial discretisation using central, forward and backward difference schemes, construction of matrices and the Crank-Nicolson scheme.

5.4 Comparison

The three methods of pricing options discussed here all have their respective advantages and disadvantages. The Monte Carlos simulation is highly flexible and relatively easy to implement however it is computationally intensive. The Fourier method is fast and effective but can be complicated to implement. The Fourier method is also limited to European options, which in this case is irrelevant. The Finite Difference Method is versatile and relatively computationally inexpensive, although it is complex to implement. For these reasons in the Methodology section of this thesis you will see I endeavoured to implement the Fourier Method and the Finite Difference Method, I will explain my implementation and full code scripts will be available in *Appendix A*.

Chapter 6

Artificial Neural Networks

Artificial Neural Networks are models that are inspired by the neural architecture of the biological brain, attempting to reverse engineer the computational principles and mimic its functionality. The aim of this section is to provide an detailed introduction to artificial neural networks and a brief discussion of the history of research into Deep Learning. I will then examine the architecture of neural networks and the mathematics behind it. Finally I will show how they can be applied in the context of option pricing in the Heston Model.

Deep Learning has an extensive history of research but has come to the forefront of people's attention in recent years with the rise in popularity of artificial intelligence. In fact Deep Learning has gone under many names. Model's of brain function can be found as early as The McCulloch-Pitts Neuron in 1943 (McCulloch & Pitts 1943) and the Perceptron in 1958 (Rosenblatt 1958), this was known as Cybernetics. There was connectionism in the 1980's which was inspired by advancements in neuroscience and theorised connecting a large number of computational units to achieve intelligent behavior. It wasn't until Hinton, Osindero and Teh (2006) where the term "Deep Learning" gained prominence. Deep Learning also goes by the name "Artificial Neural Networks" which for notation purposes I will usually refer to as ANNs in this thesis. The recent rapid development's in the effectiveness of ANNs are due to an increase in available Dataset sizes as well as the ability to expand the size of the model. This has been made possible by the inter-connectivity provided by the internet and the optimisation of computational power with better CPUs and GPUs.

ANNs are currently at the forefront of research in every sector from agriculture to medicine. They are very important in Finance for a multitude of reasons. The ability to process high volumes of data efficiently and adaptive learning allow ANNs to run in fast paced environments such as the stock market. The ability to recognise complex patterns serves to generate accurate predictions for stock prices, risk or exchange rates. In this thesis I will be looking at ANNs used to estimate the parameters of the Heston Model, hoping to mitigate the previously discussed disadvantages of the Heston Model.

6.1 Architecture

The basic building block for artificial neural networks is the neuron, they act as computational units through which data is processed. Many of these connected neurons can work together to solve complex tasks, similar to the neurons in the brain. Each neuron performs an operation on the data such as determining whether it is positive or negative. Here is a simplified diagram of a single neuron:

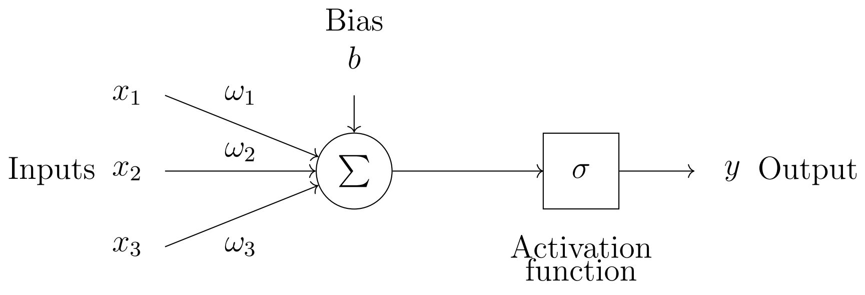


Figure 6.1: Diagram of a simple neural network node

Source: created using TikZ in LaTeX

A neuron receives one or more inputs (x_i), processes them through an activation function, and then produces an output. The inputs are weighted (w_i), which signifies the importance of each input in the computation. The output is the summation of the inputs multiplied by their corresponding weights. A bias (b) is also applied to the summation, and is important because without it, a layer wouldn't be able to produce an output that differs from 0 if the inputs were 0. If we denote the activation function of the neuron as σ and the output as y, then the formula for the output of a single neuron is:

$$y = \sigma\left(\sum_i (\omega_i x_i) + b\right)$$

Now that I have defined a singular neuron we are ready to introduce the concept of layers which form the structure of the neural network. A layer is a set of neurons all sharing the same inputs. A layer has the same number of outputs y_i as it has neurons. Below is a diagram of a two neuron layer:

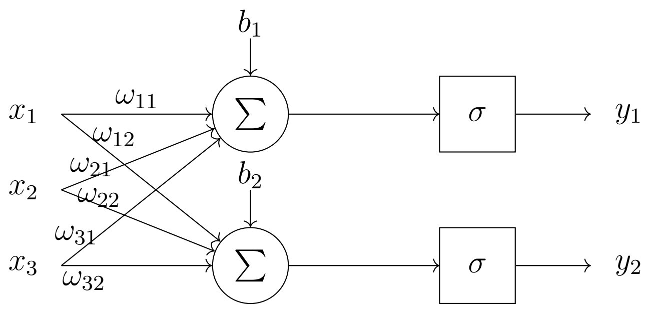


Figure 6.2: Diagram of a simple neural network with two neurons

Source: created using TikZ in LaTeX

From this we are able to introduce adjacent layers, networks like this are known as "Multi-layer Perceptrons" MLP. The first layer is the input layer is the point at which data enters the network. The next layers are known as hidden layers, which receive data from the layer before, processes it and passes the result onto the next layer in the network. This is where the majority of learning takes place. The final layer is the output layer. An N-layered neural network doesn't count the input layer, so a three-layer neural network consists of two hidden layers and the output:

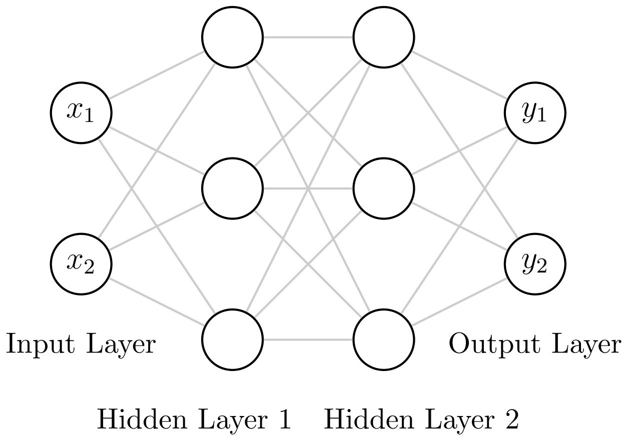


Figure 6.3: Diagram of a neural network with two hidden layers

Source: created using TikZ in LaTeX

The architecture of the Neural Network is defined by two factors: Depth and Width. The Depth is the number of hidden layers in the network and enhances the networks ability to model complex functions. The Width is the number of neurons in each layer and enhances the ability to capture detail. The simplest version of an ANN is an MLP with one hidden layer. Mathematically we define an MLP taking inspiration from a derivation found in Liu and Bohte (2019):

$$\theta = \{W_1, b_1, W_2, b_2, \dots, W_L, b_L\} \tag{6.1}$$

where W_j is a weight matrix with b_j being a bias vector in the L-th layer. A function can then be expressed as follows,

$$y(x) = F(x|\theta). \tag{6.2}$$

Let $z_j^{(l)}$ denote the value of the j-th neuron in the l-th layer, then the corresponding transfer function is defined:

$$z_j^{(l)} = \sigma^{(l)}\left(\sum_i w_{ij}^{(l)} z_i^{(l-1)} + b_j^{(l)}\right), \tag{6.3}$$

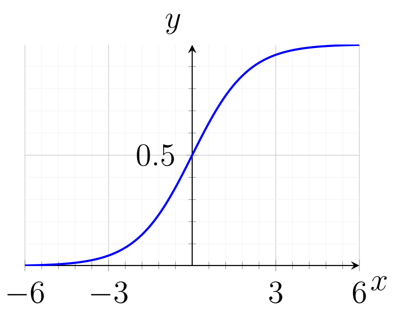
where:

- $\sigma^{(l)}(\cdot)$: The activation function applied in the l-th layer.
- $\sum_i$: The summation over all neurons i in the previous $(l-1)$-th layer.
- $w_{ij}^{(l)}$: The weight applied to the connection between the i-th neuron in the $(l-1)$-th layer and the j-th neuron in the l-th layer.
- $z_i^{(l-1)}$: The output value of the i-th neuron in the $(l-1)$-th layer, which serves as an input to the j-th neuron in the l-th layer.
- $b_j^{(l)}$: The bias term for the j-th neuron in the l-th layer.

There are different types of neural networks for different types of problems. The simplest is what I have defined here and is known as a Feedforward Neural Network (FNNs), which simply takes the output of each layer as the input into the next layer. FNNs are typically used for tasks like regression. Recurrent Neural Networks (RNNs) allow for cycles so information can be kept across steps, these are useful for tasks such as time series forcasting in Finance. Finally, Convolutional Neural Networks (CNNs) which were originally designed for image processing make use of convolutional layers and can be useful with spacial or temporal patterns.

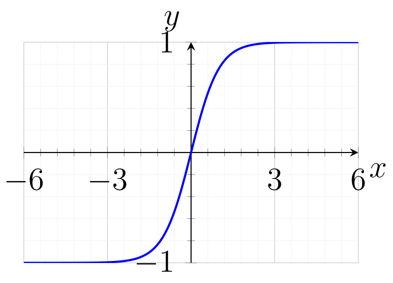
6.2 Activation Functions

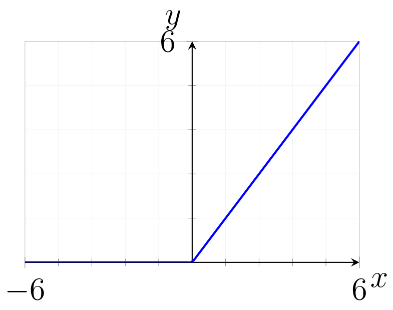
Activation functions (σ) are what allow for neural networks to model complex real-world data. Without them a neural network would behave as a linear model. The activation function transforms the input into an output to pass on to the next layer. Here I will outline some of the most common examples. For more information and examples of activation functions see LeCun, Bengio and Hinton (2015).



Sigmoid Function
$\sigma(x) = \frac{1}{1+e^{-x}}$
It maps any real-valued number to a value between 0 and 1. Commonly used as an output layer for binary classification models.

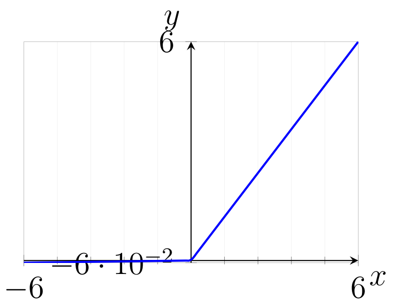
Tanh Function
$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
It maps real-valued numbers to a range between -1 and 1. Note that tanh is a rescaled sigmoid: $\tanh(x) = 2\sigma(2x) - 1$.

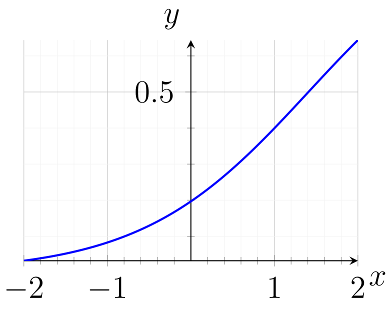




ReLU Function
$\text{ReLU}(x) = \max(0, x)$
It outputs the input directly if positive, otherwise 0.

Leaky ReLU Function
$\text{Leaky ReLU}(x) = \max(0.01x, x)$
It allows a small, non-zero gradient when the input is negative. It accounts for the 'Dying ReLU' problem where standard ReLU can get stuck if it outputs zero.





Softmax Function
$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
It maps input scores to probabilities that sum to 1.
It is useful for probability distribution in classification tasks.

Figure 6.4: Activation functions and their properties
Source: created using TikZ in LaTeX

6.3 Loss Function

The Universal Approximation Theorem originally stated in Cybenko (1989), is a critical result for the foundational theory of Neural Networks. It states that a Feedforward Neural Network with atleast one hidden layer with finite neurons and an non-linear activation function e.g Sigmoid, ReLU, can approximate any continuous function. First I define the distance between two functions which I find using the norm of the functions $||.||$,

$$D(f(x), F(x)) = \|f(x) - F(x)\| \tag{6.4}$$

where $f(x)$ is the objective function and $F(x)$ is the neural network function. In training, the loss function is defined as:

$$L(\theta) := D(f(x), F(x|\theta)). \tag{6.5}$$

where $\theta = (W_1, b_1, W_2, b_2, \ldots, W_L, b_L)$ is the parameters vector, W_j is a weight matrix, and b_j is the bias vector in the L-th neural layer. The loss function is also known as the cost function or objective function. The aim of the training or learning stage is the minimize this function which is a measure of how well the model aligns with the actual outputs, which improves accuracy. Different tasks require different loss functions such as Mean Squared Error, Cross-Entropy Loss, Hinge Loss and Huber Loss. For this purposes I choose Mean Squared Error (MSE) as it penalises larger errors more heavily which is useful in option pricing where large errors can be particularly costly. The formula for MSE is as follows:

$$L_{\text{MSE}}(f(x), F(x|\Theta)) = \frac{1}{N}\sum_{j}^{N}\left(f(x_j) - F(x_j|\Theta)\right)^2 \tag{6.6}$$

where N is the number of samples.

6.4 Optimisation

Optimisation is a crucial part of training artificial neural networks. In the process, parameters of a network, the weights and biases, are iteratively adjusted to minimize the loss function as much as possible and so that the model learns the patterns in the underlying data. When ANNs are applied to financial modeling, this is important because it often contains noisy and complex data. Hence, good optimisation is key in building models that make accurate predictions. This section describes the various techniques in optimisation with neural networks, the pros and cons of such techniques, and specific considerations applied to financial data.

6.4.1 Gradient Descent

The training process' objective is to learn the optimal weights and biases for function $y(x) = F(x|\theta)$ to make the loss function minimal. The process can be formulated as an optimisation problem,

$$\arg\min_{\theta} L(\theta(x, y)), \tag{6.7}$$

Gradient descent is the most popular technique for optimizing neural networks. There are many variants of gradient descent, I will discuss the most common here, for more information on the topic see Ruder (2016).

Batch Gradient Descent
This is vanilla gradient descent in which the gradient of the loss function is computed

with respect to the parameters of the entire dataset.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta) \tag{6.8}$$

which can be quite slow because the gradients for the entire dataset must be computed in order to perform one update.

Stochastic Gradient Descent
This model attempts to fix the slowness issue by performing a parameter update for each training example.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta, x^{(i)}, y^{(i)}) \tag{6.9}$$

which tends to be much faster and can be updated on the fly. Its issue is that since it is fluctuating, it is likely to overshoot the exact minimum.

Mini-batch Gradient Descent
This model takes the best of both worlds and performs an update for mini-batches of n examples.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta, x^{(i+n)}, y^{(i+n)}) \tag{6.10}$$

This allows for efficient and stable convergence which is exactly want from an optimisation algorithm. Mini-batches tend to range between 50 and 256.

RMSProp
An adaptive learning rate optimisation algorithm that adjusts the learning rate for each parameter by maintaining an exponentially decaying average of squared gradients.

$$E[g^2]_t = 0.9 \cdot E[g^2]_{t-1} + 0.1 \cdot g_t^2 \tag{6.11}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{6.12}$$

where $E[g^2]_t$ is the moving average of squared gradients. η is the learning rate, typically set to 0.001.ϵ is a small constant to prevent division by zero. It adapts learning rates individually for each parameter but may not perform optimally for all data distributions.

Adam
Adaptive Moment Estimation. Allows for adaptive learning rates and momentum, allowing for faster and more reliable convergence.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{6.13}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{6.14}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{6.15}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{6.16}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t \tag{6.17}$$

where m_t and v_t are the first and second moment estimates of the gradients, $\beta_1$and β_2 are the decay rates, η is the learning rate, typically set to 0.001 and ϵ is a small constant to prevent division by zero. Adam generally requires less tuning of hyper-parameters and performs well on a wide range of problems.

6.4.2 Over-fitting

A common pitfall encountered with optimisation techniques is over-fitting which is where a model performs well with trained data but fails to generalise to new data. Techniques such as Regularisation, Dropout and Early Stopping help the network mitigate these issues.

Regularisation
Regularisation is a technique used to prevent over-fitting by adding a penalty to the loss function. The two most common forms of regularisation are L1, known as Lasso, and L2, known as Ridge, regularisation. In L2 regularisation, the loss function is modified by adding a penalty term that is proportional to the sum of the squares of the model parameters:

$$L(\theta) = L_0(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2 \tag{6.18}$$

where $L_0(\theta)$ is the original loss function, such as mean squared error, λ is the parameter that controls the strength of the penalty, and θ_j are the parameters of the model. The addition of the regularisation term helps in reducing over-fitting by penalizing large coefficients, leading to a more generalised model. However, careful tuning of λ is essential, as an overly large value can cause under-fitting.

Dropout
Dropout is another regularisation technique where, in training, a random subset of neurons is ignored or "dropped out" in each pass. This technique prevents the network from becoming too reliant on any particular set of neurons, which helps improve generalisation. Mathematically, the output of each neuron is retained with a probability p and set to zero with a probability $1 - p$:

$$\hat{y}_i = \begin{cases} y_i & \text{with probability } p \\ 0 & \text{with probability } 1 - p \end{cases} \tag{6.19}$$

where $\hat{y}_i$ denotes the output after dropout, y_i is the original output before dropout, and p is the dropout probability or droupout rate. Dropout requires a careful tuning of the dropout rate, and each iteration basically trains a different network, which can increase training time.

Early Stopping
Early stopping is a technique where the training process is halted once the model's performance on a validation dataset begins to deteriorate. This method helps to prevent over-fitting by stopping the training before the model becomes too tailored to the training data. The condition for early stopping can be mathematically expressed as stopping the training if the loss on the validation set increases.

6.4.3 Hyper-parameter tuning

Hyper-parameter tuning is another important aspect of optimizing neural networks. These hyper-parameters include factors such as the width (number of nodes per layer), depth (number of layers), type of activation function, learning rate, and regularisation parameters like dropout rate or early stopping time. Unlike model parameters, which the model learns during training, hyper-parameters are set before training begins and impact the model's ability to generalize to new data. For more information on this topic see Bergstra, Bardenet, Bengio and Kegl (2011).

The objective of hyper-parameter tuning is to find the optimal combination of these factors to achieve the best performance on unseen data. There are several approaches to hyper-parameter tuning, with grid search and random search being among the most common methods.

Grid search involves evaluating every possible combination. While this method ensures that the best combination within the chosen subset is found, it can be highly computationally expensive, especially when dealing with a large number of hyper-parameters.

Random search, on the other hand, involves selecting random combinations of hyper-parameters to evaluate. By focusing on a broader range of possibilities, random search can sometimes discover optimal settings more quickly than grid search.

Many machine learning libraries have inbuilt hyper-parameter tuning techniques. For example, libraries like Scikit-learn and TensorFlow provide tools for automating the hyper-parameter tuning process. Automated tuning tools can significantly reduce the time and effort required to find the best hyper-parameters.

6.5 ANNs in Finance

The Financial industry is always looking to adopt the newest, most innovative technologies to gain a competitive edge. ANNs have become incredibly popular and are at the forefront of research in the field.

ANNs are well suited for financial modelling due to their ability to capture the non-linearity of financial markets which can sometimes be missed by traditional financial models that assume linearity or normality. They are also capable of handling large and high-dimensional datasets and are able to identify the relevant patterns in historical data and learn from them to improve the accuracy of predictions. ANNs are also adaptable which is important in the fast-changing environment of finance.

When integrated with traditional models such as the Heston model, ANNs can enhance bypass the need to solve PDE's numerically and assist in calibrating the Heston model to market data. This leads in enhanced efficiency and accuracy.

Chapter 7

Methodology and implementation

In this section I will cover the design of the Artificial Neural Network model. I will discuss the Data used to train the model and the architecture of the model itself. The code is written in Python 3.12.4 and scripts are available in the Appendix.

7.1 Data

In this section I discuss the code used to generate the data used to train the neural network model. After testing careful consideration and significant testing I decided to use the Fourier method and the Finite Difference method to generate large data sets.

7.1.1 Fourier method

The script in *Appendix A: Listing 1* outlines the implementation of the Fourier method I discussed earlier in the thesis. In line 8 I define the Heston characteristic function as before. The Call price is then derived using the Fourier inversion method, integrating the characteristic function and computing the corresponding option price.

To generate a diverse set of parameter combinations for the model, I use a process known as Latin Hypercube sampling. This is a method which divides each parameter range into equally probable intervals. It is commonly used when sampling in high-dimensional spaces making it perfect for use in this instance, for more information see (McKay, Beckman & Conover 1979). I generate parameters within sensible ranges, inspired by (Guerrero Torres 2019) as seen in table 7.1.

I use the LHS built into the SciPy module to generate the parameters N times and then solve using the Fourier inversion. For simplicity, I set $K = 1$. The data is then stored in a .csv file with the LHS generated parameters and the corresponding option price, which can then be used to train the neural network.

Parameters	**Range**	**Method**
Moneyness, $m = \frac{S_0}{K}$	(0.6, 1.4)	Grid
Time to maturity, τ	(0.1, 1.4)	LHS
Risk free rate, r	(0, 0.1)	LHS
Correlation, ρ	(-0.95, 0.0)	LHS
Reversion speed, κ	(0.0, 2.0)	LHS
Long average variance, θ	(0.0, 0.5)	LHS
Volatility of volatility, σ	(0.0, 0.5)	LHS
Initial variance, ν_0	(0.05, 0.5)	Grid

Table 7.1: Fourier parameter ranges and method

7.1.2 Finite Difference Method

The FDM generates a non-uniform grid of S_0 and v_0 which gives Moneyness and Initial variance. This grid is then paired with sets of the remaining six parameters which are generated in their respective ranges by using Latin Hypercube sampling. Below is a table of the input parameters with their corresponding ranges and sampling method.

Parameters	**Range**	**Method**
Moneyness, $m = \frac{S_0}{K}$	(0.6, 1.4)	Grid
Time to maturity, τ	(0.1, 1.4)	LHS
Risk free rate, r	(0, 0.1)	LHS
Correlation, ρ	(-0.95, 0.0)	LHS
Reversion speed, κ	(0.0, 2.0)	LHS
Long average variance, θ	(0.0, 0.5)	LHS
Volatility of volatility, σ	(0.0, 0.5)	LHS
Initial variance, ν_0	(0.05, 0.5)	Grid

Table 7.2: FDM parameter ranges and method

Within the FDM script in *Appendix A: Listing 2* the non-uniform grid for the asset price and variance spans from S_{min} to S_{max} and from V_{min} to V_{max}. The non-uniform grid to be generated differently each time. The grid was constructed with 50 price steps and 25 variance steps and is of size $N = (N_S + 1)(N_V + 1)$ meaning that I have 1,326 option prices every time I run the FDM script. This helps greatly reduce the computational cost. In the time discretisation process, time to maturity T is divided into 20 time steps, this was chosen to again improve computational efficiency whilst preserving accuracy.

Within the script are various functions such as lines 184-195 where I define the Crank-Nicolson scheme in which I added an enforced negativity condition to ensure prices remain non-negative throughout calculations. Functions for the matrices, coefficients and boundary conditions are also detailed following the mathematical theory discussed earlier.

Then in *Appendix A: Listing 3* I have the code that runs the FDM. You will notice in line 7 and in other scripts in the appendix a line making use of the time module, this is to track how long it takes to run each script. Within this script I generate the remaining parameters in the previously defined ranges using Latin Hypercube sampling from SciPy. Again, for simplicity's sake I set $K = 1$. The script runs N times, generating Heston model parameters and solving using the FDM, outputting the data to a .csv file. This data set then needs to be filtered using *Appendix A: Listing 4*, which filters the data so that moneyness and initial variance are within the ranges in the tables. See also *Appendix A: Listing 5* for a script that standardizes the data set, I will discuss later the influence of making this change.

7.1.3 Comparison

In the end I am able to generate large data sets within realistic time frames using admittedly sub-optimal hardware (A mac book air 2017). I was able to generate data sets of around a million parameter sets and corresponding option prices for each method, with the FDM being the more computationally expensive of the two. Below are tables for the mean and variance of both sets of generated parameters and their corresponding option price.

Fourier Inversion:	**mean**	**std**
moneyness	1.000000	0.230941
tau	0.750000	0.375278
r	0.050000	0.028868
rho	-0.475000	0.274242
kappa	1.000000	0.577351
theta	0.250000	0.144338
sigma	0.250000	0.144338
initial_variance	0.275000	0.129904
option_price	0.185125	0.110517

Table 7.3: Statistics for Fourier_Prices.csv

Finite Difference Method:	**mean**	**std**
moneyness	0.999965	0.230948
tau	0.749671	0.375262
r	0.050000	0.028867
rho	-0.475021	0.274231
kappa	0.999432	0.577273
theta	0.249835	0.144303
sigma	0.250259	0.144198
initial_variance	0.275186	0.129843
option_price	0.193602	0.187365

Table 7.4: Statistics for parameter_option_prices.csv

7.2 Artificial Neural Network

The implementation of the artificial neural network can be seen in *Appendix B: Listings 1 and 2*, respectively. This section explains the architecture and training methodology.

7.2.1 Model Construction

The neural network architecture is defined within *Appendix B: Listing 1.* The `build_model` function constructs a sequential Keras model. This is made using TensorFlow 2.16.2.

Layer Configuration

- **Input Layer:** The model begins with an input layer that accepts the eight Heston model parameters from the input .csv file.
- **Hidden Layers:** A series of hidden layers follow, whose number and size can be changed. Each hidden layer makes use of the ReLU activation function, promoting non-linearity and reducing issues with vanishing gradients. The architecture supports the addition of multiple hidden layers, each potentially decreasing in size.
- **Dropout Regularisation:** To prevent over-fitting, dropout layers are introduced between dense layers. The dropout rate controls the fraction of neurons deactivated during training. This enhances the model's generalisation capabilities by lessening reliance on specific neurons.
- **Output Layer:** The final layer is a dense layer with a single neuron and a linear activation function

Regularisation is added through L2 penalties on the kernel weights of the dense layers. This technique constrains the magnitude of the weights, preventing over-fitting by discouraging overly complex models.

Weight initialisation is handled using the Glorot uniform initializer for hidden layers and a random normal initializer for the output layer. Proper initialisation is crucial for effective training, ensuring that gradients move efficiently through the network.

The model is compiled with the mean squared error (MSE) loss function and the Adam optimizer. The choice of MSE is appropriate measuring the average squared difference between predicted and actual values. Adam optimizer is selected for its adaptive learning rate capabilities. Additionally, MSE is included as a metric to monitor the model's performance.

7.2.2 Training Procedure

The training process is orchestrated in the *train_model.py* script *Appendix B: Listing 2.* The `train_neural_network` function encapsulates the entire workflow from data loading to model evaluation.

The script begins by loading a dataset from a CSV file. The eight input parameters are separated from the output parameter `option_price`.

Next, the dataset is partitioned into training and testing subsets using an 80-20 split. 80% of the data is for training while 20% is for testing. This split allows for unbiased evaluation of the model's performance on unseen data.

A neural network model is made by loading the `build_model` function with the appropriate input dimensions and hyper-parameters. The model architecture is then summarised, giving a clear overview of each layer and its parameters.

To prevent over-fitting and reduce unnecessary computations, an early stopping callback is added. Set up to monitor the validation loss, the training process stops if no improvement is observed over a predefined number of epochs. This ensures that the model retains the best-performing weights encountered during training.

The model is trained using the `fit` method, which iteratively updates the network's weights based on the training data. The training process takes advantage of batch processing, where subsets of the data are used to compute gradient updates, increasing computational efficiency. Validation is performed on the test set to monitor the model's generalisation performance in real-time.

Once training is complete, the model is saved to a specified file path, allowing for future deployment or analysis without retraining. Additionally, the training history, which has metrics such as loss and validation loss over epochs, is saved to a JSON file for subsequent examination.

Predictions are made on the test set, and both actual and predicted values are stored in a JSON file. This facilitates a comprehensive comparison and evaluation of the model's predictive accuracy, which is further discussed in subsequent sections of this thesis.

7.2.3 Hyper-parameter Configuration

The neural network's performance is sensitive to various hyper-parameters, including the number of hidden layers, nodes per layer, dropout rate, regularisation strength, and training parameters like epochs and batch size. The script allows for these parameters to be adjusted, the effects of which will be discussed in the Results section of the thesis. I employed hyper-parameter tuning using the inbuilt kerastuner, the script for which is available in *Appendix B: Listing 3*

Chapter 8

Results

In this section I will discuss my training results. I used data sets with around 1 million data points that were generated using the methods discussed ealier. I will begin by presenting the best results I was able to achieve and later discuss the effects of changing certain parameters. After implementing Hyper Parameter tuning as discussed earlier the model is shown in the table below.

Parameter	**Value**
Epochs	200
Hidden Layers	2
Nodes	700
Activation Function	ReLU
Dropout Rate	0.1
L2 Regularisation	2.35536e-05

Table 8.1: Neural Network Configuration

From this I achieved respectable results which are summarised in the table below. Figures are also available below that show the accuracy of the model. A line graph showing the training history of the model over the course of the epochs, a scatter graph plotting actual vs predicted option price values and a Histogram of the residual errors.

Method	**MSE**	$\mathbf{R^2}$
Finite Difference Method	1.14039e-04	0.99515
Fourier Inversion	1.10976e-05	0.99909

Table 8.2: Performance Metrics

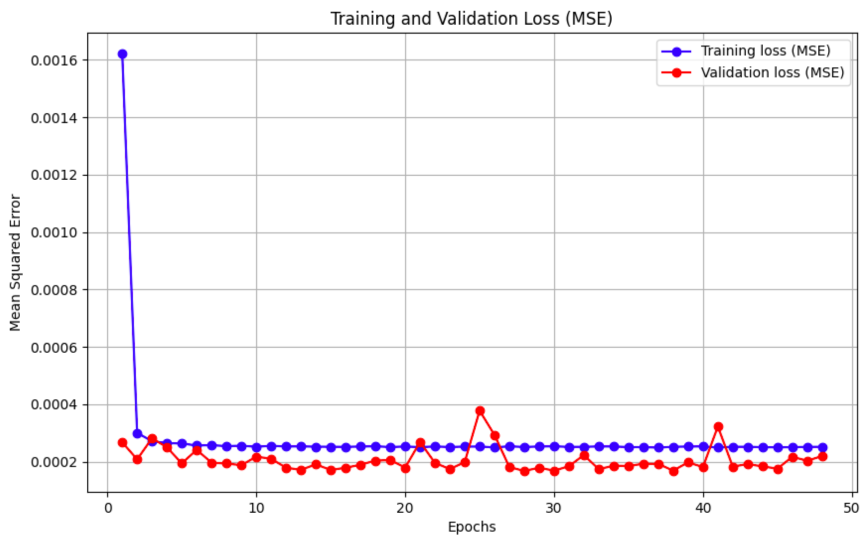


Figure 8.1: FDM training loss

Source: created using Python

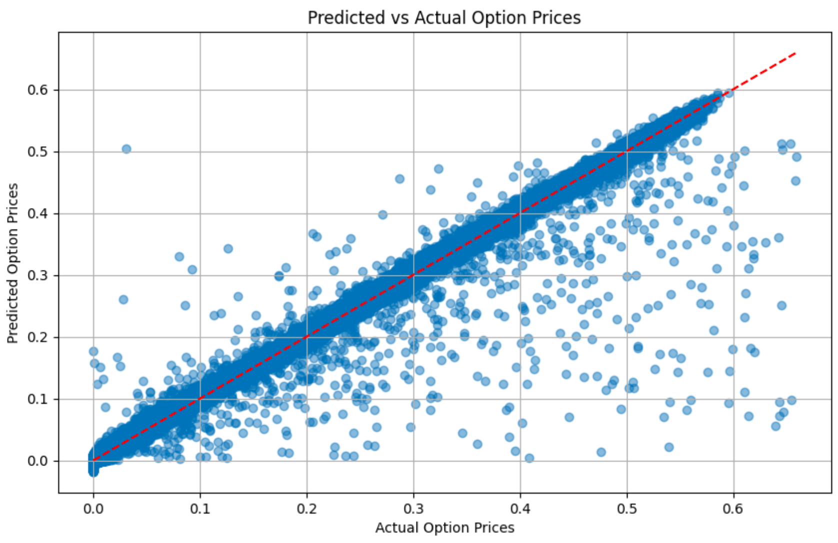


Figure 8.2: FDM Scatter Plot

Source: created using Python

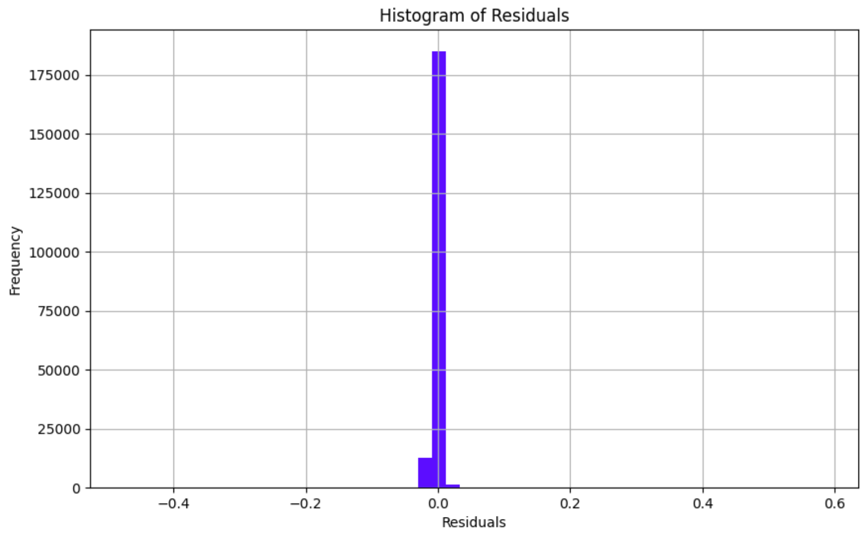


Figure 8.3: FDM Histogram of residuals

Source: created using Python

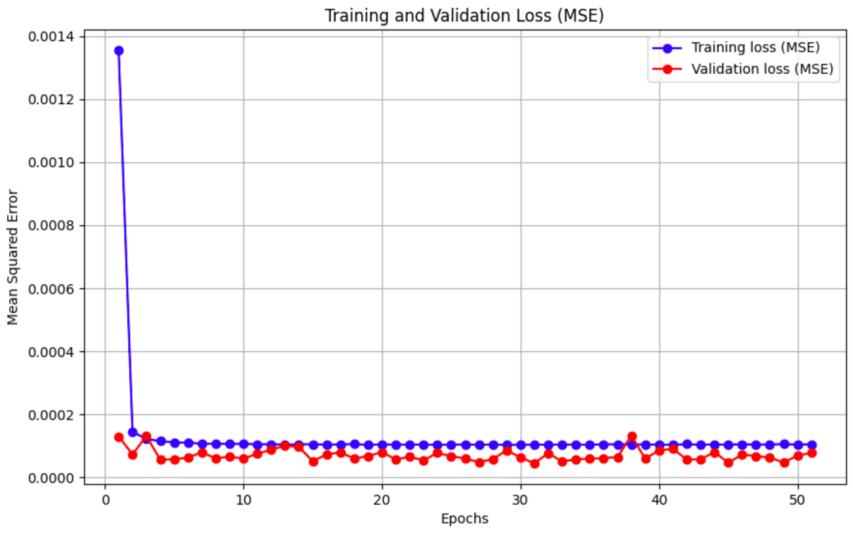


Figure 8.4: Fourier training loss

Source: created using Python

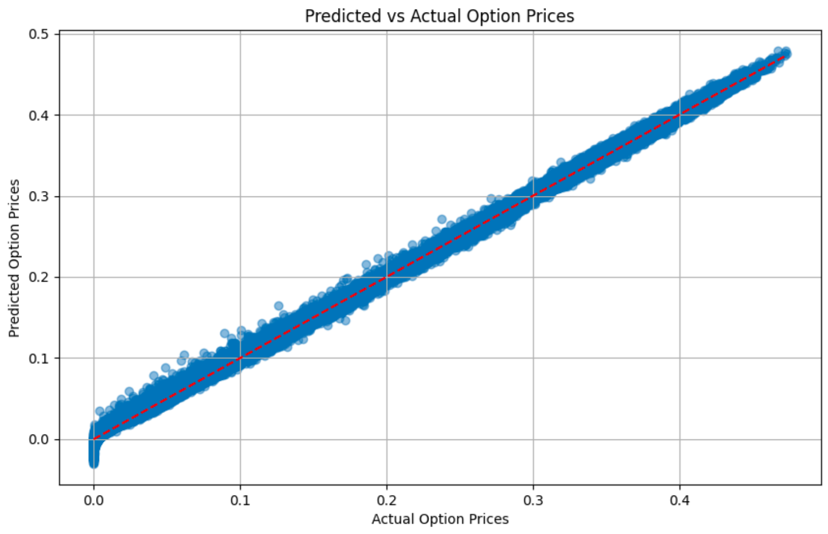


Figure 8.5: Fourier Scatter Plot

Source: created using Python

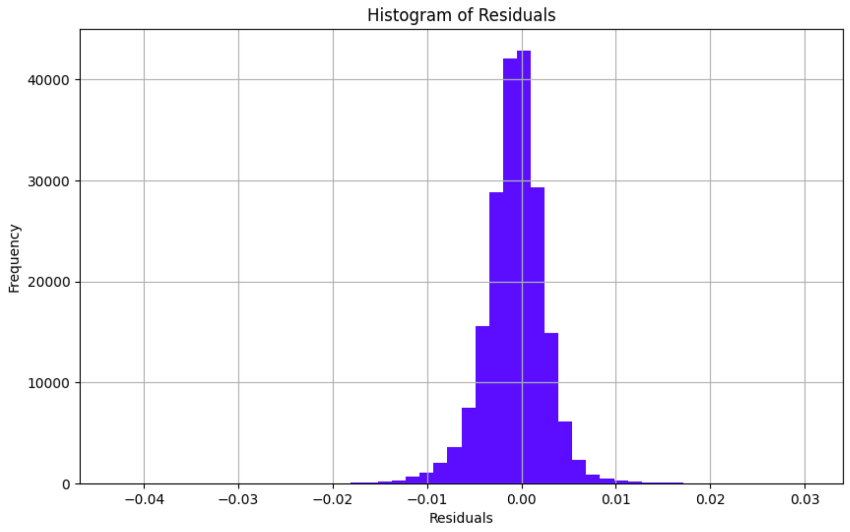


Figure 8.6: Fourier Histogram of residuals

Source: created using Python

Chapter 9

Conclusions and recommendations for future research

In this thesis, I have presented a derivation of the Heston model, providing a clear simplified understanding of its mathematical foundations and practical applications. I have compared it extensively to the well-known Black-Scholes model, highlighting the advantages and limitations of each in modeling financial markets. Additionally, I have explored various numerical methods for solving the Heston model, explaining both analytical and computational approaches.

I have also conducted an investigation into the architecture of neural networks, examining how different configurations can be used to approximate complex financial models. From this analysis, I have built a neural network model to capture the dynamics described by the Heston model, ensuring that the network's structure aligns with the unique characteristics of the data.

The developed neural network model successfully demonstrated its ability to learn the behavior of the Heston model from the provided data set. This success can be seen through various the performance metrics and visualizations, confirming that the neural network can effectively approximate the Heston model.

From Table 8.2 in the previous chapter, it is clear that I achieved respectable accuracy metrics after training the model. The Fourier data set performed significantly better—more than 10 times that of the Finite Difference Method in terms of Mean Squared Error (MSE). The FDM data displayed some outliers, as can be seen in Figure 8.2. This is likely due in part to the data generation used in this project for the FDM; it is possible that a superior implementation of the finite difference method would improve the prediction accuracy.

I observed significant impact when changing the hyper-parameters from those given by the initial tuning. Although this tuning could be improved upon by implementing different methods such as Bayesian hyper-parameter tuning, the current approach provided a good starting point for model optimization.

Exploring the use of a deeper or wider neural network architecture could maybe lead to better accuracy. However, when attempting such modifications, I encountered issues with over-fitting, which suggests that further work is needed to balance model complexity and generalization. Nevertheless, within the scope of this project, I have demonstrated that it is possible to effectively model option pricing using neural networks, achieving respectable accuracy metrics and providing insights into the potential of machine learning in financial modeling.

In future work, the task of pricing exotic options using the Heston model could be considered, where a similar neural network approach could be applied to capture the more complex features of these financial instruments. Additionally, the application of neural networks to more modern models such as the Rough Heston model or Generalized Autoregressive Conditional Heteroskedasticity (GARCH) models presents an exciting opportunity for future research. These models include additional market features like volatility clustering and long-range dependence, which could further enhance the predictive power of machine learning methods in finance. By integrating these advanced models with neural network architectures, we could potentially see similar improvements as seen in this thesis.

Overall, this thesis shows the vast potential of artificial intelligence and machine learning in revolutionizing financial modeling. By harnessing the power of neural networks, we can develop models that match the capabilities of traditional numerical methods, increase computational efficiency and improve the ability to handle complex, high-dimensional problems. As the financial industry continues to evolve, integrating AI and machine learning will be essential for advancing our understanding of markets and developing innovative solutions for pricing.

Bibliography

Albrecher, H., Mayer, P., Schoutens, W. & Tistaert, J. (2007), 'The little heston trap', *Preprint* . First version: December 6, 2005.

Andersen, L. (2007), 'Efficient simulation of the heston stochastic volatility model'. Available at SSRN: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=946405`.

Ané, T. & Labidi, C. (2001), 'Implied volatility surfaces and market activity over time', *Journal of Economics and Finance* **25**(2), 259–275.

Bakshi, G., Cao, C. & Chen, Z. (1997), 'Empirical performance of alternative option pricing models', *The Journal of Finance* **52**(5), 2003–2049.

Black, F. & Scholes, M. (1973), 'The pricing of options and corporate liabilities', *Journal of Political Economy* **81**(3), 637–654.

Boyle, P., Broadie, M. & Glasserman, P. (1997), 'Monte carlo methods for security pricing', *Journal of Economic Dynamics and Control* **21**(8-9), 1267–1321.

Bracewell, R. N. (1999), *The Fourier Transform and Its Applications*, 3rd edn, McGraw-Hill.

Broadie, M. & Kaya, O. (2006), 'Exact simulation of stochastic volatility and other affine jump diffusion processes', *Operations Research* **54**(2), 217–231.

Carr, P. & Madan, D. B. (1999), 'Option valuation using the fast fourier transform', *Journal of Computational Finance* **2**(4), 61–73.

Cox, J. C., Ingersoll, J. E. & Ross, S. A. (1985), 'A theory of the term structure of interest rates', *Econometrica* **53**(2), 385–407.

Crank, J. & Nicolson, P. (1947), 'A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type', *Proceedings of the Cambridge Philosophical Society* **43**, 50–67.

Derman, E. & Miller, M. B. (2016), *The Volatility Smile*, Wiley Finance, illustrated edn, John Wiley & Sons.

Fermi, E. (1935), 'On the recombination of neutrons and protons', *Physical Review* **48**(6), 570–570.
URL: *https://doi.org/10.1103/PhysRev.48.570*

Fischer, T. & Krauss, C. (2018), 'Deep learning with long short-term memory networks for financial market predictions', *European Journal of Operational Research* **270**(2), 654–669.

Fogarasi, N. (2004), 'Option pricing using neural networks', *Article* . Available at: `http://www.hit.bme.hu/~fogarasi/NNoptionPricing.pdf`.

Gatheral, J. (2006), *The Volatility Surface: A Practitioner's Guide*, Wiley Finance, Wiley, Hoboken, NJ.

Goodfellow, I., Bengio, Y. & Courville, A. (2016), *Deep Learning*, MIT Press.

Guerrero Torres, S. P. (2019), A neural network approach to pricing of a european call option with the heston model, Master's thesis, Universidad del Rosario, Bogotá, Colombia.

Heston, S. L. (1993), 'A closed-form solution for options with stochastic volatility with applications to bond and currency options', *The Review of Financial Studies* **6**(2), 327–343.

Hilpisch, Y. (2020), *Artificial Intelligence in Finance: A Python-Based Guide*, O'Reilly Media.

Hull, J. (2017), *Options, Futures, and Other Derivatives*, ebook, global edition edn, Pearson Education, Limited. ProQuest Ebook Central.
URL: *https://ebookcentral.proquest.com/lib/liverpool/detail.action?docID=5186416*

Hull, J. C. (1997), *Options, Futures, and Other Derivatives*, 3rd edn, Prentice Hall, Upper Saddle River, NJ.

Hull, J. & White, A. (1987), 'The pricing of options on assets with stochastic volatilities', *The Journal of Finance* **42**(2), 281–300.

Klingberg Malmer, O. & Tisell, V. (2020), 'Deep learning and the heston model: Calibration & hedging'. Thesis advisor: Alexander Herbertsson.

Marotta, D. J. (2018), 'Black monday bear: The bear market of 1987', *Forbes* . Accessed: 2024-06-28.
URL: *https://www.forbes.com/sites/davidmarotta/2018/05/21/black-monday-bear-the-bear-market-of-1987/*

Mayor, A. (2018), *Gods and Robots: Myths, Machines, and Ancient Dreams of Technology*, Princeton University Press. Accessed 22 Aug. 2024.
URL: *https://doi.org/10.2307/j.ctvc779xn*

McCulloch, W. S. & Pitts, W. (1943), 'A logical calculus of the ideas immanent in nervous activity', *The bulletin of mathematical biophysics* **5**(4), 115–133.

McKay, M. D., Beckman, R. J. & Conover, W. J. (1979), 'A comparison of three methods for selecting values of input variables in the analysis of output from a computer code', *Technometrics* **21**(2), 239–245.

Merton, R. C. (1973), 'Theory of rational option pricing', *Bell Journal of Economics and Management Science* **4**(1), 141–183.

Minsky, M. & Papert, S. (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press.

Nunno, G. D., Kubilius, K., Mishura, Y. & Yurchenko-Tytarenko, A. (2023), 'From constant to rough: A survey of continuous volatility modeling', *Mathematics* **11**(19), 4201.
URL: *https://www.mdpi.com/2227-7390/11/19/4201*

Oosterlee, C. W. & Grzelak, L. A. (2019), *Mathematical Modeling and Computation in Finance: With Exercises and Python and MATLAB Computer Codes*, World Scientific Publishing Europe Ltd.

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psychological Review* **65**(6), 386–408.

Rouah, F. D. (2013), *The Heston Model and Its Extensions in C++*, Wiley, Hoboken, NJ.

Ruf, J. & Wang, W. (2020), 'Neural networks for option pricing and hedging: A literature review', *Article* . 150 papers categorized. Available at: `https://arxiv.org/pdf/1911.05620`.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986), 'Learning representations by back-propagating errors', *Nature* **323**(6088), 533–536.

Rumelhart, D. E. & McClelland, J. L. (1986), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*, MIT Press.

Schilling, R. L. (2021), *Brownian Motion: A Guide to Random Processes and Stochastic Calculus*, De Gruyter, Berlin, Boston.
URL: *https://doi.org/10.1515/9783110741278*

Scott, L. O. (1987), 'Option pricing when the variance changes randomly: Theory, estimation and an application', *Journal of Financial and Quantitative Analysis* **22**(4), 419–438.

Shreve, S. E. (2004), *Stochastic Calculus for Finance II: Continuous-Time Models*, Springer Finance, Springer, New York, NY.

Appendix A

```python
import numpy as np
from scipy.integrate import quad
from scipy.stats import qmc
import csv
import time  # Importing the time module

def heston_char_function(u, t, S0, r, kappa, theta, sigma, rho, v0):
    """
    Heston characteristic function.
    """
    i = complex(0, 1)
    a = kappa * theta

    d = np.sqrt((rho * sigma * i * u - kappa)**2 + (sigma**2) * (i * u + u**2))
    g = (kappa - rho * sigma * i * u - d) / (kappa - rho * sigma * i * u + d)

    C = (r * i * u * t) + (a / sigma**2) * ((kappa - rho * sigma * i * u - d) * t -
        2 * np.log((1 - g * np.exp(-d * t)) / (1 - g)))

    D = ((kappa - rho * sigma * i * u - d) / sigma**2) * (1 - np.exp(-d * t)) / (1 - g * np.exp(-d * t))

    phi = np.exp(C + D * v0 + i * u * np.log(S0))
    return phi

def heston_p1_integrand(u, S0, K, t, r, kappa, theta, sigma, rho, v0):
    """
    Integrand for computing P1 in the Heston model.
    """
    i = complex(0, 1)
    phi = heston_char_function(u - i, t, S0, r, kappa, theta, sigma, rho, v0)
    phi_minus_i = heston_char_function(-i, t, S0, r, kappa, theta, sigma, rho, v0)
    numerator = np.exp(-i * u * np.log(K)) * phi
    denominator = i * u * phi_minus_i
    integrand = (numerator / denominator).real
    return integrand

def heston_p2_integrand(u, S0, K, t, r, kappa, theta, sigma, rho, v0):
    """
    Integrand for computing P2 in the Heston model.
    """
    i = complex(0, 1)
    phi = heston_char_function(u, t, S0, r, kappa, theta, sigma, rho, v0)
    numerator = np.exp(-i * u * np.log(K)) * phi
    denominator = i * u
    integrand = (numerator / denominator).real
    return integrand

def heston_call_price(S0, K, t, r, kappa, theta, sigma, rho, v0):
    """
    Computes the European call option price using the Heston model via Fourier inversion.
    """
    # Compute P1
    integrand_p1 = lambda u: heston_p1_integrand(u, S0, K, t, r, kappa, theta, sigma, rho, v0)
    integral_p1, _ = quad(integrand_p1, 0, 100, limit=100, epsabs=1e-8, epsrel=1e-8)
    P1 = 0.5 + (1 / np.pi) * integral_p1

    # Compute P2
    integrand_p2 = lambda u: heston_p2_integrand(u, S0, K, t, r, kappa, theta, sigma, rho, v0)
    integral_p2, _ = quad(integrand_p2, 0, 100, limit=100, epsabs=1e-8, epsrel=1e-8)
    P2 = 0.5 + (1 / np.pi) * integral_p2

    # Compute the call price
    call_price = S0 * P1 - K * np.exp(-r * t) * P2
    return call_price

def generate_lhs_samples(samples=100):
    """
    Generates Latin Hypercube Sampling for the eight Heston model parameters using scipy's LHS.
    """
    lhs_sampler = qmc.LatinHypercube(d=8)
    lhs_samples = lhs_sampler.random(n=samples)

    # Define parameter ranges
    m_min, m_max = 0.6, 1.4
    tau_min, tau_max = 0.1, 1.4  # Time to maturity in years
    r_min, r_max = 0.0, 0.10  # Risk-free rate
    rho_min, rho_max = -0.95, 0.0  # Correlation
    kappa_min, kappa_max = 0.0, 2.0  # Reversion speed
    theta_min, theta_max = 0.0, 0.5  # Long average variance
    sigma_min, sigma_max = 0.0, 0.5  # Volatility of volatility
    v0_min, v0_max = 0.05, 0.5  # Initial variance

    # Scale LHS samples to the parameter ranges
    m = m_min + lhs_samples[:, 0] * (m_max - m_min)
    tau = tau_min + lhs_samples[:, 1] * (tau_max - tau_min)
    r = r_min + lhs_samples[:, 2] * (r_max - r_min)
    rho = rho_min + lhs_samples[:, 3] * (rho_max - rho_min)
    kappa = kappa_min + lhs_samples[:, 4] * (kappa_max - kappa_min)
    theta = theta_min + lhs_samples[:, 5] * (theta_max - theta_min)
```

```python
    sigma = sigma_min + lhs_samples[:, 6] * (sigma_max - sigma_min)
    v0 = v0_min + lhs_samples[:, 7] * (v0_max - v0_min)

    return np.column_stack((m, tau, r, rho, kappa, theta, sigma, v0))

def save_to_csv(data, filename="Fourier_Prices.csv"):
    """
    Saves the list of call prices and parameter sets to a CSV file.
    """
    header = ['moneyness', 'tau', 'r', 'rho', 'kappa', 'theta', 'sigma', 'initial_variance', 'option_price']

    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(header)  # Write header
        for row in data:
            writer.writerow(row)

def main():
    N = int(input("Enter the number of prices to generate (N): "))  # User-defined N

    start_time = time.time()  # Start time for timing the function

    # Generate N parameter sets using Latin Hypercube Sampling
    param_sets = generate_lhs_samples(N)

    S0 = 1.0  # Fixed initial stock price
    result_data = []

    for idx, params in enumerate(param_sets):
        m, tau, r, rho, kappa, theta, sigma, v0 = params
        K = S0 / m  # Compute strike price based on moneyness

        # Compute the European call option price using the Heston model
        price = heston_call_price(S0, K, tau, r, kappa, theta, sigma, rho, v0)

        # Apply the same filtering as in the FDM script
        if (0.6 <= m <= 1.4) and (0.05 <= v0 <= 0.5):
            result_data.append([m, tau, r, rho, kappa, theta, sigma, v0, price])

        Print progress every 1000 iterations
        if (idx + 1) % 1000 == 0:
            print(f"Processed {idx + 1}/{N} parameter sets.")

    # Save the simulated data to a CSV file
    save_to_csv(result_data, "Fourier_Prices.csv")

    # Calculate and print the execution time
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"\nExecution Time: {execution_time:.4f} seconds")
    print(f"\nSimulated data for {len(result_data)} valid prices have been saved to Fourier_Prices.csv.")

if __name__ == "__main__":
    main()
```

Listing A.1: Fourier Inversion Method in Python

```
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import spsolve
from math import sinh, asinh
from scipy.sparse import lil_matrix
from scipy.interpolate import RegularGridInterpolator

def fdm(time_to_maturity, risk_free_rate, reversion_speed,
        long_average_variance, vol_of_vol, correlation,
        m, v0):
    # Set S0 and V0 based on moneyness and initial variance
    K = 1.0  # Strike price
    S0 = m * K
    V0 = v0

    # Adjust grid ranges to include S0 and V0
    S_min = S0 * 0.5
    S_max = S0 * 1.5
    V_min = max(0.05, V0 * 0.5)
    V_max = min(0.5, V0 * 1.5)

    T = time_to_maturity
    r_d = risk_free_rate  # Domestic interest rate
    rho = correlation
    sigma = vol_of_vol
    kappa = reversion_speed
    eta = long_average_variance

    # Number of price and variance steps
    ns = 50  # Number of price steps
    nv = 25  # Number of variance steps
    c = (S_max - S_min) / 10
    d = (V_max - V_min) / 10

    # Time discretisation
    N = 20
    delta_t = T / N

    # Generate the non-uniform grid, which will give us S_0 and V_0
    Vec_s, Delta_s, Vec_v, Delta_v, X, Y = make_grid(ns, S_min, S_max, c
    , nv, V_min, V_max, d)

    # Matrices
    A = make_matrices(ns, nv, rho, sigma, r_d, kappa, eta, Vec_s, Vec_v,
    Delta_s, Delta_v)
    B = make_boundaries(ns, nv, r_d, Vec_s)

    # Initial condition
    UU_0 = np.array([[max(Vec_s[i] - K, 0) for i in range(ns + 1)] for _
    in range(nv + 1)])
    U_0 = UU_0.flatten()

    # Solve the PDE using Crank-Nicolson scheme
    n = (ns + 1) * (nv + 1)
    price_cn = cn_scheme(n=n, n_tm=N, u_0=U_0, delta_t=delta_t, l=A, b=B
    )
    price_cn = np.reshape(price_cn, (nv + 1, ns + 1))

    interpolator = RegularGridInterpolator((Vec_v, Vec_s), price_cn)
    option_price = interpolator((V0, S0))

    return option_price, m, V0 # Return arrays

def make_grid(ns, s_min, s_max, c, nv, v_min, v_max, d):
    delta_zeta_i = (1.0 / ns) * (asinh((s_max - s_min) / c))
    zeta_s = [asinh((s_min - s_min) / c) + i * delta_zeta_i for i in
  range(ns + 1)]
    vec_s = [g(zeta_s[i], s_min, c) for i in range(ns + 1)]

    delta_eta = (1.0 / nv) * (asinh((v_max - v_min) / d))
    eta_v = [asinh((v_min - v_min) / d) + i * delta_eta for i in range(
  nv + 1)]
    vec_v = [h(eta_v[i], v_min, d) for i in range(nv + 1)]

    x, y = np.meshgrid(vec_s, vec_v)

    # Compute deltas
    delta_s = [vec_s[i + 1] - vec_s[i] for i in range(len(vec_s) - 1)]
    delta_v = [vec_v[i + 1] - vec_v[i] for i in range(len(vec_v) - 1)]

    return np.array(vec_s), delta_s, np.array(vec_v), delta_v, x, y

def g(xi, s_min, c):
    return s_min + c * sinh(xi)

def h(xi, v_min, d):
    return v_min + d * sinh(xi)

# Central approximation of derivatives, inner points
def central_coefficients1(i, index, delta):
    if index == -1:
        return -delta[i + 1] / (delta[i] * (delta[i] + delta[i + 1]))
    elif index == 0:
        return (delta[i + 1] - delta[i]) / (delta[i] * delta[i + 1])
    elif index == 1:
        return delta[i] / (delta[i + 1] * (delta[i] + delta[i + 1]))

def central_coefficients2(i, index, delta):
    if index == -1:
        return 2 / (delta[i] * (delta[i] + delta[i + 1]))
    elif index == 0:
        return -2 / (delta[i] * delta[i + 1])
    elif index == 1:
        return 2 / (delta[i + 1] * (delta[i] + delta[i + 1]))

# Backward approximation of derivatives, right border
def backward_coefficients(i, index, delta):
    if index == -2:
        return delta[i] / (delta[i - 1] * (delta[i] + delta[i - 1]))
    elif index == -1:
        return (-delta[i] - delta[i - 1]) / (delta[i] * delta[i - 1])
    elif index == 0:
        return (2 * delta[i] + delta[i - 1]) / (delta[i] * (delta[i] +
    delta[i - 1]))

# Forward approximation of derivatives, left border
def forward_coefficients(i, index, delta):
    if index == 0:
        return (-2 * delta[i + 1] - delta[i + 2]) / (delta[i + 1] * (
    delta[i + 1] + delta[i + 2]))
    elif index == 1:
        return (delta[i + 1] + delta[i + 2]) / (delta[i + 1] * delta[i +
    2])
    elif index == 2:
        return -delta[i + 1] / (delta[i + 2] * (delta[i + 1] + delta[i +
    2]))

def make_matrices(ns, nv, rho, sigma, r_d, kappa, eta, vec_s, vec_v,
    delta_s, delta_v):
    n = (ns + 1) * (nv + 1)
    a_0 = lil_matrix((n, n))
    a_1 = lil_matrix((n, n))
    a_2 = lil_matrix((n, n))

    # (cross derivative terms)
    for j in range(1, nv):
        for i in range(1, ns):
            c = rho * sigma * vec_s[i] * vec_v[j]
            for k in [-1, 0, 1]:
                for l in [-1, 0, 1]:
                    idx_i_j = i + j * (ns + 1)
                    idx_k_l = (i + k) + (j + l) * (ns + 1)
                    if 0 <= i + k <= ns and 0 <= j + l <= nv:
                        coeff = c * central_coefficients1(i - 1, k,
    delta_s) * central_coefficients1(j - 1, l, delta_v)
                        a_0[idx_i_j, idx_k_l] += coeff

    # (price derivative terms)
    for j in range(nv + 1):
        for i in range(1, ns):
            b = r_d * vec_s[i]
            c = 0.5 * vec_s[i] ** 2 * vec_v[j]
            idx_i_j = i + j * (ns + 1)
            for k in [-1, 0, 1]:
                if 0 <= i + k <= ns:
                    idx_k_j = (i + k) + j * (ns + 1)
                    coeff = c * central_coefficients2(i - 1, k, delta_s)
    + b * central_coefficients1(i - 1, k, delta_s)
                    a_1[idx_i_j, idx_k_j] += coeff
            a_1[idx_i_j, idx_i_j] += -0.5 * r_d

    # (variance derivative terms)
    for j in range(1, nv):
        for i in range(ns + 1):
            d = kappa * (eta - vec_v[j])
            e = 0.5 * sigma ** 2 * vec_v[j]
            idx_i_j = i + j * (ns + 1)
            for k in [-1, 0, 1]:
                if 0 <= j + k <= nv:
                    idx_i_jk = i + (j + k) * (ns + 1)
                    coeff = d * central_coefficients1(j - 1, k, delta_v)
    + e * central_coefficients2(j - 1, k, delta_v)
                        a_2[idx_i_j, idx_i_jk] += coeff
                a_2[idx_i_j, idx_i_j] += -0.5 * r_d

    # Convert the lil_matrix to csc_matrix after construction
    a_0 = a_0.tocsc()
    a_1 = a_1.tocsc()
    a_2 = a_2.tocsc()

    a = a_0 + a_1 + a_2

    return a

    # Definition of a_1 (price derivative terms)
    for j in range(nv + 1):
        for i in range(1, ns):
            b = r_d * vec_s[i]
            c = 0.5 * vec_s[i] ** 2 * vec_v[j]
            idx_i_j = i + j * (ns + 1)
            for k in [-1, 0, 1]:
                if 0 <= i + k <= ns:
                    idx_k_j = (i + k) + j * (ns + 1)
                    coeff = c * central_coefficients2(i - 1, k, delta_s)
    + b * central_coefficients1(i - 1, k, delta_s)
                    a_1[idx_i_j, idx_k_j] += coeff
            a_1[idx_i_j, idx_i_j] += -0.5 * r_d

    # Definition of a_2 (variance derivative terms)
    for j in range(1, nv):
        for i in range(ns + 1):
            d = kappa * (eta - vec_v[j])
            e = 0.5 * sigma ** 2 * vec_v[j]
            idx_i_j = i + j * (ns + 1)
            for k in [-1, 0, 1]:
                if 0 <= j + k <= nv:
                    idx_i_jk = i + (j + k) * (ns + 1)
                    coeff = d * central_coefficients1(j - 1, k, delta_v)
    + e * central_coefficients2(j - 1, k, delta_v)
                    a_2[idx_i_j, idx_i_jk] += coeff
            a_2[idx_i_j, idx_i_j] += -0.5 * r_d

    a = a_0 + a_1 + a_2

    return a

def make_boundaries(ns, nv, r_d, vec_s):
    n = (ns + 1) * (nv + 1)
    b = np.zeros(n)

    # Boundary when s = s_max
    for j in range(nv + 1):
        idx = ns + j * (ns + 1)
        b[idx] = r_d * vec_s[-1]

    # Boundary when v = v_max
    for i in range(ns + 1):
        idx = i + nv * (ns + 1)
        b[idx] += -0.5 * r_d * vec_s[i]
```

```python

    return b

def cn_scheme(n, n_tm, u_0, delta_t, l, b):
    u = u_0
    identity = csc_matrix(np.identity(n))
    lhs = identity - 0.5 * delta_t * l
    rhs_constant = 0.5 * delta_t * b

    for _ in range(n_tm):
        rhs = (identity + 0.5 * delta_t * l).dot(u) + rhs_constant
        u = spsolve(lhs, rhs)
        u = np.maximum(u, 0)  # Enforce non-negativity

    return u
```

Listing A.2: Finite Difference Method in Python

```python
import numpy as np
import pandas as pd
from scipy.stats import qmc
from fdm import fdm
import time

start_time = time.time()

# Sample size (number of parameter sets to generate)
N = 250000  # Adjust this value as needed

# Define the parameter ranges
param_ranges = {
    'm': (0.6, 1.4),           # Moneyness
    'tau': (0.1, 1.4),         # Time to maturity
    'r': (0.0, 0.10),          # Risk-free rate
    'rho': (-0.95, 0.0),       # Correlation
    'kappa': (0.0, 2.0),       # Reversion speed
    'theta': (0.0, 0.5),       # Long average variance
    'sigma': (0.0, 0.5),       # Volatility of volatility
    'v0': (0.05, 0.5),         # Initial variance
}

# Number of parameters to sample via LHS
n_params = len(param_ranges)

# Initialize the LHS sampler
sampler = qmc.LatinHypercube(d=n_params)

# Generate LHS samples
lhs_sample = sampler.random(n=N)

# Scale the samples to the parameter ranges
param_values = {}
for i, key in enumerate(param_ranges.keys()):
    low, high = param_ranges[key]
    param_values[key] = qmc.scale(lhs_sample[:, [i]], low, high).flatten
   ()

# Combine all parameters into a DataFrame
data = pd.DataFrame()
```

```python

# Add the LHS-sampled parameters to the DataFrame
for key in param_ranges.keys():
    data[key] = param_values[key]

# Initialize an empty DataFrame to hold all results
final_df = pd.DataFrame()

for index, row in data.iterrows():
    m = row['m']
    v0 = row['v0']
    tau = row['tau']
    r = row['r']
    rho = row['rho']
    kappa = row['kappa']
    theta = row['theta']
    sigma = row['sigma']

    # Call the fdm function
    option_price, moneyness, V_0 = fdm(time_to_maturity=tau, risk_free_rate=r,
                                       reversion_speed=kappa, long_average_variance=theta,
                                       vol_of_vol=sigma, correlation=rho,
                                       m=m, v0=v0)

    # Create a DataFrame for this parameter set
    df = pd.DataFrame({
        'moneyness': [moneyness],
        'initial_variance': [V_0],
        'tau': [tau],
        'r': [r],
        'rho': [rho],
        'kappa': [kappa],
        'theta': [theta],
        'sigma': [sigma],
        'option_price': [option_price]
    })

    # Append the data from this iteration to the final DataFrame
    final_df = pd.concat([final_df, df], ignore_index=True)

    # Optional progress update
    if (index + 1) % 100 == 0:
        print(f"Processed {index + 1} parameter sets")

# Write the final DataFrame to CSV once after the loop
output_filename = 'parameter_option_prices.csv'
final_df.to_csv(output_filename, mode='w', header=True, index=False)

end_time = time.time()
duration = end_time - start_time
print(f"Script ran for {duration:.2f} seconds")
print(f"Parameters and corresponding option prices saved to '{output_filename}'")
```

Listing A.3: Run FDM

```python
import pandas as pd

# Load the dataset
input_file = 'parameter_option_prices.csv'
df = pd.read_csv(input_file)

# Filter rows where option_price is above 10
filtered_df = df[df['option_price'] < 0.66]

# Save the filtered data to a new file, overwriting if it exists
output_file = 'filtered_option_prices.csv'
filtered_df.to_csv(output_file, index=False)

print(f"Filtered data saved to {output_file}")
```

Listing A.4: Filter in Python

Appendix B

```python
# build_model.py

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import initializers, regularizers

def build_model(inp_size, hidden_layers=2, nodes=100, dropout_rate=0.2,
    l2_reg=1e-4):
    """
    Builds a Sequential neural network model with regularisation.

    Parameters:
    - inp_size (int): Number of input features.
    - hidden_layers (int): Number of hidden layers.
    - nodes (int): Number of nodes in the first hidden layer.
    - dropout_rate (float): Fraction of neurons to drop.
    - l2_reg (float): L2 regularisation factor.

    Returns:
    - model (Sequential): Compiled Keras model.
    """
    # Initialize the Glorot uniform initializer
    glorot_uniform = initializers.GlorotUniform(seed=None)

    # Initialize the Sequential model
    model = Sequential()

    # Add the first hidden layer with input dimension
    model.add(Dense(
        nodes,
        input_dim=inp_size,
        kernel_initializer=glorot_uniform,
        activation='relu',
        kernel_regularizer=regularizers.l2(l2_reg)
    ))

    # Add Dropout
    model.add(Dropout(rate=dropout_rate))

    # Add additional hidden layers if specified
    for i in range(hidden_layers - 1):
        # Decrease the number of nodes in subsequent layers
        nodes_ = max(int(nodes / (i + 2)), 10)  # Ensure at least 10
    nodes
```

```
        model.add(Dense(
            nodes_,
            kernel_initializer=glorot_uniform,
            activation='relu',
            kernel_regularizer=regularizers.l2(l2_reg)
        ))
        model.add(Dropout(rate=dropout_rate))

    # Add the output layer
    model.add(Dense(
        1,
        kernel_initializer=initializers.RandomNormal(),
        activation='linear'  # Suitable for regression tasks
    ))

    # Compile the model with mean squared error loss and Adam optimizer
    model.compile(
        loss='mean_squared_error',
        optimizer='adam',
        metrics=['mean_squared_error']
    )

    return model
```

Listing B.1: Build model in Python

```
import pandas as pd
from sklearn.model_selection import train_test_split
from buildmodel import build_model
import tensorflow as tf
import joblib
import json
from tensorflow.keras.callbacks import EarlyStopping
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score

def train_neural_network(input_file='Fourier_Prices.csv',
                         model_save_path='trained_model.keras',
                         epochs=200,
                         batch_size=256,
                         hidden_layers=2,
                         nodes=700,
                         dropout_rate=0.1,
                         l2_reg=2.3553594488703456e-05,
                         test_size=0.20,
                         random_state=123,
                         data_fraction=1.0):
    """
    Trains a neural network on the standardised dataset and saves
   predictions.

    Parameters:
    - input_file (str): Path to the standardised CSV file.
    - model_save_path (str): Path to save the trained model.
    - epochs (int): Number of training epochs.
    - batch_size (int): Size of training batches.
    - hidden_layers (int): Number of hidden layers in the model.
    - nodes (int): Number of nodes in the first hidden layer.
    - dropout_rate (float): Dropout rate for regularisation.
    - l2_reg (float): L2 regularisation factor.
    - test_size (float): Proportion of the dataset to include in the test split.
    - random_state (int): Seed used by the random number generator.
    - data_fraction (float): Fraction of the dataset to use for training. Default is 1.0 (use all data).
    """
    # Load the standardised dataset
    df = pd.read_csv(input_file)
    print(f"Loaded standardised data from ’{input_file}’ with shape {df.shape}")

    # Use a fraction of the data
    if data_fraction < 1.0:
        df = df.sample(frac=data_fraction, random_state=random_state).reset_index(drop=True)
        print(f"Using {data_fraction*100:.0f}% of the data: {df.shape[0]} rows")

    # Define feature columns and target column
    feature_columns = [’moneyness’, ’initial_variance’, ’tau’, ’r’, ’rho’, ’kappa’, ’theta’, ’sigma’]
    target_column = ’option_price’

    # Separate features and target
    X = df[feature_columns].values
    y = df[target_column].values
    print(f"Features shape: {X.shape}, Target shape: {y.shape}")

    # Split into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=test_size,
        random_state=random_state
    )
    print(f"Training set: {X_train.shape}, Test set: {X_test.shape}")

    # Build the neural network model
    model = build_model(
        inp_size=X_train.shape[1],
        hidden_layers=hidden_layers,
        nodes=nodes,
        dropout_rate=dropout_rate,
        l2_reg=l2_reg
    )
    print("Neural network model has been built.")

    # Display the model architecture
    model.summary()

    # Define Early Stopping
    early_stopping = EarlyStopping(
        monitor=’val_loss’,
        patience=20,  # Number of epochs with no improvement after which training will be stopped
        restore_best_weights=True,
        verbose=1
    )
```

```python

    # Train the model
    print("Starting training...")
    history = model.fit(
        X_train, y_train,
        epochs=epochs,
        batch_size=batch_size,
        verbose=1,
        validation_data=(X_test, y_test),
        callbacks=[early_stopping]
    )
    print("Training completed.")

    # Save the trained model
    model.save(model_save_path)
    print(f"Trained model saved to '{model_save_path}'")

    # Save the training history for future analysis
    with open('training_history.json', 'w') as f:
        json.dump(history.history, f)
    print("Training history saved to 'training_history.json'")

    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Save actual and predicted values to a JSON file for future
   analysis
    predictions = {
        "actual_values": y_test.tolist(),
        "predicted_values": y_pred.flatten().tolist()  # Flatten to
   avoid nested arrays
    }
    with open('predictions.json', 'w') as f:
        json.dump(predictions, f)
    print("Predictions saved to 'predictions.json'")

    # Calculate and print the MSE and R-squared value
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
    print(f"Best MSE on the test set: {mse}")
    print(f"Corresponding R-squared value: {r2}")

if __name__ == "__main__":
    # Example: Train with 25% of the data
    train_neural_network(data_fraction=1)
```

Listing B.2: Train model in Python