UNIVERSITY OF BIRMINGHAM

BIRMINGHAM BUSINESS SCHOOL

2019-2020

MSc DISSERTATION COVER SHEET

I confirm that I have read and understood the regulations on plagiarism* and have acknowledged the work of others that I have included in this dissertation.

Please read the following statement and **tick ONE box** regarding permission, or denial thereof, to view your dissertation by other students:

- **I AGREE** to allow my dissertation to be seen by future students. ☑

By signing this form, I agree to allow access to students of the Business School, as part of the University of Birmingham, to view my dissertation, or part thereof, for guidance as an example of good practice. For its part, the University will grant access to Birmingham Business School students as it deems appropriate, but in so doing forbids anyone to copy or use my dissertation in any other way or for any other purpose.
I understand that my dissertation will be available to view via Canvas and that any personal references will be anonymised.
I further understand that the University has no control over the actions of third parties, and should I have any concerns, my permission may be withdrawn, at any time, by advising the Business School in writing.

- **I DO NOT AGREE** to allow my dissertation to be seen by future students. ☐

Print Name: Chun Kiat ONG

Student ID: 2031611

Date: 9th September 2020

----

*Plagiarism, in this context, is the reproduction of material from books and articles without acknowledgement. It is the act of passing off another person's work as your own, copying a fellow student's work or reproducing work submitted by a past student. Such actions are seen as a form of cheating and, as such, are penalised by examiners according to their extent and gravity.

You should not quote existing work without quotation marks and appropriate referencing. An attempt to present the work of someone else as your own may lead to your dissertation being awarded a mark of zero. You are required to state the full references of all sources that you use. If quotations are made, they must be explicitly and fully referenced, including stating the relevant page number(s). You will be penalised very severely if examiners find that you have presented a section of a book, an article or a paper without appropriate referencing. If you are not sure about how to quote an existing work, please ask for advice from your supervisor.

UNIVERSITY OF BIRMINGHAM

MASTER OF SCIENCE THESIS

The Performance of Artificial Neural Networks on Rough Heston Model

Author:
Chun Kiat ONG
(ID:2031611)

Supervisor:
Dr. Daniel J.DUFFY

A thesis submitted in partial fulfillment of the requirements for the degree of MSc Mathematical Finance in the
Birmingham Business School

9th September, 2020

Declaration of Authorship

I, Chun Kiat ONG (ID:2031611), declare that this thesis titled, "The Performance of Artificial Neural Networks on Rough Heston Model" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: CHUN KIAT ONG

Date: 9th September 2020

UNIVERSITY OF BIRMINGHAM

Abstract

Birmingham Business School

MSc Mathematical Finance

The Performance of Artificial Neural Networks on Rough Heston Model

by Chun Kiat ONG (ID:2031611)

There has been extensive research on finding various methods to price option more accurately and more quickly. The rise of artificial neural networks could provide an alternative means for this application. Although there exist many methods for approximating option prices under the rough Heston model, our objective is to investigate the performance of artificial neural networks on rough Heston model option pricing as well as implied volatility approximations since it is part of the calibration process. We use simulated data to train the neural network instead of real market data for regulatory reason. We also adapt the image-based implicit training method for ANN implied volatility approximations where the output of the ANN is the entire implied volatility surface. The results shows that artificial neural networks can indeed generate accurate option prices and implied volatility approximations but it is not as efficient as the existing methods. Hence, we conclude that ANN is a feasible alternative method for pricing option under the rough Heston model but an ineffective one.

Acknowledgements

I would like to take a moment to thank those who helped me throughout the writing of this dissertation.

First, I wish to thank my supervisor, Dr. Daniel Duffy for his invaluable support and expertise guidance. I would also like to thank Dr. Colin Rowat for designing such a challenging and rigorous course. I truly enjoyed it.

I would also like to take this opportunity to acknowledge the support and great love of my family, my girlfriend and my friends. They kept me going on and this work would not have been possible without them.

Contents

List of Figures

List of Tables

List of Abbreviations

NN	**N**eural **N**etworks
ANN	**A**rtificial **N**eural **N**etworks
DNN	**D**eep **N**eural **N**etworks
CNN	**C**onvolutional **N**eural **N**etworks
SGD	**S**tochastic **G**radient **D**escent
SV	**S**tochastic **V**olatility
CIR	**C**ox, **I**ngersoll, and **R**oss
fBm	**f**ractional **B**rownian **m**otion
FSV	**F**ractional **S**tochastic **V**olatility
RFSV	**R**ough **F**ractional **S**tochastic **V**olatility
FFT	**F**ast **F**ourier **T**ransform
ELU	**E**xponential **L**inear **U**nit
ReLU	**R**ectified **L**inear **U**nit

Chapter 0

Project's Overview

This chapter intends to give a brief overview of the methodology of the project without delving too much into the details.

0.1 Project's Overview

It was advised by the supervisor to divide the problem into smaller, achievable steps like (Mandara, 2019) did. The 4 main steps are:

- Define the objective.
- Formulate the procedures.
- Implement the procedures.
- Evaluate the results.

The objective is to investigate the performance of artificial neural networks (ANN) on vanilla option pricing and implied volatility approximation under the rough Heston model. The performance metrics in this context are accuracy and run-time performance.

Figure 1 shows the flow chart of implementation procedures. The procedure starts with synthetic data generation. Then, we split the synthetic data set into 3 portions - one for training (in-sample), one for validation and the last one for testing (out-of-sample). Before the data is fed into the ANN for training, it requires to be scaled or normalised as it has different magnitude and range. In this thesis, we use the training method proposed by (Horvath et al., 2019), the image-based implicit training approach for implied volatility surface approximation. Then, we test the trained ANN model with unseen data set to evaluate its accuracy.

The run-time performance of ANN will be compared with that from the traditional approximation methods. This project requires the use of two programming languages, C++ and Python. The data generation process is done in C++. The wide range of machine learning libraries (Keras, TensorFlow, PyTorch) in Python makes it the logical choice for the neural networks computations. The experiment starts with the classical Heston model as a concept validation first and then proceed to the rough Heston model.

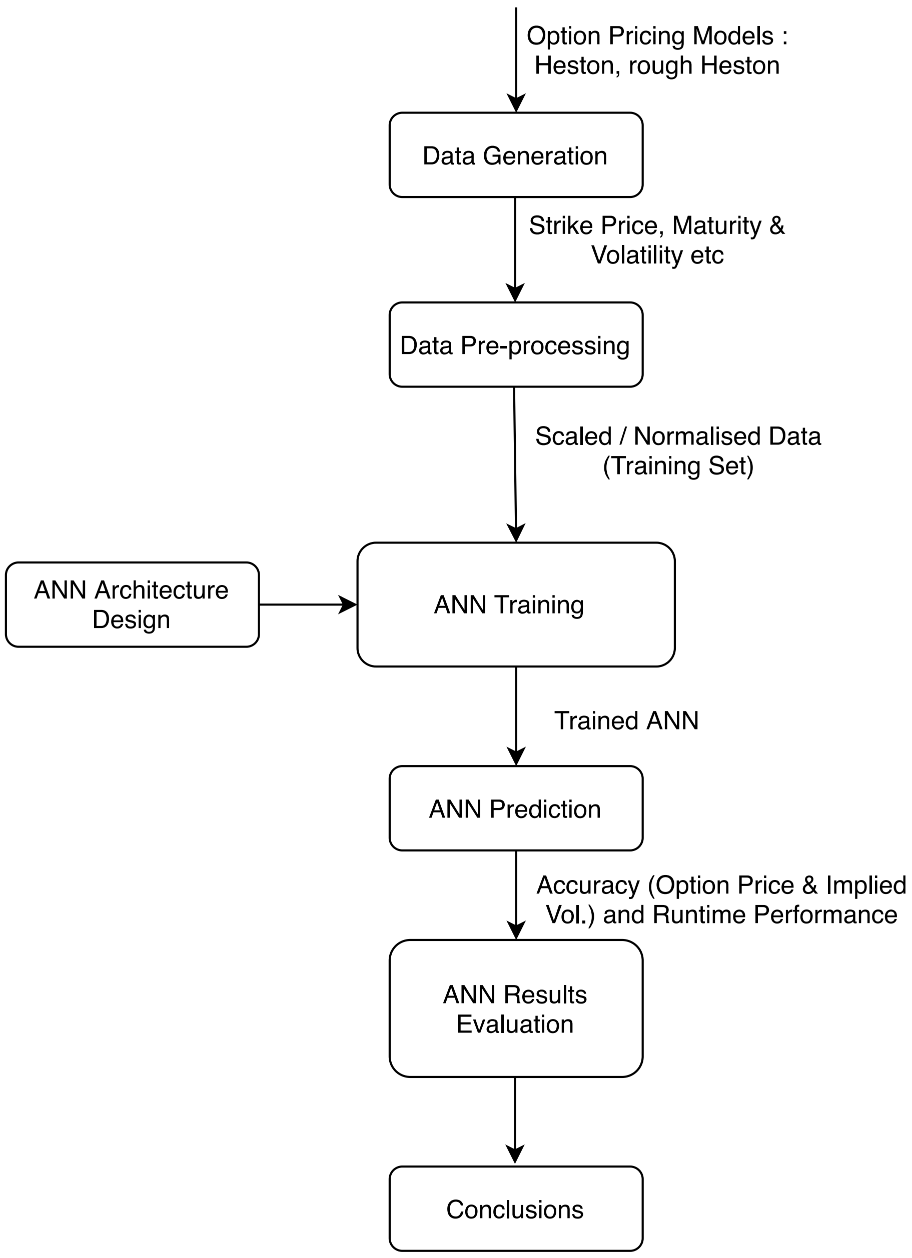

Option Pricing Models : Heston, rough Heston
Data Generation
Strike Price, Maturity & Volatility etc
Data Pre-processing
Scaled / Normalised Data (Training Set)
ANN Architecture Design
ANN Training
Trained ANN
ANN Prediction
Accuracy (Option Price & Implied Vol.) and Runtime Performance
ANN Results Evaluation
Conclusions

FIGURE 1: Implementation Flow Chart.

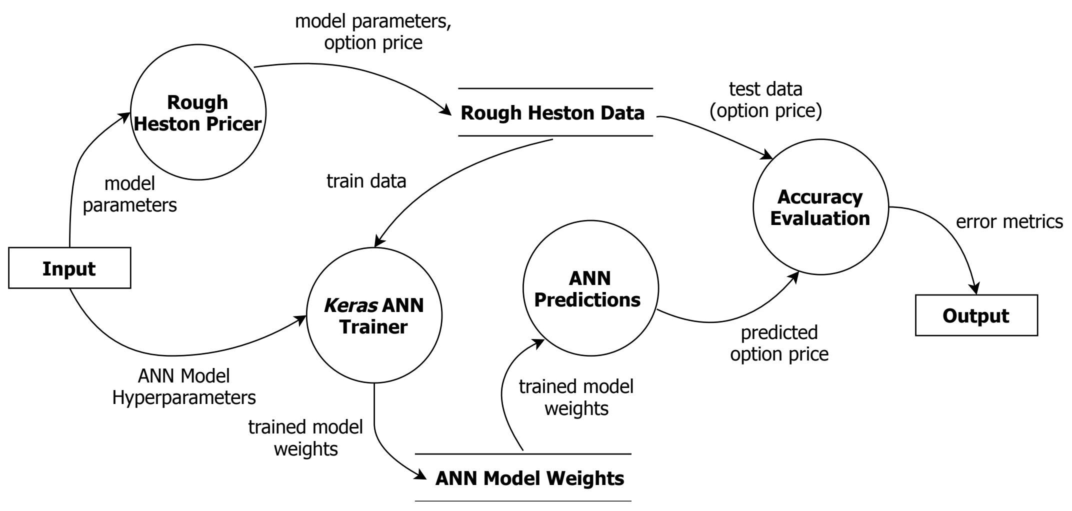


FIGURE 2: Data flow diagram for ANN on Rough Heston Pricer.

Chapter 1

Introduction

Option pricing has always been a major research area in financial mathematics. Various methods and techniques have been developed to price options more accurately and more quickly since the introduction of the well-known Black-Scholes model in 1973 (Black et al., 1973). The Black-Scholes model has an analytical solution and has only one parameter: volatility required to be calibrated. This makes it very easy to implement. Its simplicity comes at the expense of many unrealistic assumptions. One of its biggest flaws is the assumption of constant volatility term. This assumption simply does not hold in the real market. With this assumption, the Black-Scholes model fails to capture the volatility dynamics exhibited by the market and so is unable to value option accurately.

In 1993, (Heston, 1993) developed a stochastic volatility model in an attempt to better capture the volatility dynamics. The Heston model assumes the volatility of an underlying asset to follow a geometric Brownian motion (Hurst parameter, $H = 1/2$). Similarly, the Heston model also has a closed-form analytical solution and this makes it a strong alternative to the Black-Scholes model.

Whilst the Heston model is more realistic than the Black-Scholes model, the generated option prices do not match the observed vanilla option prices consistently (Gatheral et al., 2014). Recent research shows that more accurate option prices can be estimated if the volatility process is modelled as a fractional Brownian motion with $H < 1/2$. When $H < 1/2$, the volatility process path appears to be "rougher" than when $H \geq 1/2$ and hence the volatility is describe as rough.

(Euch et al., 2019) introduce the rough volatility version of Heston model which combines the best of both worlds. However, the calibration of rough Heston model relies on the notoriously slow Monte Carlo method due to its non-Markovian nature. This obstacle is the reason that rough Heston struggles to find its way into industrial applications. Although there exist many approximation methods for rough Heston model to help to solve this problem, we are interested in the feasibility of machine learning algorithms.

Recent advancements of machine learning algorithms could potentially provide an alternative means for this problem. Specifically, the artificial neural networks (ANN) offers the best hope because it is capable of learning complex functional relationships between input and output data, and then making predictions instantly.

In this project, our objective is to investigate the accuracy and run-time performance of ANN on European call option price predictions and implied volatility approximations under the rough Heston model. We start off by using the Heston model as a concept validation then proceed to its rough volatility version. For regulatory purposes, we use data simulated by option pricing models for training.

1.1 Organisation of This Thesis

This thesis is organised as follows: In Chapter 2, we review some of the related work and provide justifications for our research. In Chapter 3, we discuss the theory of option pricing models of our interest. We provide a brief introduction of ANN and some techniques to improve their performance in Chapter 4. Then, we explain the tools we used for the data generation process and some useful techniques to speed up the process in Chapter 5. In the same chapter we also discuss about the important data pre-processing procedures. Chapter 6 shows our network architecture and experimental setups. We present the results and discuss our findings in Chapter 7. Finally, we conclude our findings and discuss about potential future work in Chapter 8.

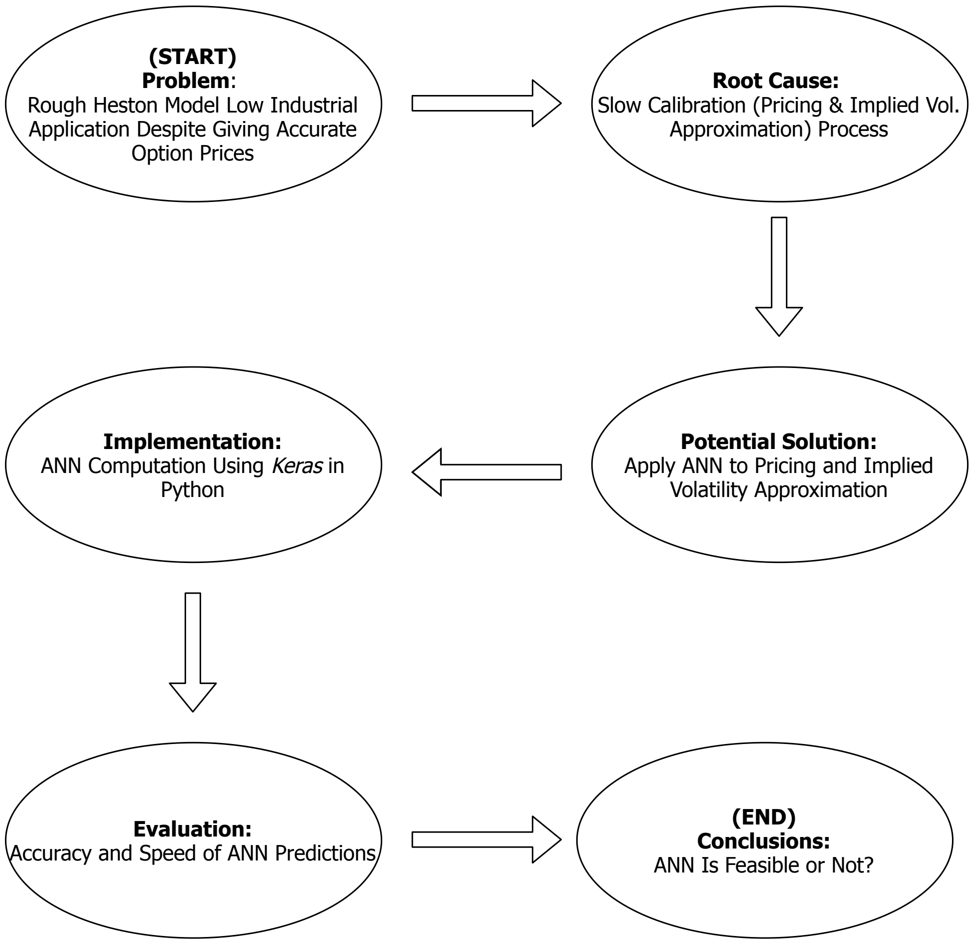

(START)
Problem:
Rough Heston Model Low Industrial Application Despite Giving Accurate Option Prices
Root Cause:
Slow Calibration (Pricing & Implied Vol. Approximation) Process
Potential Solution:
Apply ANN to Pricing and Implied Volatility Approximation
Implementation:
ANN Computation Using Keras in Python
Evaluation:
Accuracy and Speed of ANN Predictions
(END)
Conclusions:
ANN Is Feasible or Not?

FIGURE 1.1: Problem Solving Process.

Chapter 2

Literature Review

In this chapter we intend to provide a review of the current literature and highlight the research gap. The focus of our review is the application of ANN in option pricing models. We justify our methodology and our choice of option pricing models. We would also provide some critiques.

2.1 Application of ANN in Finance

Option pricing has always been a hot topic in academia and the financial industry. Researchers have been looking for various means to price option more accurately and quickly. As the computational technology advances, the application of ANN has gained popularity in solving complex problems in many different industries. In quantitative finance, the idea of utilising ANN for pricing derivatives comes from its ability to make fast and accurate predictions once it is trained.

The application of ANN in option pricing begun with (Hutchinson et al., 1994). The authors were the pioneers in applying ANN for pricing and hedging derivatives. Since then, there are more than 100 research papers on this topic. It is also worth noting that complexity of the ANN's architecture used in the research papers increases over the years. Whilst there has been much research on the applications of ANN in option pricing, there is only about 10% of papers focus on implied volatility (Ruf et al., 2020). The feasibility of using ANN for implied volatility approximation is equally as important as for option pricing since they are both part of the calibration process. Recently, there are more research papers focus on implied volatility and calibration, such as (Mostafa et al., 2008), (Liu et al., 2019a), (Liu et al., 2019b) and (Horvath et al., 2019). A key contribution of (Horvath et al., 2019)'s work is the introduction of imaged-based implicit training method for implied volatility surface approximation. This method is allegedly robust and fast for approximating the entire implied volatility surface.

A significant portion of the research papers uses real market data in their research. Currently there is no standardised framework for deciding the architecture and parameters of ANN. Hence, training the ANN directly with market data might pose some issues when it comes to regulatory compliance. (Horvath et al., 2019) is one of the few papers that uses simulated data for ANN training. The simulated data is generated from Heston and rough Bergomi models. Using simulated data removes the regulatory concerns as the functional mapping from model parameters to option price is known and tractable.

The application of ANN requires splitting the entire data set for training, validation and testing. Most of the papers that use real market data ensure the data splitting process is chronological to preserve the time series structure of the data. This is not a concern in our case we as are using simulated data.

The results in (Horvath et al., 2019) shows very accurate predictions. However, after examining the source code, we discover that the ANN is validated and tested on the same data set. Thus, the high accuracy results does not give any information about how well the trained ANN generalises to unseen data. In our experiment, we will use different data sets for validation and testing.

Common error metrics used include mean absolute error (MAE), mean absolute percentage error (MAPE) and mean squared error (MSE). We suggest a new metric, maximum absolute error as it is useful in a risk management perspective for quantifying the largest prediction error in the worst case scenario.

2.2 Option Pricing Models

In terms of the choice of option pricing models, over 80% of the papers select Black-Scholes model or its extensions in their research (Ruf et al., 2020). As mentioned before, there are other models that can give better option prices than Black-Scholes such as the Stochastic Volatility models. (Liu et al., 2019b) investigated the application of ANN in Heston and Bates models.

According to (Gatheral et al., 2014), modelling the volatility as rough volatility can capture the volatility dynamics in the market better. Rough volatility models struggle to find their ways into the industry due to their slow calibration process. Hence, it is natural to for the direction of ANN research to move towards rough volatility models.

Some of the recent examples include (Stone, 2019), (Bayer et al., 2018) and (Horvath et al., 2019). (Stone, 2019) investigates the calibration of rough Bergomi model using Convolutional Neural Networks (CNN) whilst (Bayer et al., 2018) study the calibration of Heston and rough Bergomi models using Deep Neural Networks (DNN). Contrary to these papers, where they perform direct calibration via ANN in one step, (Horvath et al., 2019) split the calibration of rough Bergomi model into two steps. Firstly, the ANN is trained to learn the pricing equation during a so-called offline session. Then, the trained ANN is deterministic and stored for online calibration session later. This approach can speed up the online calibration by a huge margin.

2.3 The Research Gap

In our thesis, we will be using the the approach in (Horvath et al., 2019). That is to train the network to learn the pricing and implied volatility functional mappings separately. We would leave the model calibration step as the future outlook of this project. We choose to work with rough Heston model, another popular rough volatility model that has not been investigated yet. We would adapt the image-based implicit training method in our research. To re-iterate, we would validate and test our ANN with different data set to obtain a true measure of ANN's performance on unseen data.

Chapter 3

Option Pricing Models

In this chapter we discuss about the option pricing models of our interest: Heston and rough Heston models. The idea of modelling volatility as rough fractional Brownian motion will also be presented. Then, we derive their pricing and implied volatility equations, and also explain their implementations for data generation.

The Black-Scholes model is simple but unrealistic for real world applications as its assumptions simply do not hold. One of its greatest criticisms is the assumption of constant volatility of the underlying asset price. (Dupire, 1994) extended the Black-Scholes framework to give local volatility models. He modelled the volatility as a deterministic function of the underlying price and time. The function is selected such that its outputs match observed European option prices. This extension is time-inhomogeneous and the dynamics it exhibits is still unrealistic. Thus, it is not able to produce future volatility surfaces that emulate our observations.

Stochastic Volatility (SV) models were introduced in which the volatility of the underlying is modelled as a geometric Brownian motion. Recent research shows that rough volatility can capture the true volatility dynamics better and so this leads to the rough volatility models.

3.1 The Heston Model

One of the most popular SV models is the one proposed by (Heston, 1993), its popularity is due to the fact that its pricing equation for European options is analytically tractable. The property allows calibration procedures to be done efficiently.

The Heston model for a one-dimensional spot price, S follows a stochastic process

$$dS = \mu S dt + \sqrt{\nu} S dW_1 \tag{3.1}$$

And its instantaneous variance, ν follows a Cox, Ingersoll, and Ross (CIR) process (Cox et al., 1985)

$$d\nu = \kappa(\theta - \nu)dt + \xi\sqrt{\nu}dW_2 \tag{3.2}$$

$$\langle dW_1, dW_2 \rangle = \rho dt \tag{3.3}$$

where

- μ is the (deterministic) rate of return of the spot
- θ is the long term mean volatility
- ξ is the volatility of volatility
- κ is the mean reversion rate of volatility.
- ρ is the correlation between spot and volatility moves
- W_1 and W_2 are Wiener processes

3.1.1 Option Pricing Under Heston Model

In this section, we follow (Gatheral, 2006) closely. We start by forming a portfolio Π containing option being priced V, number of stocks $-\Delta$ and a quantity of $-\Delta_1$ of another asset whose value is V_1

$$\Pi = V - \Delta S - \Delta_1 V_1$$

The change in portfolio during dt is

$$\begin{aligned} d\Pi = &\left[\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V}{\partial \nu^2 \partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V}{\partial \nu^2}\right] dt \\ &- \Delta_1\left[\frac{\partial V_1}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V_1}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V_1}{\partial \nu \partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V_1}{\partial \nu^2}\right] dt \\ &+ \left[\frac{\partial V}{\partial S} - \Delta_1\frac{\partial V_1}{\partial S} - \Delta\right] dS + \left[\frac{\partial V}{\partial \nu} - \Delta_1\frac{\partial V_1}{\partial \nu}\right] d\nu \end{aligned}$$

Note the explicit dependence on t of S and ν has been removed for simplicity.

We can make the portfolio instantaneously risk-free by eliminating dS and $d\nu$ terms

$$\frac{\partial V}{\partial S} - \Delta_1\frac{\partial V_1}{\partial S} - \Delta = 0$$

And,

$$\frac{\partial V}{\partial \nu} - \Delta_1\frac{\partial V_1}{\partial \nu} = 0$$

So, we are left with

$$\begin{aligned} d\Pi = &\left[\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V}{\partial \nu \partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V}{\partial \nu^2}\right] dt \\ &- \Delta_1\left[\frac{\partial V_1}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V_1}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V_1}{\partial \nu \partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V_1}{\partial \nu^2}\right] dt \\ =& r\Pi dt \\ =& r(V - S - \Delta_1 V_1)dt \end{aligned}$$

Then, we apply the no arbitrage principle: the return of a risk-free portfolio must be equal to the risk-free rate. And we assume the risk-free rate to be deterministic for our case.

Rearranging the above equation by collecting V terms on the left-hand side and V_1 terms on the right-hand side

$$\frac{\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V}{\partial\nu\partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V}{\partial\nu^2} + rS\frac{\partial V}{\partial S} - rV}{\frac{\partial V}{\partial\nu}}$$
$$= \frac{\frac{\partial V_1}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V_1}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V_1}{\partial\nu\partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V_1}{\partial\nu^2} + rS\frac{\partial V_1}{\partial S} - rV_1}{\frac{\partial V_1}{\partial\nu}}$$

It is obvious that the ratio is equal to some function of S, ν and t: $f(S,\nu,t)$. Thus, we have:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V}{\partial\nu\partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V}{\partial\nu^2} + rS\frac{\partial V}{\partial S} - rV = f\frac{\partial V}{\partial\nu}$$

In the case of Heston model, f is chosen as:

$$f = -(\kappa(\theta-\nu) - \lambda\sqrt{\nu})$$

where $\lambda(S,\nu,t)$ is called the market price of volatility risk. We do not delve into the choice of f and the concept of $\lambda(S,\nu,t)$ any further here. See (Wilmott, 2006) for his arguments.

(Heston, 1993) assumed in his paper that $\lambda(S,\nu,t)$ is linear in the instantaneous variance ν_t to retain the form of the equation under the transformation from the statistical measure to the risk-neutral measure. Here, we are only interested in pricing the options, so we can set $\lambda(S,\nu,t)$ to be zero (Gatheral, 2006).

Hence, we arrived at:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\nu S^2\frac{\partial^2 V}{\partial S^2} + \rho\xi\nu S\frac{\partial^2 V}{\partial\nu\partial S} + \frac{1}{2}\xi^2\nu\frac{\partial^2 V}{\partial\nu^2} + rS\frac{\partial V}{\partial S} - rV = \kappa(\nu-\theta)\frac{\partial V}{\partial\nu} \quad (3.4)$$

According to (Heston, 1993), the price of an undiscounted European call option price, C with strike price, K and time to maturity, T has solution in the form:

$$C(S,K,\nu,T) = \frac{1}{2}(F-K) + \frac{1}{\pi}\int_0^\infty (F * f_1 - K * f_2)du \quad (3.5)$$

where

$$f_1 = Re\left[\frac{e^{-iu\ln K}\varphi(u-i)}{iuF}\right]$$
$$f_2 = Re\left[\frac{e^{-iu\ln K}\varphi(u)}{iu}\right]$$
$$\varphi(u) = \mathbb{E}(e^{iu\ln S_T})$$
$$F = Se^{\mu T}$$

The evaluation of 3.5 requires the computation of Fourier inversion integrals. And it is susceptible to numerical instabilities due to the involved complex logarithms. (Kahl et al., 2006) proposed an integration scheme to compute the Fourier inversion integral by applying some transformations to the asymptotic structure. Thus, the solution becomes:

$$C(S,K,v,T) = \int_0^1 y(x)dx, \quad x \in \mathbb{R} \tag{3.6}$$

where

$$y(x) = \frac{1}{2}(F-K) + \frac{F * f_1\left(-\frac{\ln x}{C_\infty}\right) - K * f_2\left(-\frac{\ln x}{C_\infty}\right)}{x * \pi * C_\infty}$$

The limits at the boundaries of the integral are:

$$\lim_{x\to 0} y(x) = \frac{1}{2}(F-K)$$

and

$$\lim_{x\to 1} y(x) = \frac{1}{2}(F-K) + \frac{F * \lim_{u\to 0} f_1(u) - K * \lim_{u\to 0} f_2(u)}{\pi * C_\infty}$$

Equation 3.6 provides a more robust pricing method for moderate to long maturities or strong mean-reversion options. For the full derivation of equation 3.6, see (Kahl et al., 2006). The source code to implement equation 3.6 was provided by the supervisor and is the source code for (Duffy et al., 2012).

3.1.2 Heston Model Implied Volatility Approximation

Before the model is used for pricing options, the model's parameters need to be calibrated so that it can return current market prices (at least to some degree of accuracy). That is, the unmeasurable model parameters are obtained by calibrating to implied volatility that are observed on the market. Hence, this motivates the need for fast implied volatility computation.

Unfortunately, there is no closed-form analytical formula for calculating implied volatility Heston model. There are however, many closed-form approximations for Heston implied volatility.

In this project, we select the implied volatility approximation method for Heston proposed by (Lorig et al., 2019) that allegedly outperforms other methods such as the one by (Fouque et al., 2012). (Lorig et al., 2019) derive a family of asymptotic expansions for European-style option implied volatility. Their method provides an explicit approximation and requires no special functions, not even numerical integration and so it is faster to compute than the counterparts.

We follow closely the derivation steps outlined in (Lorig et al., 2019). Consider the Heston model in Section (3.1). According to (Andersen et al., 2007), ρ must be set to negative to avoid moment explosion. To circumvent this, we apply the following change of variables $(X(t), V(t)) := (\ln S(t), e^{\kappa t}\nu)$. Using the asymptotic expansions for general stochastic models (see Appendix B) and Ito's formula we have

$$dX = -\frac{1}{2}e^{-\kappa t}V dt + \sqrt{e^{-\kappa t}V} dW_1, \qquad X(0) = x := \ln s \tag{3.7}$$

$$dV = \theta\kappa e^{\kappa t} dt + \xi\sqrt{e^{\kappa t}V} dW_2, \qquad V(0) = \nu := z > 0 \tag{3.8}$$

$$\langle dW_1, dW_2 \rangle = \rho dt \tag{3.9}$$

The parameters ν, κ, θ, ξ, W_1, W_2 play the same role as in Section (3.1). Then, we apply the second order differential operator $\mathcal{A}(t)$ to (X, V):

$$\mathcal{A}(t) = \frac{1}{2}e^{-\kappa t}v(\partial_x^2 - \partial_x) + \theta\kappa e^{\kappa t}\partial_v + \frac{1}{2}\xi^2\xi e^{\kappa t}v\partial_v^2 + \xi\rho\nu\partial_x\partial_v$$

Define T as time to maturity, k as log-strike. Using the time-dependent Taylor series expansion of $\mathcal{A}(T)$ with $(\bar{x}(T), \bar{v}(T)) = (X_0, \mathbb{E}[V(T)]) := (x, \theta(e^{\kappa T} - 1))$ to give (see Appendix B)

$$\begin{aligned}
\sigma_0 =& \sqrt{\frac{-\theta + \theta\kappa T + e^{-\kappa T}(\theta - v) + v}{\kappa T}} \\
\sigma_1 =& \frac{\xi\rho z e^{-\kappa T}(-2\bar{v} - \bar{v}\kappa T - e^{\kappa T}(\bar{v}(\kappa T - 2) + v) + \kappa T v + v)}{\sqrt{2}\kappa^2\sigma_0^2 T^{\frac{3}{2}}} \\
\sigma_2 =& \frac{\xi^2 e^{-2\kappa T}}{32\kappa^4\sigma_0^5 T^3}[-2\sqrt{2}\kappa\sigma_0^3 T^{\frac{3}{2}} z(-\bar{v} - 4e^{\kappa T}(\bar{v} + \kappa T(\bar{v} - v)) + e^{2\kappa T}(\bar{v}(5 - 2\kappa T) - 2v) + 2v) \\
& + \kappa\sigma_0^2 T(4z^2 - 2)(\bar{v} + e^{2\kappa T}(-5\bar{v} + 2\bar{v}\kappa T + 8\rho^2(\bar{v}(\kappa T - 3) + v) + 2v)) \\
& + \kappa\sigma_0^2 T(4z^2 - 2)(4e^{\kappa T}(\bar{v} + \bar{v}\kappa T + \rho^2(\bar{v}(\kappa T(\kappa T + 4) + 6) - v(\kappa T(\kappa T + 2) + 2)) \\
& - \kappa T v) - 2v) \\
& + 4\sqrt{2}\rho^2\sigma_0\sqrt{T}z(2z^2 - 3)(-2 - \bar{v}\kappa T - e^{\kappa T}(\bar{v}(\kappa T - 2) + v) + \kappa T v + v)^2 \\
& + 4\rho^2(4(z^2 - 3)z^2 + 3)(-2\bar{v} - \bar{v}\kappa T - e^{\kappa T}(\bar{v}(\kappa T - 2) + v) + \kappa T v + v)^2] \\
& - \frac{\sigma_1^2(4(x - k)^2 - \sigma_0^4 T^2)}{8\sigma_0^3 T}
\end{aligned}$$

where

$$z = \frac{x - k - \frac{\sigma_0^2 T}{2}}{\sigma_0\sqrt{2T}}$$

The explicit expression for σ_3 is too long to be included here. See (Lorig et al., 2019) for the full derivation. The explicit expression for the implied volatility approximation is:

$$\sigma_{implied} = \sigma_0 + \sigma_1 z + \sigma_2 z^2 + \sigma_3 z^3 \tag{3.10}$$

The C++ source code to implement this is available here: Heston Implied Volatility Approximation.

3.2 Volatility is Rough

We start this section by introducing the fractional Brownian motion (fBm), which is a generalisation of Brownian motion.

Definition 3.2.1 *Fractional Brownian Motion (fBm) is a centered Gaussian process $B_t^H, t \geq 0$ with the autocovariance function:*

$$\mathbb{E}[B_t^H B_s^H] = \frac{1}{2}(|t|^{2H} + |s|^{2H} - |t-s|^{2H}) \tag{3.11}$$

where $H \in (0,1)$, is the Hurst index or Hurst parameter associated with the fractional Brownian motion.

The Hurst parameter is a measure of the long-term memory of a time series. It was first introduced by (Hurst, 1951) for modelling water levels of the Nile river in order to determine the optimum dam size. As it turns out, it is also suitable for modelling time series data from other natural systems. (Peters, 1991) and (Peters, 1994) are some of the earliest examples of applying this concept in financial time series modelling. A time series can be classified into 3 categories based on the Hurst parameter

1. $0 < H < \frac{1}{2}$: Negative correlation in the increment/decrement of the process. A mean reverting process, which implies an increase in value is more likely followed by a decrease in value, and vice versa. The smaller the value the greater the mean-reverting effect.

2. $H = \frac{1}{2}$: No correlation in the increment/decrement of the process (geometric Brownian motion or Wiener process).

3. $\frac{1}{2} < H < 1$: Positive correlation in the increment/decrement of the process. A trend reinforcing process, which means the direction of the next value is more likely the same as current value. The strength of the trend is greater when H is closer to 1.

Empirical evidence shows that log-volatility time series behaves similar to a fBm, with Hurst parameter of order 0.1 (Gatheral et al., 2014). Essentially, all the statistical stylised facts of volatility can be captured when modelling it as a rough fBm.

This motivates the use of the fractional stochastic volatility (FSV) model by (Comte et al., 1998). Our model is called rough fractional stochastic volatility (RFSV) model to emphasise that $H < 1/2$. The term 'rough' refers to the roughness of the path of fBm. From Figure 3.1, one can notice that the fBm path gets 'rougher' as H decreases.

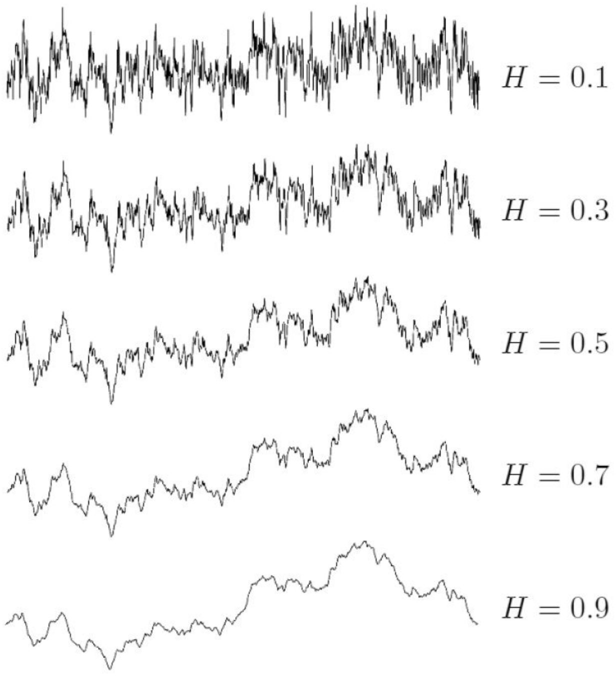


FIGURE 3.1: Paths of fBm for different values of H. Adapted from (Shevchenko, 2015).

3.3 The Rough Heston Model

In this section we introduce the rough volatility version of Heston model and derive its pricing and implied volatility equations.

As stated before, rough volatility models can fit the volatility surface notably well with very few parameters. Hence, it is natural to modify the Heston model and consider its rough version.

The rough Heston model for a one-dimensional asset price S with instantaneous variance $\nu(t)$ takes the form as in (Euch et al., 2016):

$$
\begin{gathered}
dS = S\sqrt{\nu}dW_1 \\
\nu(t) = \nu(0) + \frac{1}{\Gamma(\alpha)}\int_0^t (t-s)^{\alpha-1}\kappa(\theta-\nu(s))ds + \frac{\xi}{\Gamma(\alpha)}\int_0^t (t-s)^{\alpha-1}\sqrt{\nu(s)}dW_2 \\
\langle dW_1, dW_2 \rangle = \rho dt
\end{gathered}
\tag{3.12}
$$

where

- $\alpha = H + \frac{1}{2} \in (1/2, 1)$, governs the roughness of the volatility sample paths
- κ is the mean reversion rate
- θ is the long term mean volatility
- $\nu(0)$ is the initial volatility

- ξ is the volatility of volatility
- Γ is the Gamma function
- W_1 and W_2 are two Brownian motions with correlation ρ

When $\alpha = 1$ $(H = 0.5)$, we recover the classical Heston model in Chapter (3.1).

3.3.1 Rough Heston Pricing Equation

The pricing equation of rough Heston model is inspired by the classical Heston one. In classical Heston model, option price can be obtained by applying Fourier inversion integrals to the characteristic function that is expressed in terms of the solution of a Riccati equation. The rough Heston model displays a similar structure but with the Riccati equation being replaced by a fractional Riccati equation.

(Euch et al., 2016) derived a characteristic function for the rough Heston model; this result is particularly important as the non-Markovian nature of the fractional Brownian motion makes other numerical pricing methods (e.g. Monte Carlo) difficult to implement.

Definition 3.3.1 *The fractional integral of order $r \in (0,1]$ of a function f is*

$$I^r f(t) = \frac{1}{\Gamma(r)} \int_0^t (t-s)^{r-1} f(s) ds \tag{3.13}$$

whenever the integral exists, and the fractional derivative of order $r \in (0,1]$ is

$$D^r f(t) = \frac{1}{\Gamma(1-r)} \frac{d}{dt} \int_0^t (t-s)^{-r} f(s) ds \tag{3.14}$$

whenever it exists.

Then, we quote the results from (Euch et al., 2016) directly. For the full proof see Section 6 of (Euch et al., 2016).

Theorem 3.3.1 *For all $t \geq 0$, the characteristic function of the log-price $X_t = \ln \frac{S(t)}{S(0)}$ is*

$$\phi_X(a,t) = \mathbb{E}[e^{iaX_t}] = exp[\kappa\theta I^1 h(a,t) + \nu(0) I^{1-\alpha} h(a,t)] \tag{3.15}$$

where h is solution of the fractional Riccati equation

$$D^\alpha h(a,t) = -\frac{1}{2}a(a+i) + \kappa(ia\rho\xi - 1)h(a,t) + \frac{(\xi)^2}{2} h^2(a,t),\ I^{1-\alpha} h(a,0) = 0 \tag{3.16}$$

which admits a unique continuous solution. $a \in \mathcal{C}$, a suitable region in the complex plane (see Definition 3.3.2).

Definition 3.3.2 *We define the strip in the complex plane relevant to the computation of option prices characteristic function in Theorem (3.3.1)*

$$\mathcal{C} = \left\{ z \in \mathbb{C} : \Re(z) \geq 0, -\frac{1}{1-\rho^2} \leq \Im(z) \leq 0 \right\} \tag{3.17}$$

where $\Re$ and $\Im$ denote real and imaginary parts respectively.

There is no explicit solution to equation (3.16) so it has to be solved numerically. (Gatheral et al., 2019) present a simple rational approximation to the solution of the rough Heston Riccati equation, which is inevitably faster than the numerical schemes.

3.3.1.1 Rational Approximation of Rough Heston Riccati Solution

Firstly, the **short-time** series expansion of the solution by (Alos et al., 2017) can be written as

$$h_s(a,t) = \sum_{j=0}^{\infty} \frac{\Gamma(1+j\alpha)}{\Gamma[1+(j+1)\alpha]} \beta_j(a) \xi^j t^{(j+1)\alpha} \tag{3.18}$$

with

$$\begin{aligned} \beta_0(a) &= -\frac{1}{2}a(a+i) \\ \beta_k(a) &= \frac{1}{2}\sum_{i,j=0}^{k-2} \mathbb{1}_{i+j=k-2}\, \beta_i(a)\beta_j(a) \frac{\Gamma(1+i\alpha)}{\Gamma[1+(i+1)\alpha]} \frac{\Gamma(1+j\alpha)}{\Gamma[1+(j+1)\alpha]} \\ &\quad + i\rho a \frac{\Gamma[1+(k-1)\alpha]}{\Gamma(1+k\alpha)} \beta_{k-1}(a) \end{aligned}$$

Again, from (Alos et al., 2017), the **long-time** expansion is

$$h_l(a,t) = r_- \sum_{k=0}^{\infty} \gamma_k \frac{\xi t^{-k\alpha}}{A^k \Gamma(1-k\alpha)} \tag{3.19}$$

where

$$\begin{aligned} \gamma_1 &= -\gamma_0 = -1 \\ \gamma_k &= -\gamma_{k-1} + \frac{r_-}{2A} \sum_{i,j=1}^{\infty} \mathbb{1}_{i+j=k}\, \gamma_i \gamma_j \frac{\Gamma(1-k\alpha)}{\Gamma(1-i\alpha)\Gamma(1-j\alpha)} \\ A &= \sqrt{a(a+i) - \rho^2 a^2} \\ r_\pm &= -i\rho a \pm A \end{aligned}$$

Now, we can construct the global approximation. Using the same techniques in (Atkinson et al., 2011), the rational approximation of $h(a,t)$ is given by

$$h^{(m,n)}(a,t) = \frac{\sum_{i=1}^{m} p_i(\xi t^\alpha)^i}{\sum_{j=0}^{n} q_j(\xi t^\alpha)^j}, \quad q_0 = 1 \tag{3.20}$$

Numerical experiments in (Gatheral et al., 2019) showed $h^{(3,3)}$ is a good choice for high accuracy and fast computation. Thus, we can express explicitly

$$h^{(3,3)}(a,t) = \frac{p_1\xi(t^\alpha) + p_2\xi^2(t^\alpha)^2 + p_3\xi^3(t^\alpha)^3}{1 + q_1\xi(t^\alpha) + q_2\xi^2(t^\alpha)^2 + q_3\xi^3(t^\alpha)^3} \tag{3.21}$$

Thus, from equations (3.18) and (3.19) we have

$$\begin{aligned} h_s(a,t) =& \frac{\Gamma(1)}{\Gamma(1+\alpha)}\beta_0(a)(t^\alpha) + \frac{\Gamma(1+\alpha)}{\Gamma(1+2\alpha)}\beta_1(a)\xi(t^\alpha)^2 \\ &+ \frac{\Gamma(1+2\alpha)}{\Gamma(1+3\alpha)}\beta_2(a)\xi^2(t^\alpha)^3 + \mathcal{O}(t^{4\alpha}) \end{aligned}$$

$$h_s(a,t) = b_1(t^\alpha) + b_2(t^\alpha)^2 + b_3(t^\alpha)^3 + \mathcal{O}[(t^\alpha)^4]$$

and

$$\begin{aligned} h_l(a,t) &= \frac{r_-\gamma_0}{\Gamma(1)} + \frac{r_-\gamma_1}{A\Gamma(1-\alpha)\xi}\frac{1}{(t^\alpha)} + \frac{r_-\gamma_2}{A^2\Gamma(1-2\alpha)\xi^2}\frac{1}{(t^\alpha)^2} + \mathcal{O}\left[\frac{1}{(t^\alpha)^3}\right] \\ h_l(a,t) &= g_0 + g_1\frac{1}{(t^\alpha)} + g_2\frac{1}{(t^\alpha)^2} + \mathcal{O}\left[\frac{1}{(t^\alpha)^3}\right] \end{aligned}$$

Matching the coefficients of equation (3.21) to h_s and h_l forces the coefficients b_i and g_j to satisfy

$$p_1\xi = b_1 \tag{3.22}$$

$$(p_2 - p_1q_1)\xi^2 = b_2 \tag{3.23}$$

$$(p_1q_1^2 - p_1q_2 - p_2q_1 + p_3)\xi^3 = b_3 \tag{3.24}$$

$$\frac{p_3\xi^3}{q_3\xi^3} = g_0 \tag{3.25}$$

$$\frac{(p_2q_3 - p_3q_2)\xi^5}{(q_3^2)\xi^6} = g_1 \tag{3.26}$$

$$\frac{(p_1q_3^2 - p_2q_2q_3 - p_3q_1q_3 + p_3q_2^2)\xi^7}{(q_3^3)\xi^9} = g_2 \tag{3.27}$$

The solution of this system is

$$p_1 = \frac{b_1}{\xi} \tag{3.28}$$

$$p_2 = \frac{b_1^3 g_1 + b_1^2 g_0^2 + b_1 b_2 g_0 g_1 - b_1 b_3 g_0 g_2 + b_1 b_3 g_1^2 + b_2^2 g_0 g_2 - b_2^2 g_1^2 + b_2 g_0^3}{(b_1^2 g_2 + 2 b_1 g_0 g_1 + b_2 g_0 g_2 - b_2 g_1^2 + g_0^3)\xi^2} \tag{3.29}$$

$$p_3 = g_0 q_3 \tag{3.30}$$

$$q_1 = \frac{b_1^2 g_1 - b_1 b_2 g_2 + b_1 g_0^2 - b_2 g_0 g_1 - b_3 g_0 g_2 + b_3 g_1^2}{(b_1^2 g_2 + 2 b_1 g_0 g_1 + b_2 g_0 g_2 - b_2 g_1^2 + g_0^3)\xi} \tag{3.31}$$

$$q_2 = \frac{b_1^2 g_0 - b_1 b_2 g_1 - b_1 b_3 g_2 + b_2^2 g_2 + b_2 g_0^2 - b_3 g_0 g_1}{(b_1^2 g_2 + 2 b_1 g_0 g_1 + b_2 g_0 g_2 - b_2 g_1^2 + g_0^3)\xi^2} \tag{3.32}$$

$$q_3 = \frac{b_1^3 + 2 b_1 b_2 g_0 + b_1 b_3 g_1 - b_2^2 g_1 + b_3 g_0^2}{(b_1^2 g_2 + 2 b_1 g_0 g_1 + b_2 g_0 g_2 - b_2 g_1^2 + g_0^3)\xi^3} \tag{3.33}$$

3.3.1.2 Option Pricing Using Fast Fourier Transform

After solving the Riccati equation using the rational approximation equation (3.21), we can compute the characteristic function, $\phi_X(a,t)$ in equation (3.15). Once the characteristic function is obtained, many methods are available to compute call option prices, see (Carr and Madan, 1999), (Itkin, 2010), (Lewis, 2001) and (Schmelzle, 2010). In this project, we select the Carr and Madan approach which is a fast option pricing method that uses fast Fourier transform (FFT). Suppose $k = \ln K$ and $C(k,T)$ is the value of a European call option with strike K and maturity T. Unfortunately, the FFT cannot be applied directly to evaluate the call option price because the call pricing function is not square-integrable. A key contribution of (Carr and Madan, 1999) is the modification of the call pricing function with respect to k. With this modification, a whole range of option prices can be obtained with one inverse Fourier transform.

Consider the modified call price $c(k,T)$

$$c(k,T) := e^{\beta k} C(k,T),\ \beta > 0 \tag{3.34}$$

And its Fourier transfrom is

$$\psi_X(u,T) = \int_{-\infty}^{\infty} e^{iuk} c(k,T)dk, \quad u \in \mathbb{R}_{>0} \tag{3.35}$$

which can be expressed as

$$\psi_X(u,T) = \frac{e^{-rT}\phi_X[u-(\beta+1)i,T]}{\beta^2+\beta-u^2+i(2\beta+1)u} \tag{3.36}$$

where r is the risk-free rate, $\phi_X(.)$ is the characteristic function defined in equation (3.15).

The purpose of β is to assist the integrability of the modified call pricing function over the negative log-strike axis. Hence, a sufficient condition is that $\psi_X(0)$ being finite

$$\psi_X(0,T) < \infty \tag{3.37}$$

$$\implies \phi_X[-(\beta+1)i,T] < \infty \tag{3.38}$$

$$\implies exp[\theta\kappa I^1 h(-(\beta+1)i,T) + \nu(0)I^{1-\alpha}h(-(\beta+1)i,T)] < \infty \tag{3.39}$$

Using equation (3.39) we can determine the upper bound value of β such that condition (3.37) is satisfied. (Carr and Madan, 1999) suggest one quarter of the upper bound serves a good choice for β.

The call price $C(k,T)$ can be recovered by applying the inverse Fourier transform

$$C(k,T) = \frac{e^{-\beta k}}{2\pi}\int_{-\infty}^{\infty} \Re[e^{-iuk}\psi_X(u,T)]du \tag{3.40}$$

$$= \frac{e^{-\beta k}}{\pi}\int_{0}^{\infty} \Re[e^{-iuk}\psi_X(u,T)]du \tag{3.41}$$

The semi-infinite integral in the call price equation can be evaluated by numerical methods such as the trapezoidal rule and FFT as shown in (Kwok et al., 2012)

$$C(k,T) \approx \frac{e^{-\beta k}}{\pi}\sum_{j=1}^{N} e^{-iu_j k}\psi_X(u_j,T)\Delta u \tag{3.42}$$

where $u_j = (j-1)\Delta u,\ j = 1,2,\ldots,N$.

We end this section with a summary of the steps required to price call option under rough Heston model:

1. Compute the solution of fractional Riccati equation (3.16), h using equation (3.21).
2. Compute the characteristic function in equation (3.15).
3. Determine the suitable value for β using equation (3.39).
4. Apply the Fourier transform in equation (3.36).
5. Evaluate the call option price by implementing FFT in equation (3.42).

3.3.2 Rough Heston Model Implied Volatility

(Euch et al., 2019) present an almost instantaneous and accurate approximation method to compute rough Heston implied volatility. This method allows us to approximate rough Heston implied volatility by simply scaling the volatility of volatility parameter ν and feed this scaled parameter as input into the classical Heston implied volatility approximation equation in Chapter 3.1.2. For a given maturity T, The scaled volatility of volatility parameter $\tilde{\nu}$ is

$$\tilde{\nu} = \sqrt{\frac{3}{2H+2}}\frac{\nu}{\Gamma(H+\frac{3}{2})}\frac{1}{T^{\frac{1}{2}-H}} \tag{3.43}$$

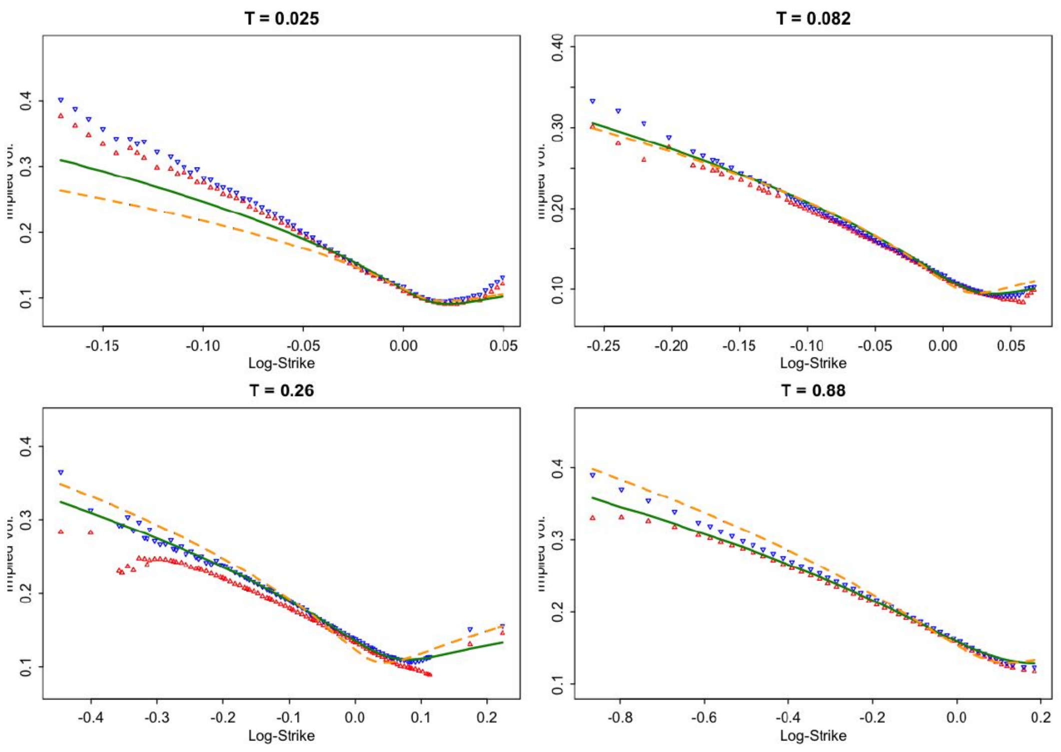


FIGURE 3.2: Implied Volatility Smiles of SPX as of 14th August 2013, with maturity T in years. Red and blue dots represent bid and ask SPX implied volatilities; green plots are from the calibrated rough Heston model ; dashed orange lines are from the classical Heston model calibrated to these smiles. Adapted from (Euch et al., 2019)

Chapter 4

Artificial Neural Networks

This chapter starts with an introduction to Artificial Neural Networks (ANN). We introduce the terminologies and basic elements of ANN, and explain the methods that we used to increase training speed and predictive accuracy. We also discuss about the training process for ANN, common pitfalls and their remedies. Finally, we discuss about the ANN type of interest and extend the discussions to Deep Neural Networks (DNN).

The concept of ANN has come a long way. It started off with modelling a simple neural network after electrical circuits, which was proposed by Warren McCulloch, a neurologist, and a young mathematician, Walter Pitts, in 1943 (McCulloch et al., 1943). This idea was further strengthen by Donald Hebb (Hebb, 1949).

ANN's ability to learn complex non-linear functional mappings by deciphering the pattern between independent and dependent variables has found its applications in many fields. With the computational resources becoming more accessible and the availability of big data set, ANN's popularity has grown exponentially in recent years.

4.1 Dissecting the Artificial Neural Networks

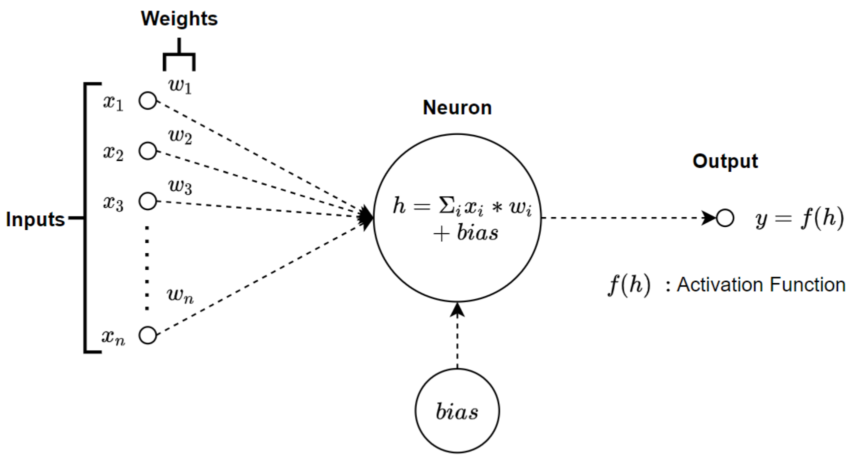


FIGURE 4.1: A Simple Neural Network

Before we delve deeper into the world of ANN, let us explain what do parameters and hyperparameters mean.

Parameters refer to the variables that are updated throughout the training process, they include weights and biases. Hyperparameters are the configuration variables that define the architecture of ANN and stay constant throughout the training process. Hyperparameters include number of neurons, hidden layers, activation functions and number of epochs etc.

4.1.1 Neurons

Like a biological neuron, an **artificial neuron** is the fundamental working unit in an ANN. It is essentially a mathematical function that receives multiple inputs and generates an output. More precisely, it takes the weighted sum of inputs and applies the activation function to the sum to generate an output. In the case of simple ANN (as shown in Figure 4.1), this will be the network output. In more sophisticated ANN, the output of a neuron can be the inputs of other neurons.

4.1.2 Input, Output and Hidden Layers

An **input layer** is the first layer of an ANN. It receives and sends the training data into the network for further processing. The number of input neurons is equal to the number of input features of training data.

The **output layer** is the final layer of the network. It outputs the predictions for a given set of input features. It can have more than one output neuron.

A **hidden layer** is the layer between input and output layers. It contains neuron(s) and receives weighted inputs from the input layer. Whilst ANN contains only one input layer and one output layer, the number of hidden layers can be more than one. One hidden layer can only be used for solving linearly separable problems. For more complex problems, multiple hidden layers are needed.

Systematic experimentation is required to identify the appropriate number of hidden layers to prevent under-fitting and over-fitting, and thus increasing predictive accuracy.

4.1.3 Connections, Weights and Biases

From input layer to output layer, each **connection** is assigned a weight that denotes its relative importance. Random **weights** will be initialised when training begins. As training continues these weights will be updated.

Sometimes a **bias** (different from the bias term in statistics) will be added to the neuron. It is merely a constant input with weight 1. Intuitively speaking, the purpose of a bias is to shift the activation function away from the origin, thereby allowing the neuron to give non-zero output when the inputs are 0.

4.1.4 Forward and Backward Propagation, Loss Functions

Forward propagation is the passing of input data in the forward direction through the network to generate output. Then, the difference between output and target output is estimated by a **loss function**. It is a measure of how well an ANN performs with the current set of weights. Typical loss functions for regression problems include mean squared error (MSE) and mean absolute error (MAE). They are defined as

Definition 4.1.1

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{i,predict} - y_{i,actual})^2 \tag{4.1}$$

Definition 4.1.2

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_{i,predict} - y_{i,actual}| \tag{4.2}$$

where $y_{i,predict}$ is the i-th predicted output and $y_{i,actual}$ is the i-th actual output.

Subsequently, this information is **propagated backward** from output layer to input layer and the gradients of the loss function w.r.t. the weights is computed. The algorithm then makes adjustments to the weights accordingly to minimise the loss function. An iteration is one complete cycle of forward and backward propagation.

ANN's training is an unconstrained optimisation problem with the loss function as the objective function. The algorithm navigates through the search space to find optimal weights to minimise the loss function (e.g. MSE). It can be expressed like so,

$$\min_{\bar{w}} \quad \frac{1}{n}\sum_{i=1}^{n}(y_{i,predict} - y_{i,actual})^2 \tag{4.3}$$

where $\bar{w}$ is the vector that contains the weights of all the connections within the ANN.

4.1.5 Activation Functions

An activation function is a mathematical function applied to the output of a neuron to determine whether it should be activated (or "fired") or not, based on the relevance of the neuron's inputs to the prediction. This is analogous to the neuronal firing activity in human's brain.

Activation functions can be linear or non-linear. Linear activation function is only suitable when the functional relationship is linear. Non-linear activation functions are used for most applications. In our case, non-linear activation functions are more suitable as the pricing and implied volatility functional mappings are complex.

The common activation function includes,

Definition 4.1.3 *Rectified Linear Unit (ReLU)*

$$f_{ReLU}(x) = \begin{cases} 0 & , \textit{if } x \leq 0 \\ x & , \textit{if } x > 0 \end{cases} \quad \in [0, \infty) \tag{4.4}$$

Definition 4.1.4 *Exponential Linear Unit (ELU)*

$$f_{ELU}(x) = \begin{cases} \alpha(e^x - 1) & , \textit{if } x \leq 0 \\ x & , \textit{if } x > 0 \end{cases} \quad \in (-\alpha, \infty) \tag{4.5}$$

where $\alpha > 0$.

The value of α is usually chosen to be 1 for most applications.

Definition 4.1.5 *Linear or identity*

$$f_{linear}(x) = x \quad \in (-\infty, \infty) \tag{4.6}$$

4.1.6 Optimisation Algorithms and Learning Rate

Learning rate refers to the amount of change is applied to the weights in each iteration of the training process. High learning rate results in fast training speed but there is risk of divergence whereas low learning rate may reduce the training speed.

Common **optimisation algorithms** for training ANN are: Stochastic Gradient Descent (SGD) and its extensions such as RMSProp and Adam. The SGD pseudocode is the following:

Algorithm 1 Stochastic Gradient Descent

```
Initialise a vector of random weights $\bar{w}$ and learning rate $l_r$.
while $Loss > error$ do
    Randomly shuffle the training data of size $n$.
    for $i = 1, 2, \cdots, n$ do
        $\bar{w}_{new} = \bar{w}_{current} - l_r \frac{\partial Loss}{\partial \bar{w}_{current}}$
    end for
end while
```

SGD is popular because it is simple to implement and can provide good results for most applications. In SGD, the learning rate l_r is fixed throughout the process. This is found to be an issue because the appropriate learning rate is difficult to determine. Selecting a high learning rate is prone to divergence whereas using a low learning rate can result in slow training process. Hence, improvements have been made to SGD to allow the learning rate to be adaptive. Root Mean Square Propagation (RMSProp) is one of the the extensions that the learning rate to be variable throughout the training process (Hinton et al., 2015). It is a method that divides the learning rate by the exponential moving average of past gradients. The author suggests the exponential decay rate to be $b = 0.9$ and its pseudocode is presented below:

Algorithm 2 Root Mean Square Propagation (RMSProp)

```
Initialise a vector of random weights $\bar{w}$, learning rate $l_r$, $v = 0$ and $b = 0.9$.
while $Loss > error$ do
    Randomly shuffle the training data of size $n$.
    for $i = 1, 2, \cdots, n$ do
        $v_{current} = bv_{previous} + (1-b)[\frac{\partial Loss}{\partial \bar{w}_{current}}]^2$
        $\bar{w}_{new} = \bar{w}_{current} - \frac{l_r}{\sqrt{v_{current}+\epsilon}} \frac{\partial Loss}{\partial \bar{w}_{current}}$
    end for
end while
```

where ϵ is a small number to prevent division by zero.

A further improved version was proposed by (Kingma et al., 2015), called the Adaptive Moment Estimation (Adam). In RMSProp, the algorithm adapts the learning rate based on the first moment only. Adam improves this further by using the average of the second moments of the gradients to make adaptation in the learning rate. The advantage of Adam is that it can handle sparse gradients on noisy data. The pseudocode of Adam is given below:

Algorithm 3 Adaptive Moment Estimation (Adam)

```
Initialise a vector of random weights $\bar{w}$, learning rate $l_r$, $v = 0$, $m = 0$, $b = 0.9$ and $b_1 = 0.999$.
while $Loss > error$ do
    Randomly shuffle the training data of size $n$.
    for $i = 1, 2, \cdots, n$ do
        $m_{current} = bm_{previous} + (1-b)[\frac{\partial Loss}{\partial \bar{w}_{current}}]^2$
        $v_{current} = b_1 v_{previous} + (1-b_1)[\frac{\partial Loss}{\partial \bar{w}_{current}}]^2$
        $\hat{m}_{current} = \frac{m_{current}}{1-b_{current}}$
        $\hat{v}_{current} = \frac{v_{current}}{1-b_{1,current}}$
        $\bar{w}_{new} = \bar{w}_{current} - \frac{l_r}{\sqrt{\hat{v}_{current}+\epsilon}} \hat{m}_{current} \frac{\partial Loss}{\partial \bar{w}_{current}}$
    end for
end while
```

where b is the exponential decay rate for the first moment and b_1 is the exponential decay rate for the second moment.

4.1.7 Epochs, Early Stopping and Batch Size

Epoch is the number of times that the entire training data set is passed through the ANN. Generally, we want the number of epoch as high as possible. However, this may cause the ANN to over-fit. **Early stopping** would be useful here to stop the training before the ANN becomes over-fit. It works by stopping the training process when loss function shows no improvement. **Batch size** is the number of input samples processed before the hyperparameters are updated.

4.2 Feedforward Neural Networks

Feedforward Neural Networks are the most common type of ANN in practical applications. They are called feed forward because the training data only flows from input layer to output layer, there is no feedback loop within the network.

4.2.1 Deep Neural Networks

Deep Neural Networks (DNN) are Feedforward Neural Networks with more than one hidden layer. The depth here refers to the number of hidden layers. This the type of ANN of our interest and the following theorems provide justifications.

Theorem 4.2.1 *(Hornik et al., 1989). Universal Approximation Theorem: Let $\mathcal{NN}^a_{d_{in},d_{out}}$ be the set of neural networks with activation function $f_a : \mathbb{R} \to \mathbb{R}$, input dimension $d_{in} \in \mathbb{N}$ and output dimension $d_{out} \in \mathbb{N}$. Then, if f_a is continuous and non-constant $\mathcal{NN}^a_{d_{in},d_{out}}$ is dense in Lebesgue space $L^p(\mu)$ for all finite measures μ.*

Theorem 4.2.2 *(Hornik et al., 1990). Universal Approximation Theorem for Derivatives: Let the function to be approximated $F^* \in C^n$, the mapping of neural network $F : \mathbb{R}^{d_0} \to \mathbb{R}$ and $\mathcal{NN}^a_{d_{in},d_{out}}$ be the set of single-layer neural networks with activation function $f_a : \mathbb{R} \to \mathbb{R}$, input dimension $d_{in} \in \mathbb{N}$ and output dimension 1. Then, if the activation function (non-constant) is $f_a \in C^n(\mathbb{R})$, then $\mathcal{NN}^a_{d_{in},1}$ arbitrarily approximates F^* and all its derivatives up to order n.*

Theorem 4.2.3 *(Eldan et al., 2016). The Power of depth of Neural Networks: There exists a simple function on R^d, expressible by a small 3-layer feedforward neural networks, which cannot be approximated by any 2-layer network, to more than a certain constant accuracy, unless its width is exponential in the dimension.*

According to Theorem 4.2.2, it is crucial to have an activation function that is n-differentiable if the approximation of the nth-derivative of the function F^* is required. Theorem 4.2.3 motivates the choice of deep network. It states that the depth of network can improve the predictive capabilities of network. However, it is worth noting that network deeper than 4 hidden layers does not provide much performance gain (Ioffe et al., 2015).

4.3 Common Pitfalls and Remedies

Here we discuss some common pitfalls in ANN training and the corresponding remedies.

4.3.1 Loss Function Stagnation

During training, the loss function may display infinitesimal improvements after each epoch.

To resolve this problem:

- **Increase the batch size** could potentially fix this issue as this helps the ANN model to learn better the relationships between data samples and data labels before updating the parameters.
- **Increase the learning rate**. Too small of learning rate can cause the ANN model to stuck in local minima.
- **Use a deeper network**. According to the power of depth theorem, deeper network tends to perform better than shallower ones. However, if the NN model already have 4 hidden layers then this methods may not be suitable as there is not much performance gain (see Chapter 4.2.1).
- **Use a different activation function** for the output layer. Make sure the range of the output activation function is appropriate for the problem.

4.3.2 Loss Function Fluctuation

During training, the loss function may display infinitesimal improvements after each epoch. Potential fix includes:

- **Increase the batch size**. The ANN model might not be able to interpret the true relationships between input and output data before each update when the batch size is too small.
- **Lower the initialised learning rate**.

4.3.3 Over-fitting

Over-fitting occurs when the ANN model is able to perform well on training data but fail to generalise on unseen test data.

Potential fix includes:

- **Reduce the complexity of ANN model**. Limit the number of hidden layers to 4, as (Eldan et al., 2016) shows not much advantage can be achieved beyond 4 hidden layers. Furthermore, experiment with narrower (less neurons) hidden layer.
- **Introduce early stopping**. Reduce the number of epochs if more epoch shows only little improvements. It is important to note that
- **Regularisation**. This can reduce the complexity of loss function and speed up convergence.

- **K-fold Cross-validation**. It works by splitting the training data into K equal-size portions. The K-1 portions are used for training and 1 portion is used for validation. This is repeated with different portions for validation K times.

4.3.4 Under-fitting

Under-fitting occurs when the ANN model fails to make sense of the relationships between data samples and labels.

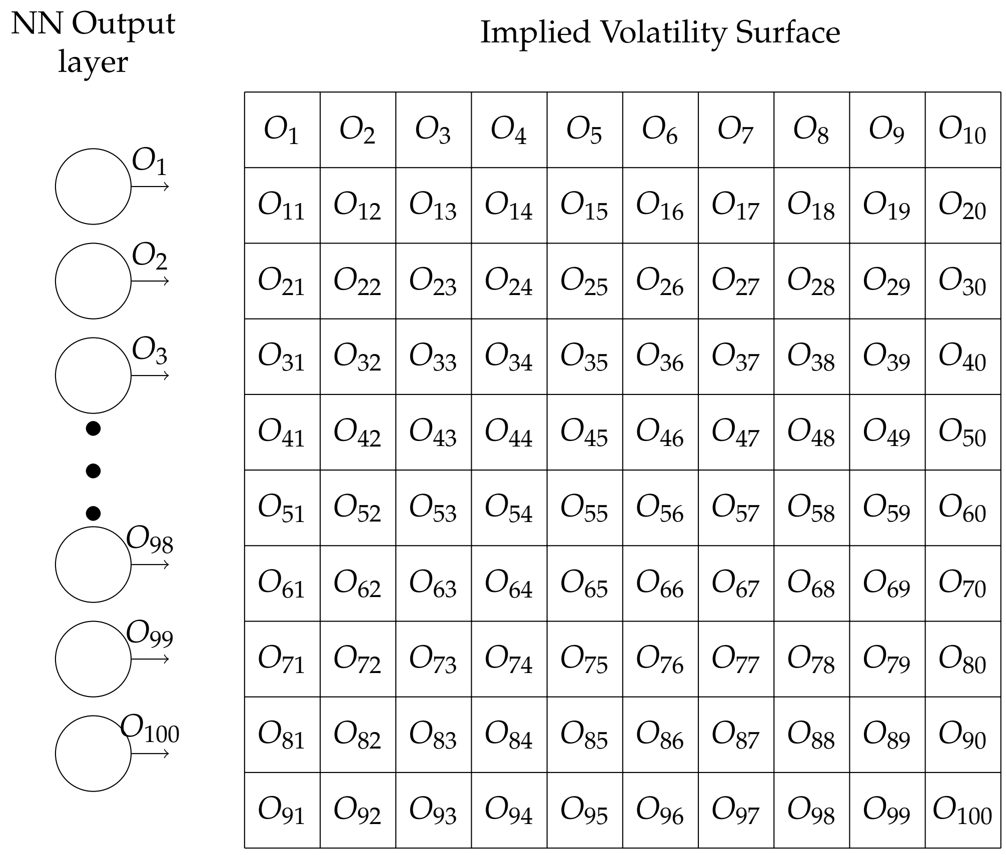
Potential fix includes:

- **Increase size of training data**. Complex functional mappings require the ANN model the train on more data before it can learn the relationships well.
- **Increase the depth of model**. It is very likely that the model is not deep enough for the problem. As a rule of thumb, limit the number of hidden layers to 3-4 for complex problems.

4.4 Application in Finance: ANN Image-based Implicit Method for Implied Volatility Approximation

For ANN's application in implied volatility predictions, we apply the image-based implicit method proposed by (Horvath et al., 2019). It is allegedly a powerful and robust method for approximating implied volatility. Rather than using one implied volatility value as network output, the image-based implicit method uses the entire implied volatility surface as the output. The implied volatility surface is represented by 10×10 "pixels", with each pixel corresponds to one maturity and one strike price. The input data corresponding to one implied volatility surface output is all the model parameters except for strike price and maturity, they are implicitly being learned by the network. This method offers the following benefits:

- More information is learned by the network with less input data. The information of neighbouring volatility points is incorporated in the training process.
- The network is learning the mapping of model parameters and the implied volatility surface directly. Volatility smile and term structure are available simultaneously and so it is more efficient than having only one single implied volatility value as output.
- The neighbouring volatility points can be numerically approximated almost instantly if intended.



O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	O_9	O_{10}
O_{11}	O_{12}	O_{13}	O_{14}	O_{15}	O_{16}	O_{17}	O_{18}	O_{19}	O_{20}
O_{21}	O_{22}	O_{23}	O_{24}	O_{25}	O_{26}	O_{27}	O_{28}	O_{29}	O_{30}
O_{31}	O_{32}	O_{33}	O_{34}	O_{35}	O_{36}	O_{37}	O_{38}	O_{39}	O_{40}
O_{41}	O_{42}	O_{43}	O_{44}	O_{45}	O_{46}	O_{47}	O_{48}	O_{49}	O_{50}
O_{51}	O_{52}	O_{53}	O_{54}	O_{55}	O_{56}	O_{57}	O_{58}	O_{59}	O_{60}
O_{61}	O_{62}	O_{63}	O_{64}	O_{65}	O_{66}	O_{67}	O_{68}	O_{69}	O_{70}
O_{71}	O_{72}	O_{73}	O_{74}	O_{75}	O_{76}	O_{77}	O_{78}	O_{79}	O_{80}
O_{81}	O_{82}	O_{83}	O_{84}	O_{85}	O_{86}	O_{87}	O_{88}	O_{89}	O_{90}
O_{91}	O_{92}	O_{93}	O_{94}	O_{95}	O_{96}	O_{97}	O_{98}	O_{99}	O_{100}

FIGURE 4.2: The "pixels" of implied volatility surface are represented by the outputs of neural network.

Chapter 5

Data

In this chapter we discuss about the data for our applications. We explain how the data is generated and some essential data pre-processing procedures.

Simulated data is chosen to train the ANN model mainly due to regulatory purposes. One might argue that training ANN with market data directly would yield better predictions. However, the lack of evidence for the stability and robustness of this method is yet to be found. Furthermore, there is no straightforward interpretation between inputs and outputs of ANN model if it is trained using this method. The main advantage of training ANN with simulated data is that the functional relationship is known and so the model is tractable. This property is important as tractable models tend to be more transparent and is required by regulations.

For neural network computation, Python is the logical language choice as it has many powerful machine learning libraries, such as *Keras*. To obtain good results, we require $5,000,000$ rows of data for the ANN experiment. Hence, we choose C++ instead of Python for the data generation process as it is computationally faster. We also use the lightweight header-only Python library, *pybind11* that can harmoniously integrate C++ and Python in one project (see Appendix A for tutorial on *pybind11*).

5.1 Data Generation and Storage

The data generation process is done entirely in C++ mainly because of its execution speed. Initially, we consider applying GPU parallel computing to speed up the process even further. After consulting with the supervisor, we decided to use the "std::future" class template instead. The "std::future" belongs to the standard C++11 library and it is very simple to implement compared to parallel computing. It allows the C++ program to run multiple functions asynchronously on different threads (see page 942 in (Duffy, 2018) for example). Our machine contains 2 physical cores, with 2 threads per physical core. This allows us to run 4 operations asynchronously without overhead costs and therefore in theory, improves the data generation speed by four-fold. Note that asynchronous programming is not meant to replace parallel computing completely. In cases where execution speed is absolutely critical, parallel computing should be used. Without asynchronous programming, the data generation process for 5,000,000 rows of Heston option price data would take more than 8 hours. Now, it only takes less than 2 hours to complete the process.

We follow the steps outline in Appendix A to wrap the C++ program using the *pybind11* Python library. The wrapped C++ function can be called in Python just like any other Python libraries. The *pybind11* library is very similar to the *Boost* library conceptually. The reason we choose *pybind11* is that it is a lightweight header-only library that is easy to use. This enables us to enjoy both the execution speed of C++ and the access to machine learning libraries of Python at the same time.

After the data is generated, it is stored in MySQL so that we do not need to re-generate the data each time we restart the computer or when the Python IDE crashes unexpectedly. We use the *mysql connector* Python library to communicate with MySQL on Python.

5.1.1 Heston Call Option Data

We follow the method and theory described in Chapter 3 to compute the call option price under Heston model. We only consider call option in this project. To train ANN that can also predict put option price, one can simply add an extra input feature that only takes two values: "1′ or "-1" with "1" represents call option and "-1" represents put option.

For this application we generated 5,000,000 rows of data that is uniformly distributed over a specified range. In this context, one row of data contains all the input features and the corresponding output. We use 10 model parameters as the input features. They include spot price (S), strike price (K), risk-free rate (r), dividend yield (D), current/instantaneous volatility (ν), maturity (T), long term volatility (θ), mean-reversion rate (κ), volatility of volatility (ξ) and correlation (ρ). In this case there will only be one output that is the call option price. See table 5.1 for their respective range.

TABLE 5.1: Heston Model Call Option Data

	Model Parameters	Range
Input	Spot Price (S)	$\in [20, 80]$
	Strike Price (K)	$\in [40, 55]$
	Risk-free rate (r)	$\in [0.03, 0.045]$
	Dividend Yield (D)	$\in [0, 0.1]$
	Instantaneous Volatility (ν)	$\in [0.2, 0.6]$
	Maturity (T)	$\in [0.03, 2.0]$
	Long Term Volatility (θ)	$\in [0.2, 1.0]$
	Mean Reversion Rate (κ)	$\in [0.1, 1.0]$
	Volatility of Volatility (ξ)	$\in [0.2, 0.5]$
	Correlation (ρ)	$\in [-0.95, 0.95]$
Output	Call Option Price (C)	$\in \mathbb{R}_{>0}$

5.1.2 Heston Implied Volatility Data

We generate the Heston implied volatility data with the theory explained in Chapter 3. For ANN application in implied volatility approximation, we decided to use the image-based implicit method in which the ANN output layer dimension is 100 with each output represents one pixel on the implied volatility surface.

Using this method, we need 100 times more data in order to have the same number of distinct rows of data as the normal, point-wise method (e.g Heston Call Option Pricing) does. However, due to hardware constraints, we decided to generate 10,000,000 rows of uniformly distributed data. That translates to only 100,000 distinct rows of data.

We use 6 model parameters as the input features. They include spot price (S), current/instantaneous volatility (ν), long term volatility (θ), mean-reversion rate (κ), volatility of volatility (ξ) and correlation (ρ).

Note that strike price and maturity are not being used as the input features as mentioned in Chapter 4.4. This is because the shape of one full implied volatility surface is dependent on the 6 model parameters mentioned above. The strike price and maturity correspond to one specific point on the full implied volatility surface. Nevertheless, we still require these two parameters to generate implied volatility data. We use the following strike price and maturity values:

- strike price $= \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4\}$
- maturity $= \{0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$

In this case there will be $10 \times 10 = 100$ outputs that is the implied volatility surface. See table 5.2 for their respective range.

TABLE 5.2: Heston Model Implied Volatility Data

	Model Parameters	Range
Input	Spot Price (S)	$\in [0.2, 2.5]$
	Instantaneous Volatility (ν)	$\in [0.2, 0.6]$
	Long Term Volatility (θ)	$\in [0.2, 1.0]$
	Mean Reversion Rate (κ)	$\in [0.1, 1.0]$
	Volatility of Volatility (ξ)	$\in [0.2, 0.5]$
	Correlation (ρ)	$\in [-0.95, 0.95]$
Output	Implied Volatility Surface (σ_{imp})	$\in \mathbb{R}^{10\times 10}_{>0}$

5.1.3 Rough Heston Call Option Data

Again we follow the theory in Chapter 3 and implement it in C++ to generate call option price under rough Heston model.

Similarly, we only consider call option here and generate 5,000,000 rows of data. We use 9 model parameters as the input features. They include strike price (K), risk-free rate (r), current/instantaneous volatility (ν), maturity (T), long term volatility (θ), mean-reversion rate (κ), volatility of volatility (ξ), correlation (ρ) and Hurst parameter (H). In this case there will only be one output that is the call option price. See table 5.3 for their respective range.

TABLE 5.3: Rough Heston Model Call Option Data

	Model Parameters	Range
Input	Strike Price (K)	$\in [0.5, 5]$
	Risk-free rate (r)	$\in [0.2, 0.5]$
	Instantaneous Volatility (ν)	$\in [0., 1.0]$
	Maturity (T)	$\in [0.1, 3.0]$
	Long Term Volatility (θ)	$\in [0.01, 0.1]$
	Mean Reversion Rate (κ)	$\in [1.0, 4.0]$
	Volatility of Volatility (ξ)	$\in [0.01, 0.5]$
	Correlation (ρ)	$\in [-0.95, 0.95]$
	Hurst Parameter (H)	$\in [0.05, 0.1]$
Output	Call Option Price (C)	$\in \mathbb{R}_{>0}$

5.1.4 Rough Heston Implied Volatility Data

Finally, We generate the rough Heston implied volatility data with the theory explained in Chapter 3. As before, we also use the image-based implicit method.

The data size we use is 10,000,000 rows of data and that translates to 100,000 distinct rows of data. We use 7 model parameters as the input features. They include spot price (S), current/instantaneous volatility (ν), long term volatility (θ), mean-reversion rate (κ), volatility of volatility (ξ), correlation (ρ) and Hurst parameter (H). We use the following strike price and maturity values:

- strike price $= \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4\}$
- maturity $= \{0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$

As before, there will be $10 \times 10 = 100$ outputs that is the rough Heston implied volatility surface. See table 5.4 for their respective range.

TABLE 5.4: Heston Model Implied Volatility Data

	Model Parameters	Range
Input	Spot Price (S)	$\in [0.2, 2.5]$
	Instantaneous Volatility (ν)	$\in [0.2, 0.6]$
	Long Term Volatility (θ)	$\in [0.2, 1.0]$
	Mean Reversion Rate (κ)	$\in [0.1, 1.0]$
	Volatility of Volatility (ξ)	$\in [0.2, 0.5]$
	Correlation (ρ)	$\in [-0.95, 0.95]$
	Hurst Parameter (H)	$\in [0.05, 0.1]$
Output	Implied Volatility Surface (σ_{imp})	$\in \mathbb{R}^{10\times 10}_{>0}$

5.2 Data Pre-processing

In this section we discuss about some necessary data pre-processing procedures. We talk about data splitting for validation and evaluation (testing), data standardisation and normalisation, and data partitioning for K-fold cross validation.

5.2.1 Data Splitting: Train - Validate - Test

Generally, not 100% of available data is used for training. We would reserve some of the data for validation and for performance evaluation.

We split the data into three sets: 70% (train), 15% (validate) and 15% (test). The first 70% of data is used for training whilst 15% of the data is used for validation during training. The last 15% is for evaluating the trained ANN's predictive accuracy.

The validation data set is the data set used for tuning the hyperparameters of the ANN. This is important to help prevent over-fitting of the network. The test data set is not used during training. It is the unseen (out-of-sample) data that is used for evaluating the trained ANN's predictions.

5.2.2 Data Standardisation and Normalisation

Before the data is fed into the ANN for training, it is absolutely essential to scale the data to speed up convergence and improve performance.

The magnitude of the each data feature is different. It is essential to rescale the data so that the ANN model will not misinterpret their relative importance based on their magnitude. Typical scaling techniques include standardisation and normalisation.

Data standardisation is defined as

$$x_{scaled} = \frac{x - x_{mean}}{x_{std}} \tag{5.1}$$

where

- x is data
- x_{mean} is the mean of that data feature
- x_{std} is the standard deviation of that data feature

(Horvath et al., 2019) claims that this works especially well for quantity that is positively unbounded such as implied volatility.

For other model parameters, we apply data normalisation, which is defined as

$$x_{scaled} = \frac{2x - (x_{max} + x_{min})}{x_{max} + x_{min}} \in [-1, 1] \tag{5.2}$$

where

- x is data

- x_{max} is the maximum value of that data feature
- x_{min} is the minimum value of that data feature

After data scaling (standardisation or normalisation), ANN is able to learn the true relationships between inputs and outputs without the bias from the difference in their magnitudes

5.2.3 Data Partitioning

In this section, we explain how the data is partitioned for K-fold cross validation.

First, we shuffle the entire training data set. Then, the entire data set is divided into K equal sized portions. For each portion, we use it as the test data set and we train the ANN on the other $K - 1$ portions. This is repeated K times, with each of the K portions used exactly once as the test data set. The average of K results is the performance measure of the model.

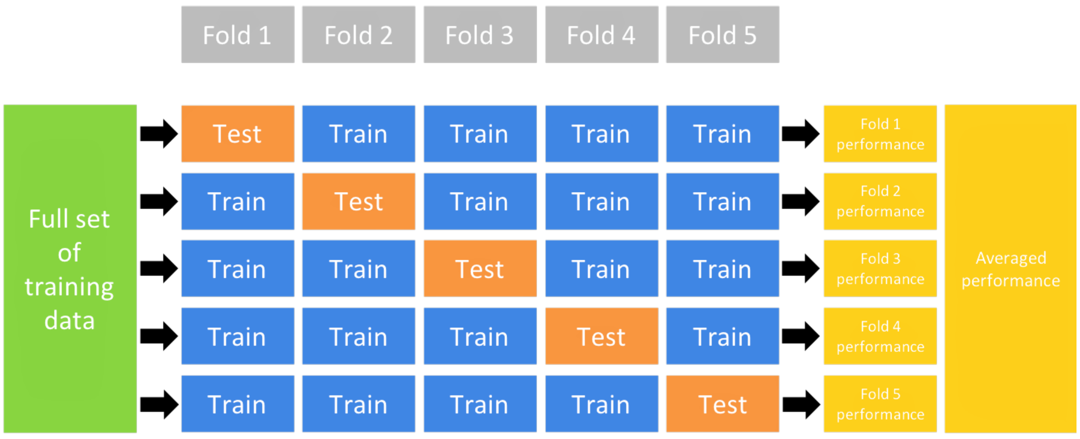


FIGURE 5.1: shows how the data is partitioned for 5-fold cross validation. Adapted from (*Chapter 2 Modeling Process: k-fold cross validation*).

Chapter 6

Neural Networks Architecture and Experimental Setup

We start this chapter by discussing the appropriate ANN architecture and experimental setup for our applications. We apply ANN to four different applications: option pricing under Heston model, Heston implied volatility surface approximation, option pricing under rough Heston model and rough Heston implied volatility surface approximation. We first apply ANN in Heston model as a concept validation and then proceed to rough Heston model.

Subsequently, we discuss about the relevant evaluation metrics and validation methods for our ANN models. Finally, we end this chapter with the implementation steps for ANN in *Keras*.

6.1 Neural Networks Architecture

Configuring ANN's architecture is a combination of art and science. It requires experience and systematic experiment to decide the right parameters and architecture for a specific problem. For a particular problem, there might be multiple designs that are appropriate. To begin with, we would use the network architecture used by (Horvath et al., 2019) and made adjustment to the hyperparameters if it performs poorly for our applications.

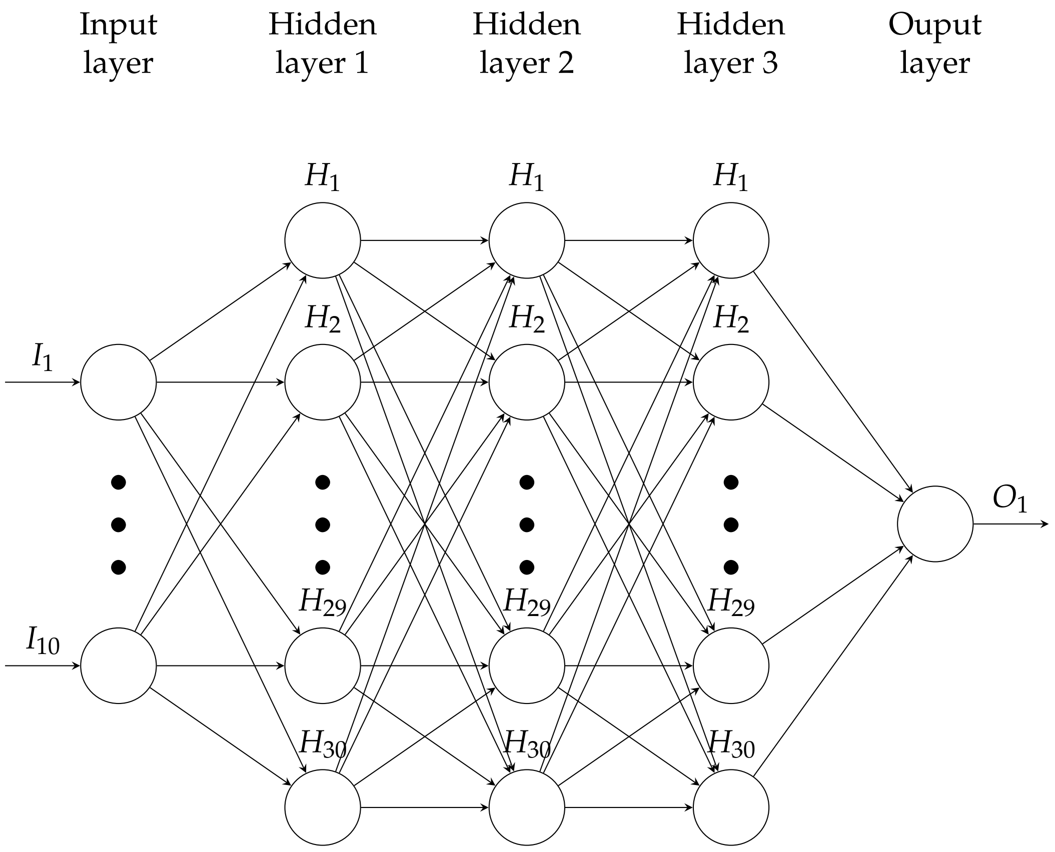


FIGURE 6.1: Typical ANN Architecture for Option Pricing Application

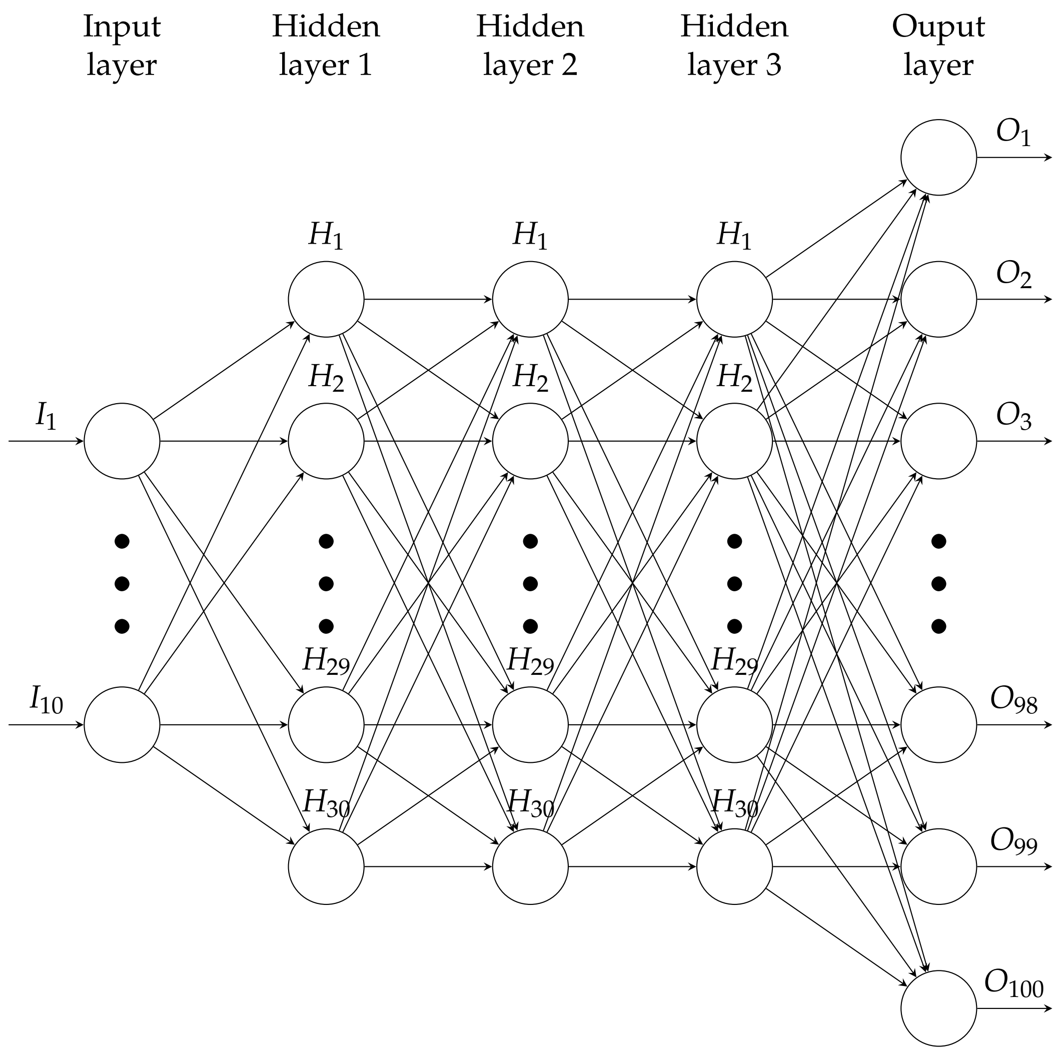


FIGURE 6.2: Typical ANN Architecture for Implied Volatility Surface Approximation

6.1.1 ANN Architecture: Option Pricing under Heston Model

ANN with 3 hidden layers is approapriate for this application. The number of neurons will be 50 with exponential linear unit (ELU) as the activation function for each hidden layer.

Initially, the rectified linear activation function (ReLU) was considered. It is the choice for many applications as it can overcome the vanishing gradient problem whilst being less computationally expensive compared to other activation functions. However it has some major drawbacks:

- The nth-derivative of ReLU is not continuous for all $n > 0$.
- The ReLU cannot produce negative outputs.
- For activation $x < 0$, ReLU has zero gradient.

From Theorem 4.2.2, we know choosing a n-differentiable activation function is necessary for computing the nth-derivatives of the functional mapping. This is crucial for our application because we would be interested in option Greeks computation using the same ANN if it shows promising results for option pricing. For risk management purposes, selecting an activation function that option Greeks can be calculated is crucial.

The obvious alternative would be the ELU (defined in Chapter 4.1.4). For $x < 0$, it has a smooth output until it approaches to $-\alpha$. When $x > 0$, the output of ELU is positively unbounded, which is suitable for our application as the values of option price and implied volatility are also in theory, positively unbounded.

Since our problem is a regression problem, linear activation function (defined in Chapter 4.1.5) would be appropriate for the output layer as its output is unbounded. We discover that batch size of 200 and number of epochs of 30 would yield good accuracy.

To summarise, the **ANN model for Heston Option Pricing** is,

- **Input Layer**: 10 neurons
- **Hidden Layer 1**: 50 neurons, ELU as activation function
- **Hidden Layer 2**: 50 neurons, ELU as activation function
- **Hidden Layer 3**: 50 neurons, ELU as activation function
- **Output Layer**: 1 neuron, linear function as activation function

6.1.2 ANN Architecture: Implied Volatility Surface of Heston Model

For approximating the full implied volatility surface of Heston model we apply the image-based implicit method (see Chapter 4.4). This implies the output dimension is 100.

We use 3 hidden layers with 80 neurons. The activation function for hidden layers is exponential linear unit (ELU). As before we use linear activation function for the output layer. We discover that batch size of 20 and 200 epochs would yield good accuracy. Since the number of epochs is relatively high, we introduce an early stopping feature to stop the training when validation loss stops improving for 25 epochs.

To summarise, the **ANN model for Heston implied volatility surface approximation** is,

- **Input Layer**: 6 neurons
- **Hidden Layer 1**: 80 neurons, ELU as activation function
- **Hidden Layer 2**: 80 neurons, ELU as activation function
- **Hidden Layer 3**: 80 neurons, ELU as activation function
- **Hidden Layer 4**: 80 neurons, ELU as activation function
- **Output Layer**: 100 neurons, linear function as activation function

6.1.3 ANN Architecture: Option Pricing under Rough Heston Model

For option pricing under rough Heston model, the **ANN model for rough Heston Option Pricing** is,

- **Input Layer**: 9 neurons
- **Hidden Layer 1**: 50 neurons, ELU as activation function
- **Hidden Layer 2**: 50 neurons, ELU as activation function
- **Hidden Layer 3**: 50 neurons, ELU as activation function
- **Output Layer**: 1 neuron, linear function as activation function

As before, we use batch size of 200 and 30 epochs.

6.1.4 ANN Architecture: Implied Volatility Surface of Rough Heston Model

The **ANN model for rough Heston implied volatility surface approximation** is,

- **Input Layer**: 7 neurons
- **Hidden Layer 1**: 80 neurons, ELU as activation function
- **Hidden Layer 2**: 80 neurons, ELU as activation function
- **Hidden Layer 3**: 80 neurons, ELU as activation function
- **Output Layer**: 100 neuron, linear function as activation function

We use batch size of 20 and number of epochs of 200 with early stopping feature to stop the training when validation loss stops improving for 25 epochs.

6.2 Performance Metrics and Validation Methods

We require some metrics to evaluate the performance of the ANN in terms of accuracy and speed.

For accuracy, we have introduced the MSE and MAE in Chapter 4.1.4. It is common to use the coefficient of determination (R^2) to quantify the performance of a regression model. It is a statistical measure that quantify the proportion of the variances for dependent variables that is explained by independent variables. The formula is

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_{i,actual} - y_{i,predict})^2}{\sum_{i=1}^{n}(y_{i,actual} - \bar{y}_{actual})^2} \tag{6.1}$$

We also suggest a user-defined accuracy metrics: Maximum Absolute Error. It is defined as

$$Max\ Abs.\ Error = max(|y_{i,actual} - y_{i,predict}|) \tag{6.2}$$

This metric provides a measure of how large the error could be in the worst case scenario. It is especially important from a risk management perspective.

As for the run-time performance of ANN. We simply use the Python built-in timer function to quantify this property.

In Chapter 5.2.3 we explained about the data partitioning process. The motivation for partitioning data is to do K-fold cross validation. Cross validation is a statistical technique to assess how well the predictive model can be generalised to an independent data set. It is a good tool for identifying over-fitting.

As explained earlier, the process involves training and testing different permutation of the partitioned data sets multiple times. The indication of a well generalised model is that the error magnitudes in each round are consistent with each other and their average. We select $K = 5$ for our applications.

6.3 Implementation of ANN in *Keras*

In this section we outline the standard ANN computation process using *Keras*.

1. Construct the ANN model by specifying the number of input neurons, number of hidden neurons, number of hidden layers, number of output neurons and the activation functions for each layer. Print model summary to check for errors.

2. Configure the settings of the model by using the "compile" function. We select the MSE as the loss function, and we specify MAE, R^2 and our user-defined maximum absolute error as the metrics. For all of our applications we choose ADAM as the optimisation algorithm. To prevent over-fitting, we use the early stopping function, especially when the number of epochs is high. It is important to note that the validation loss (not training loss) should be used as the early stopping criteria. This means the training will stop early if there is no reduction in validation loss.

3. Starts training the ANN model. It is important to specify the "validation split" argument here for splitting the training data into a train set and validation set. Note that the test set should not be used as validation set, it should be reserved for evaluating ANN's predictions after training.

4. When the training is complete, generate the plot of MSE (loss function) versus number of epochs to visualise the training process. MSE should decline gradually and then level off as number of epochs increases. See Chapter 4.3 for potential training issues and remedies.

5. Apply 5-fold cross validation to check for over-fitting. Evaluate the trained model using "evaluate" function.

6. Generate predictions. Visualise the predictions by plotting predicted values versus actual values on the same graph.

See Appendix C for Python code example.

6.4 Summary of Neural Networks Architecture

TABLE 6.1: Summary of Neural Networks Architecture for Each Application

ANN Models	**Applications**	**Input Layer**	**Hidden Layers**	**Output Layer**	**Batch Size**	**Epoch**
Heston Option Pricing ANN	Heston Call Option Pricing	10 neurons	3 layers $\times$ 50 neurons	1 neuron	200	30
Heston Implied Volatility ANN	Heston Implied Volatility Surface	6 neurons	3 layers $\times$ 80 neurons	100 neurons	20	200
Rough Heston Option Pricing ANN	Rough Heston Call Option Pricing	9 neurons	3 layers $\times$ 50 neurons	1 neuron	200	30
Rough Heston Implied Volatility ANN	Rough Heston Implied Volatility Surface	7 neurons	3 layers $\times$ 80 neurons	100 neurons	20	200

Chapter 7

Results and Discussions

In this chapter we present our results and discuss our findings.

7.1 Heston Option Pricing ANN Results

Figure 7.1 shows the loss function plot of Heston Option Pricing ANN learning curves. The loss function used here is the MSE. As the ANN is trained, both the curves of training loss function and validation loss function decline until they converge to a point. It took less than 30 epochs for the ANN to reach a satisfactory level of accuracy. There is a small gap between the training loss function and validation loss function. This indicates the ANN model is well trained.

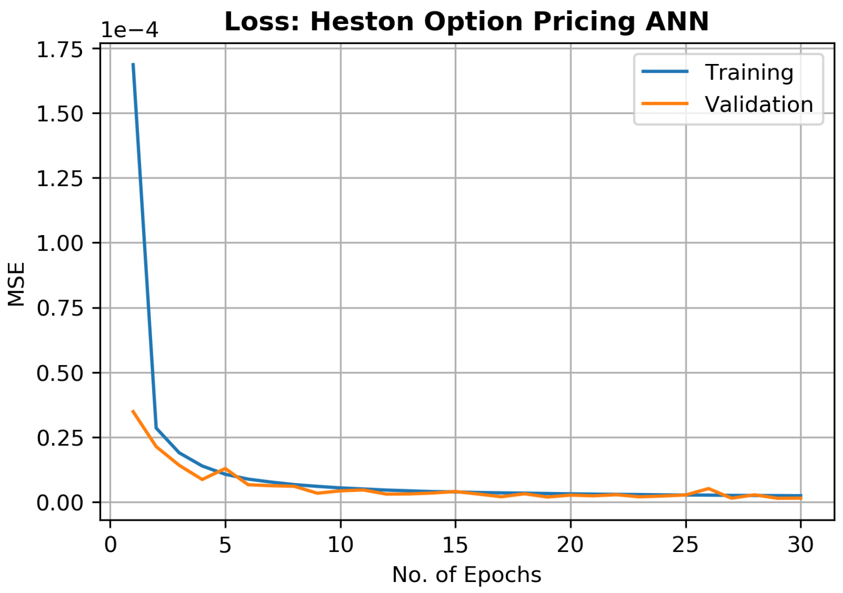


FIGURE 7.1: Plot of MSE Loss Function vs No. of Epochs for Heston Option Pricing ANN.

Table 7.1 summarises the error metrics of Heston Option Pricing ANN model's predictions on unseen test data. The MSE, MAE, Max Abs. Error and R^2 are 2.00×10^{-6}, 9.58×10^{-4}, 6.50×10^{-3} and 0.999 respectively. The low values in error metrics and high R^2 value imply the ANN model generalises well and is able to give high accuracy predictions for option price under Heston model. Figure 7.2 attests its high accuracy. Almost all of the predictions lie on the perfect predictions line (red line).

TABLE 7.1: Error Metrics of Heston Option Pricing ANN Predictions on Unseen Data

MSE	MAE	Max Abs. Error	R^2
2.00×10^{-6}	9.58×10^{-4}	6.50×10^{-3}	0.999

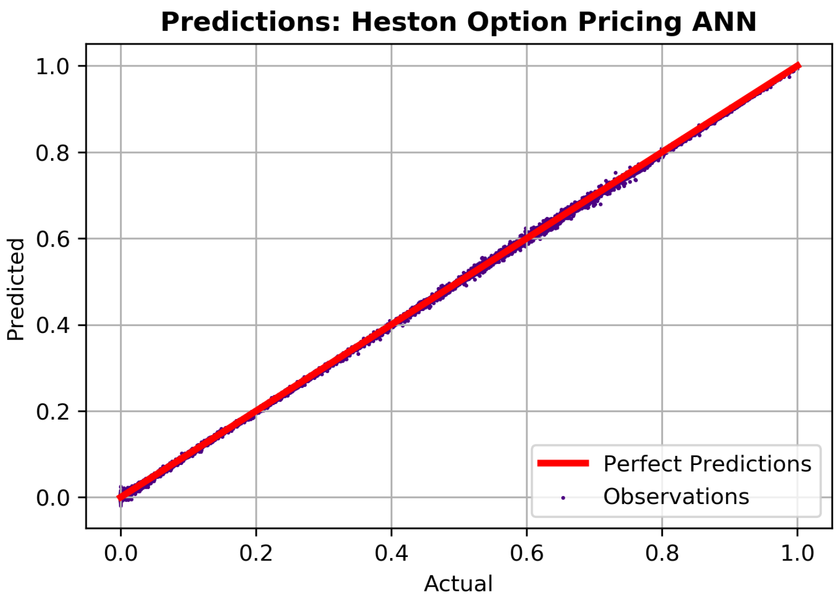


FIGURE 7.2: Plot of Predicted vs Actual values for Heston Option Pricing ANN.

TABLE 7.2: 5-fold Cross Validation Results of Heston Option Pricing ANN.

	MSE	MAE	Max Abs. Error
Fold 1	5.78×10^{-7}	5.45×10^{-4}	2.04×10^{-3}
Fold 2	9.90×10^{-7}	7.69×10^{-4}	2.40×10^{-3}
Fold 3	7.15×10^{-7}	6.21×10^{-4}	2.20×10^{-3}
Fold 4	1.24×10^{-6}	8.67×10^{-4}	2.53×10^{-3}
Fold 5	6.54×10^{-7}	5.79×10^{-4}	2.21×10^{-3}
Average	8.36×10^{-7}	6.76×10^{-4}	2.27×10^{-3}

Table 7.2 shows the results from 5-fold cross validation which again confirms the ANN model can give accurate predictions and generalises well to unseen data. The values of each fold are consistent with each other and with their average values.

7.2 Heston Implied Volatility ANN Results

Figure 7.3 shows the loss function plot of Heston Implied Volatility ANN learning curves. Again, we use the MSE as the loss function here. We can see that the validation loss curve experiences some fluctuations during training. We experimented with various ANN setups and this is the least fluctuating learning curve we can achieve. Despite the fluctuations, the overall value of validation loss function declines to a satisfactory low level and the gap between validation loss and training loss is negligible. Due to the comparatively more complex ANN architecture, it took about 200 epochs for the ANN to reach a satisfactory level of accuracy.

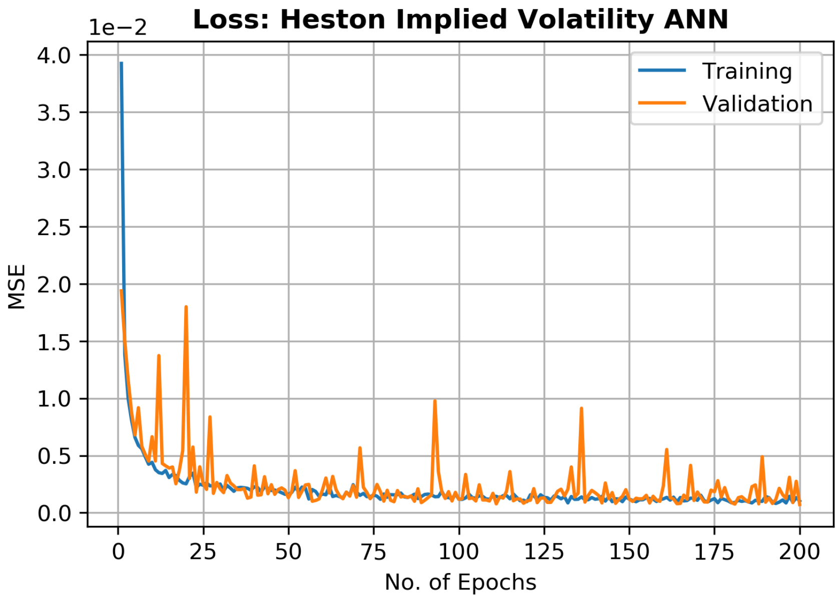


FIGURE 7.3: Plot of MSE Loss Function vs No. of Epochs for Heston Implied Volatility ANN.

Table 7.3 summarises the error metrics of Heston Implied Volatility ANN model's predictions on unseen test data. The MSE, MAE, Max Abs. Error and R^2 are 6.24×10^{-4}, 1.27×10^{-2}, 3.71×10^{-1} and 0.999 respectively. The low values in error metrics and high R^2 value imply the ANN model generalises well and is able to give accurate predictions for implied volatility surface under Heston model. Figure 7.4 shows the ANN predicted volatility smiles and the actual smiles for 10 different maturities. We can see that almost all of the predicted values are consistent with the actual values.

TABLE 7.3: Accuracy Metrics of Heston Implied Volatility ANN Predictions on Unseen Data

MSE	MAE	Max Abs. Error	R^2
6.24×10^{-4}	1.27×10^{-2}	3.71×10^{-1}	0.999

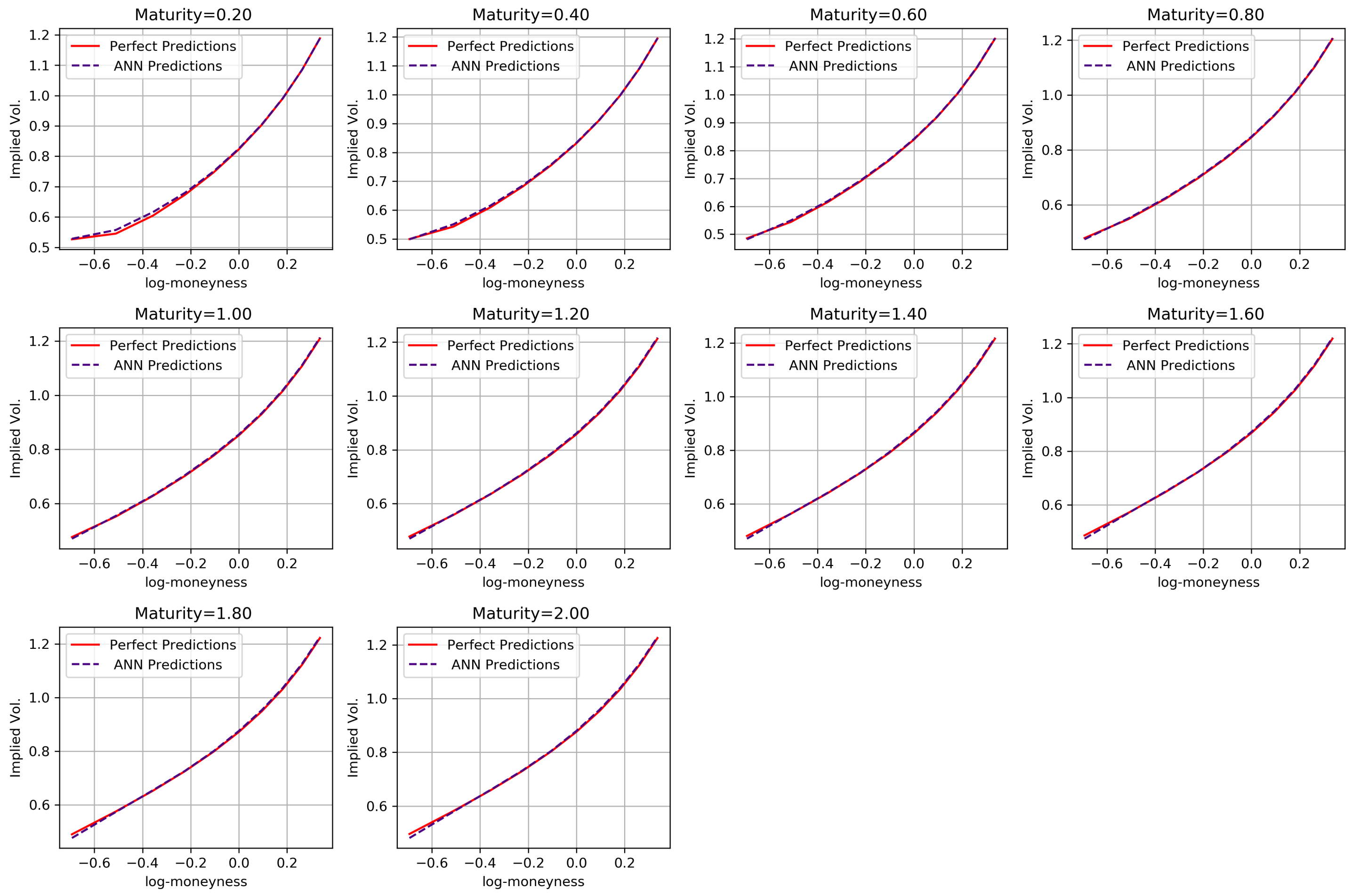


FIGURE 7.4: Heston Implied Volatility Smiles ANN Predictions vs Actual Values.

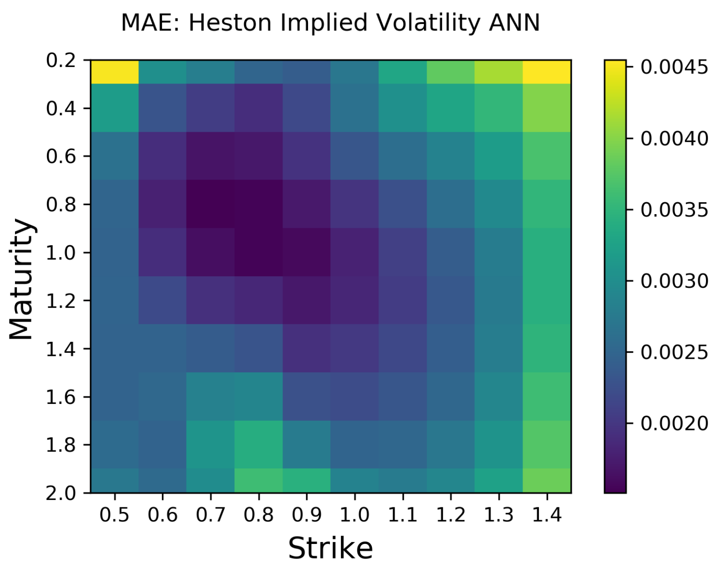


FIGURE 7.5: Mean Absolute Error of Full Heston Implied Volatility Surface ANN Predictions on Unseen Data.

Figure 7.5 shows the MAE values of Heston implied volatility ANN predictions across the entire implied volatility surface. The largest MAE value is 0.0045. A large area of the surface has MAE values lower than 0.0030. It is worth noting that the accuracy is relatively poor when the strike price is at the extremes and maturity is small. However, the ANN gives good predictions overall for the whole surface.

The 5-fold cross validation results for Heston implied volatility ANN model shows consistent values. The results is summarised in Table 7.4. The consistent values indicate the ANN model is well-trained and is able to generalise to unseen test data.

TABLE 7.4: 5-fold Cross Validation Results of Heston Implied Volatility ANN

	MSE	MAE	Max Abs. Error
Fold 1	8.15×10^{-4}	1.13×10^{-2}	3.88×10^{-1}
Fold 2	4.74×10^{-4}	1.08×10^{-2}	3.13×10^{-1}
Fold 3	1.37×10^{-3}	1.74×10^{-2}	4.46×10^{-1}
Fold 4	3.38×10^{-4}	8.61×10^{-3}	2.19×10^{-1}
Fold 5	4.60×10^{-4}	1.02×10^{-2}	2.67×10^{-1}
Average	6.92×10^{-4}	1.12×10^{-2}	3.27×10^{-1}

7.3 Rough Heston Option Pricing ANN Results

Figure 7.6 shows the loss function plot of Rough Heston Option Pricing ANN learning curves. As before, MSE is used as the loss function. As the ANN progresses, both the curves of training loss function and validation loss function decline until they converge to a point. Since this is a relatively simple model, it took less than 30 epochs of training to reach a good level of accuracy. The small discrepancy between the training loss and validation indicates the ANN model is well-trained.

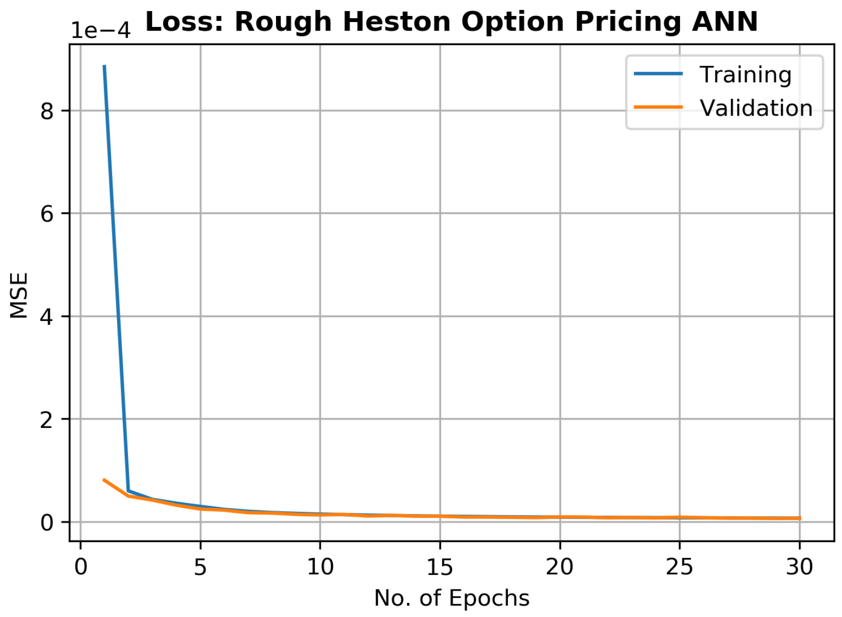


FIGURE 7.6: Plot of MSE Loss Function vs No. of Epochs for Rough Heston Option Pricing ANN.

Table 7.5 summarises the error metrics of Rough Heston Option Pricing ANN model's predictions on unseen test data. The MSE, MAE, Max Abs. Error and R^2 are 9.00×10^{-6}, 2.28×10^{-3}, 1.76×10^{-1} and 0.999 respectively. These metrics imply the ANN model generalises well and is able to give high accuracy predictions for option prices under rough Heston model. Figure 7.7 attests its high accuracy again. We can see that most of the predictions lie on the perfect predictions line (red line).

TABLE 7.5: Accuracy Metrics of Rough Heston Option Pricing ANN Predictions on Unseen Data.

MSE	MAE	Max Abs. Error	R^2
9.00×10^{-6}	2.28×10^{-3}	1.76×10^{-1}	0.999

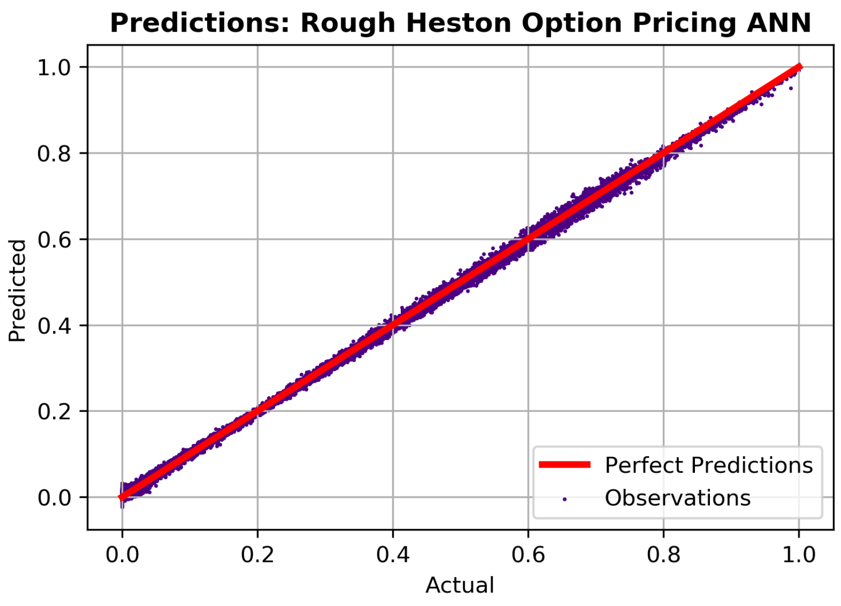


FIGURE 7.7: Plot of Predicted vs Actual values for Rough Heston Pricing ANN.

TABLE 7.6: 5-fold Cross Validation Results of Rough Heston Option Pricing ANN

	MSE	MAE	Max Abs. Error
Fold 1	3.79×10^{-6}	1.35×10^{-3}	5.99×10^{-3}
Fold 2	2.33×10^{-6}	9.96×10^{-4}	4.89×10^{-3}
Fold 3	2.06×10^{-6}	9.88×10^{-3}	4.41×10^{-3}
Fold 4	2.16×10^{-6}	1.08×10^{-3}	4.07×10^{-3}
Fold 5	1.84×10^{-6}	1.01×10^{-3}	3.74×10^{-3}
Average	2.44×10^{-6}	4.62×10^{-3}	1.08×10^{-3}

The highly consistent values from 5-fold cross validation again confirm the ANN model is able to generalise to unseen data.

7.4 Rough Heston Implied Volatility ANN Results

Figure 7.8 shows the loss function plot of Rough Heston Implied Volatility ANN learning curves. We can see that the validation loss curve experiences some fluctuations during the training. Again, we select the least fluctuating setup. Despite the fluctuations, the overall value of validation loss function declines to a satisfactory low level and the discrepancy between validation loss and training loss is negligible. Initially, the training epoch is 200 but it was stopped early at around 85 since the validation loss value has not improved for 25 epochs. It achieves a good accuracy nonetheless.

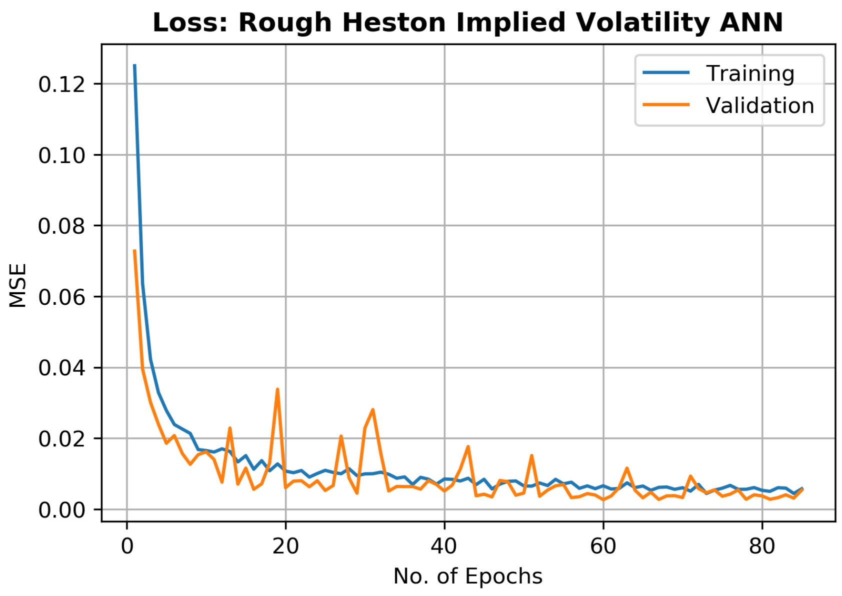


FIGURE 7.8: Plot of MSE Loss Function vs No. of Epochs for Rough Heston Implied Volatility ANN.

Table 7.7 summarises the error metrics of Rough Heston Implied Volatility ANN model's predictions on unseen test data. The MSE, MAE, Max Abs. Error and R^2 are 5.66×10^{-4}, 1.08×10^{-2}, 3.49×10^{-1} and 0.999 respectively. These metrics show the ANN model can produce accurate predictions on unseen data. This is again confirmed by Figure 7.9, which shows the ANN predicted volatility smiles and the actual smiles for 10 different maturities. We can see that almost all of the predicted values are consistent with the actual values.

TABLE 7.7: Accuracy Metrics of Rough Heston Implied Volatility ANN Predictions on Unseen Data.

MSE	MAE	Max Error	R^2
5.66×10^{-4}	1.08×10^{-2}	3.49×10^{-1}	0.999

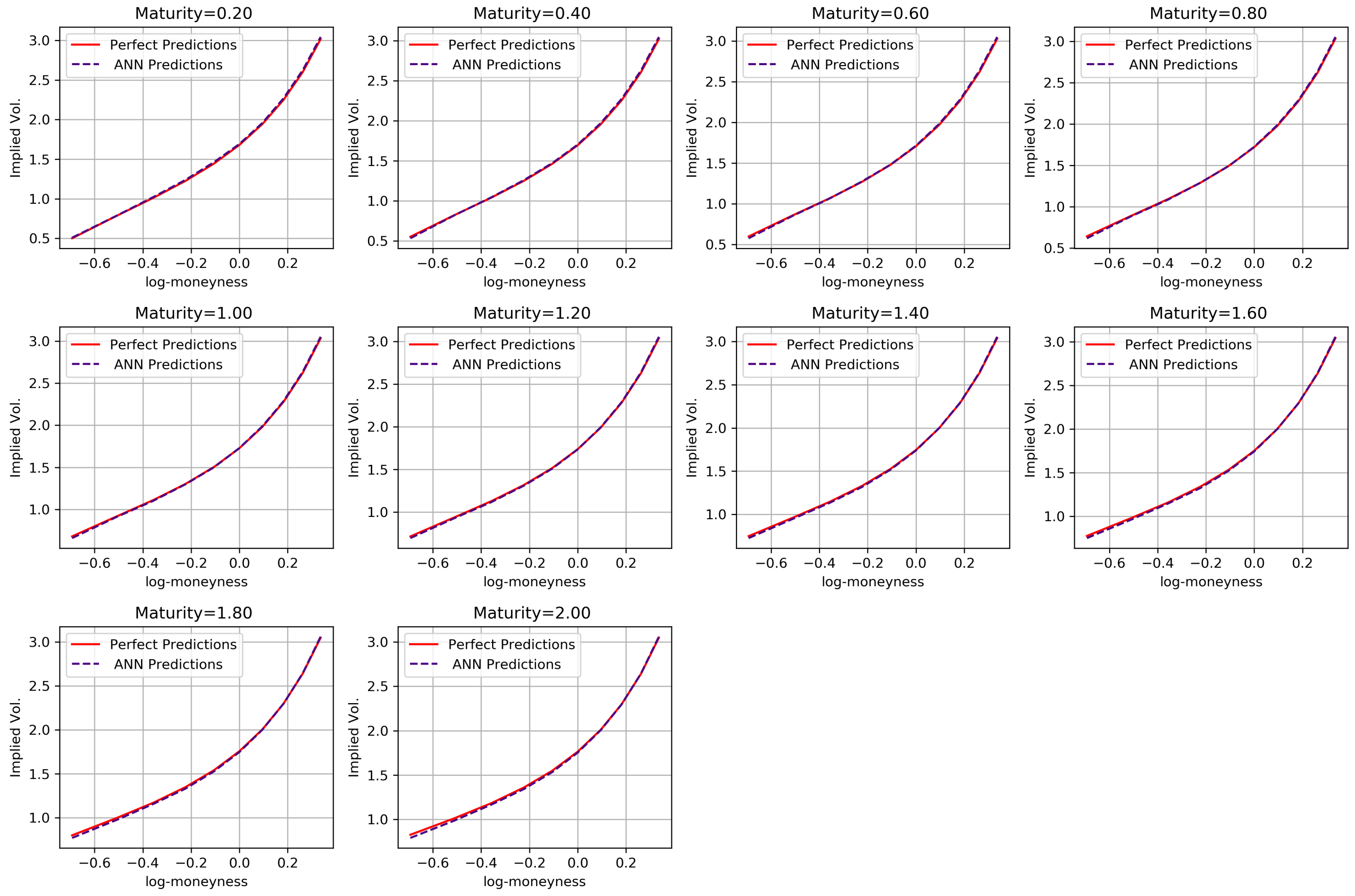


FIGURE 7.9: Rough Heston Implied Volatility Smiles ANN Predictions vs Actual Values.

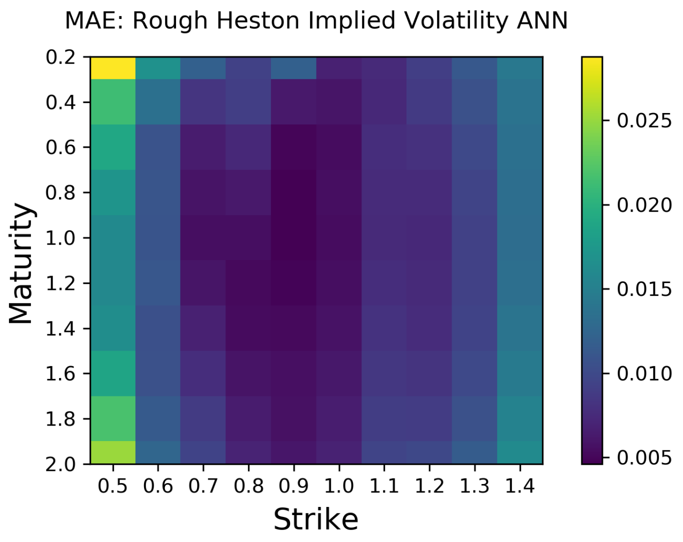


FIGURE 7.10: Mean Absolute Error of Full Rough Heston Implied Volatility Surface ANN Predictions on Unseen Data.

Figure 7.10 shows the MAE values of Rough Heston Implied Volatility ANN predictions across the entire implied volatility surface. The largest MAE value is below 0.0030. A large area of the surface has MAE values lower than 0.0015. It is worth noting that the accuracy is relatively poor when the strike price and maturity values are small. However, the ANN gives good predictions overall for the whole surface. In fact, it is more accurate than the Heston implied volatility ANN model despite having one extra input and similar architecture. We think this is entirely due to the stochastic nature of the optimiser.

The 5-fold cross validation results for rough Heston implied volatility ANN model shows consistent values just like the previous three cases. The results is summarised in Table 7.8. The consistency in values indicate the ANN model is well-trained and is able to generalise to unseen test data.

TABLE 7.8: 5-fold Cross Validation Results of Rough Heston Implied Volatility ANN.

	MSE	MAE	Max Error
Fold 1	8.15×10^{-4}	1.13×10^{-2}	3.88×10^{-1}
Fold 2	4.74×10^{-4}	1.08×10^{-2}	3.13×10^{-1}
Fold 3	1.37×10^{-3}	1.74×10^{-2}	4.46×10^{-1}
Fold 4	3.38×10^{-4}	8.61×10^{-3}	2.19×10^{-1}
Fold 5	4.60×10^{-4}	1.02×10^{-2}	2.67×10^{-1}
Average	6.92×10^{-4}	1.12×10^{-2}	3.27×10^{-1}

7.5 Run-time Performance

In this section we present the training time and the execution speed of ANN versus traditional methods. The speed tests are run on a machine with Intel i5-7200U chip (up to 3.10 GHz) and 8GB of RAM.

Table 7.9 summarises the training time required for each application. We can see that although the training time per epoch for option pricing applications is higher than that for implied volatility applications, the resulting total training time is similar for both types of applications. This is because number of epochs for training implied volatility ANN models is higher.

We also emphasise that the number of distinct rows of data available for implied volatility ANN training is less. Thus, the training time is similar to that of option pricing ANN training despite the more complex ANN architecture.

TABLE 7.9: Training Time

Applications	Training Time per Epoch	Total Training Time
Heston Option Pricing	30 *s*	900 *s*
Heston Implied Volatility	5 *s*	1000 *s*
rHeston Option Pricing	30 *s*	900 *s*
rHeston Implied Volatility	5 *s*	1000 *s*

TABLE 7.10: Execution Time for Predictions using ANN and Traditional Methods

Applications	Traditional Methods	ANN
Heston Option Pricing	8.84 *ms*	57.9 *ms*
Heston Implied Volatility	2.31 *ms*	43.7 *ms*
rHeston Option Pricing	6.52 *ms*	52.6 *ms*
rHeston Implied Volatility	2.87 *ms*	48.3 *ms*

Table 7.10 summarises the execution time for generating predictions using ANN and traditional methods. Before the experiment, we expect the ANN to generate predictions faster. However, results shows that ANN is actually up to 7 times slower than the traditional methods for option pricing (both Heston and rough Heston); up to 18 times slower than the traditional methods for implied volatility surface approximation (both Heston and rough Heston). This shows the traditional approximation methods are more efficient.

Chapter 8

Conclusions and Outlook

To summarise, results shows that ANN has the ability to approximate option price and implied volatility under Heston and rough Heston models to a high degree of accuracy. However, the efficiency of ANN is not as good as the traditional methods. Hence, we conclude that such an approach may therefore be considered an ineffective alternative. There are more efficient methods such as the numerical integration of FFT for option price computation. The traditional method we used for approximating implied volatility does not even require any numerical schemes and hence its efficiency.

As for the future projects, it would be interesting to investigate the proven rough Bergomi model applications with exotic options such as lookback and digital options. Moreover, it would also be worthwhile to apply gradient-free optimisers for ANN training. Some of the interesting examples include differential evolution, dual annealing and simplicial homology global optimisers.

Appendix A

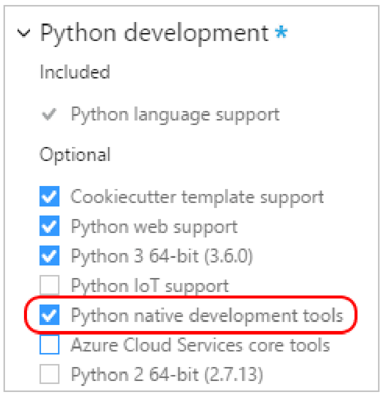
pybind11: A step-by-step tutorial with examples

This tutorial assumes basic knowledge of C++ and Python.

Sources: *Create a C++ extension for Python*, *Using the C++ eigen library to calculate matrix inverse and determinant*, *pybind11 Documentation*, *Eigen: The Matrix Class*

A.1 Software Prerequisites

- Visual Studio 2017 or later with both the **Desktop Development with C++** and **Python Development** workloads installed with default options.
- In the **Python Development** workload, also select the box on the right for **Python native development tools**. This option sets up most of the configuration required. (This option also includes the C++ workload automatically.)



Python development *

Included

✓ Python language support

Optional

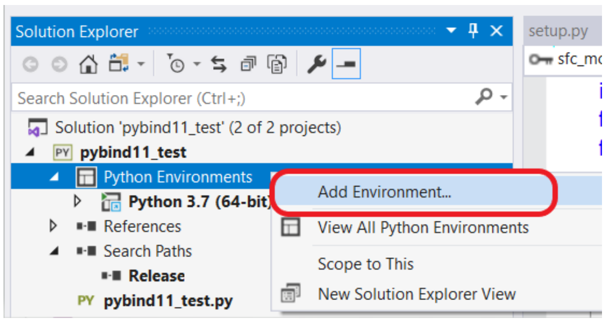
- [x] Cookiecutter template support
- [x] Python web support
- [x] Python 3 64-bit (3.6.0)
- [ ] Python IoT support
- [x] Python native development tools
- [ ] Azure Cloud Services core tools
- [ ] Python 2 64-bit (2.7.13)

Source: *Create a C++ extension for Python*

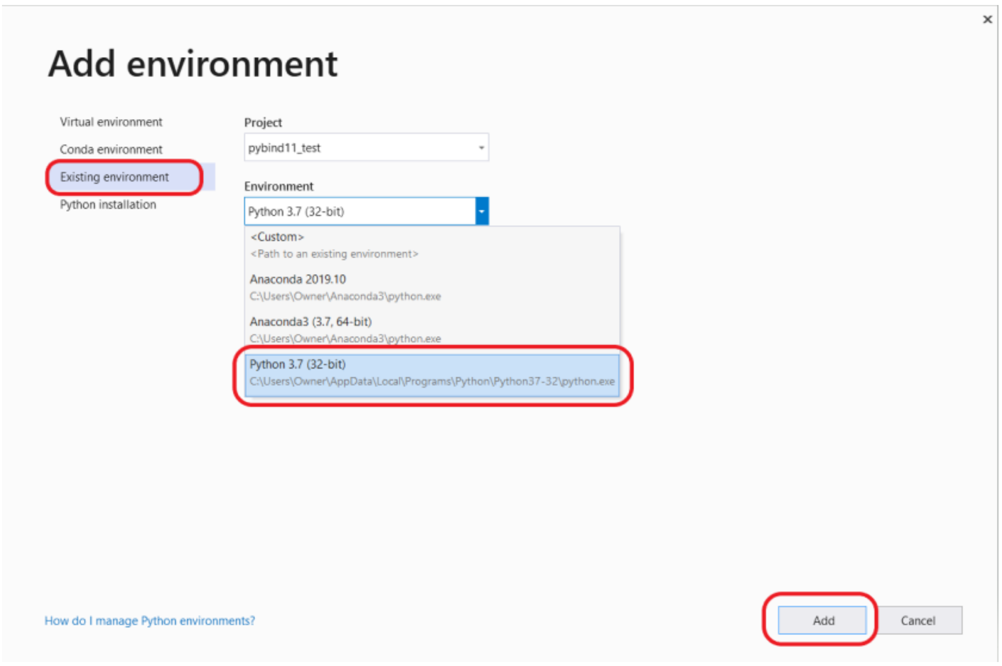
A.2 Create a Python Project

1. To create a Python project in Visual Studio, select **File** > **New** > **Project**. Search for "Python", select the **Python Application** template, give it a suitable name and location, and select **OK**.

2. pybind11 requires that you use a 32-bit Python interpreter (Python 3.6 or above recommended). In the **Solution Explorer** window of Visual Studio, expand the project node, then expand the **Python Environments** node. If you don't see a 32-bit environment as the default (either in bold, or labeled with **global default**), then right-click the **Python Environments** node and select **Add Environment**, or select **Add Environment** from the environment drop-down in the Python toolbar.

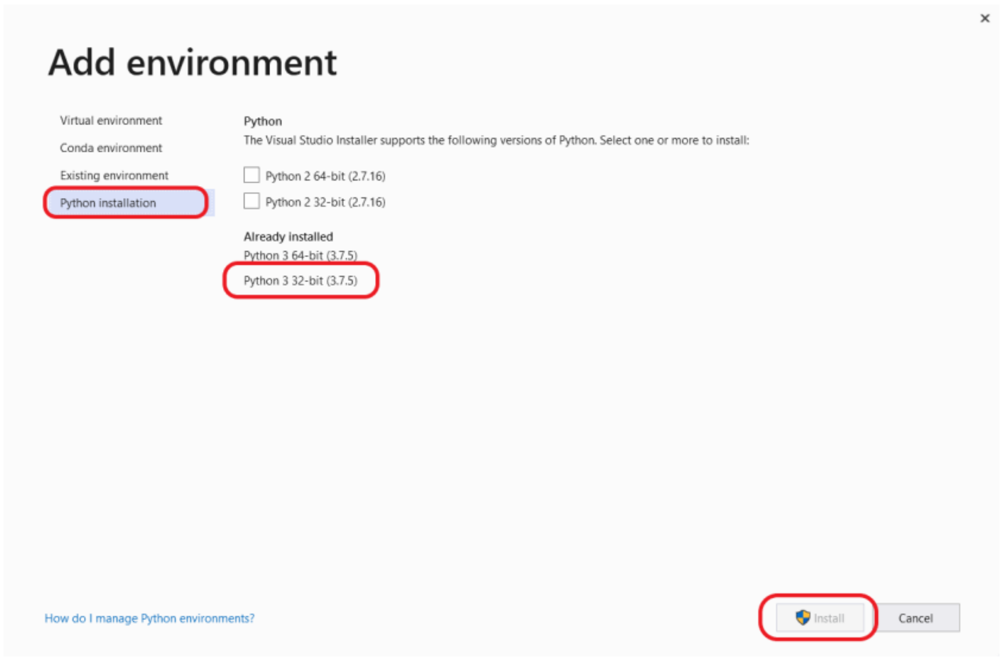


Once in the **Add Environment** dialog box, select the **Existing environment** tab, then select the 32-bit interpreter from the **Environment** drop down list.

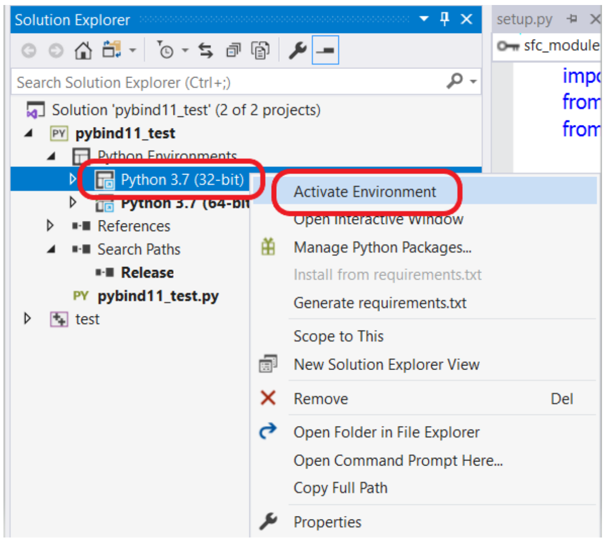

Add environment
Virtual environment
Conda environment
Existing environment
Python installation
<Custom>
<Path to an existing environment>
Anaconda 2019.10
C:\Users\Owner\Anaconda3\python.exe
Anaconda3 (3.7, 64-bit)
C:\Users\Owner\Anaconda3\python.exe
Python 3.7 (32-bit)
C:\Users\Owner\AppData\Local\Programs\Python\Python37-32\python.exe
How do I manage Python environments?
Add
Cancel

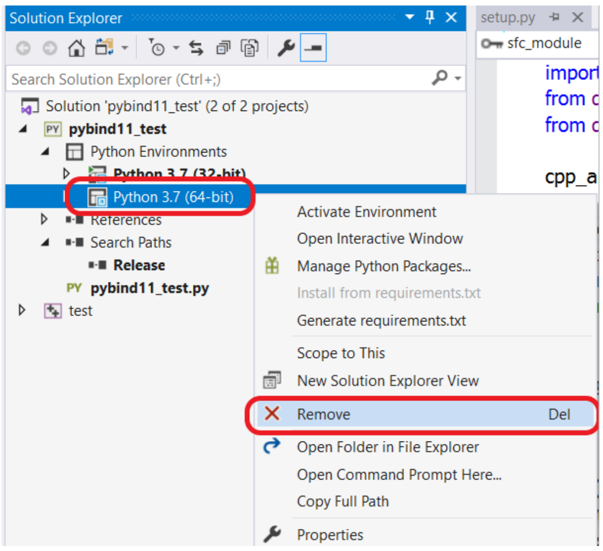
Project: pybind11_test
Environment: Python 3.7 (32-bit)

If you don't have a 32-bit interpreter installed, you can install standard python interpreters from the **Add Environment** dialog. Select the **Add Environment** command in the **Python Environments** window or the Python toolbar, select the **Python installation** tab, indicate which interpreters to install, and select **Install**.



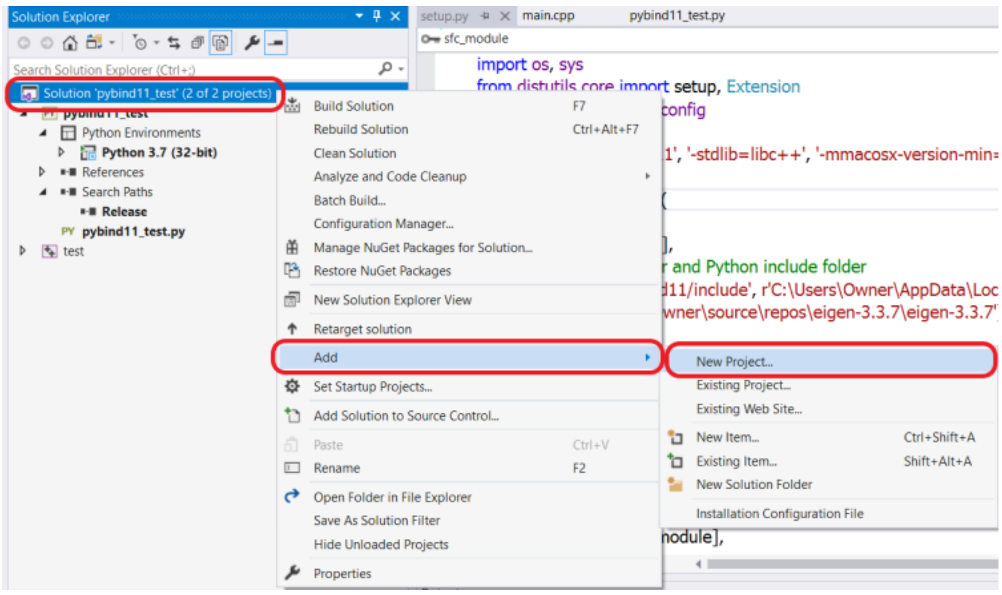
If you already added an environment other than the global default to a project, you may need to activate a newly added environment. Right-click that environment under the **Python Environments** node and select **Activate Environment**. To remove an environment from the project, select **Remove**.





A.3 Create a C++ Project

1. A Visual Studio solution can contain both Python and C++ projects together. To do so, right-click the solution in **Solution Explorer** and select **Add** > **New Project**.



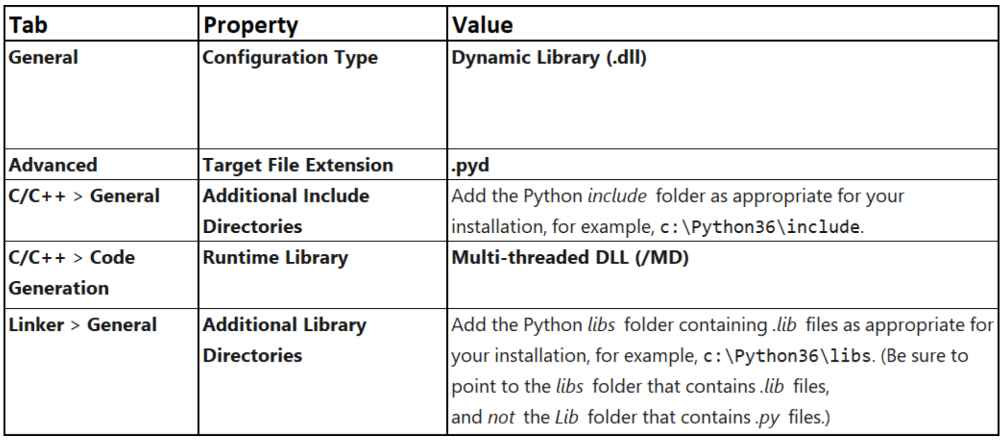
2. Search on "C++", select **Empty project**, specify the name "test", and select **OK**.

3. Create a C++ file in the new project by right-clicking the **Source Files** node, then select **Add** > **New Item**, select **C++ File**, name it main.cpp, and select **OK**.

4. Right-click the C++ project in **Solution Explorer**, select **Properties**.

5. At the top of the **Property Pages** dialog that appears, set **Configuration** to **All Configurations** and **Platform** to **Win32**.

6. Set the specific properties as described in the following table, then select **OK**.



Tab	Property	Value
General	Configuration Type	Dynamic Library (.dll)
Advanced	Target File Extension	.pyd
C/C++ > General	Additional Include Directories	Add the Python *include* folder as appropriate for your installation, for example, `c:\Python36\include`.
C/C++ > Code Generation	Runtime Library	Multi-threaded DLL (/MD)
Linker > General	Additional Library Directories	Add the Python *libs* folder containing *.lib* files as appropriate for your installation, for example, `c:\Python36\libs`. (Be sure to point to the *libs* folder that contains *.lib* files, and *not* the *Lib* folder that contains *.py* files.)

7. Right-click the C++ project and select **Build** to test your configurations (both **Debug** and **Release**). The *.pyd* files are located in the **solution** folder under **Debug** and **Release**, not the C++ project folder itself.

8. Add the following code to the C++ project's main.cpp file:

```
#include <iostream>

int main(){
  std::cout << "Hello World from C++" << std::endl;

  return 0;
}
```

9. Build the C++ project again to confirm the code is working.

A.4 Convert the C++ project to extension for Python

To make the C++ DLL into an extension for Python, first modify the exported methods to interact with Python types. Then, add a function that exports the module (main function).

1. Install pybind11 using pip:

```
pip install pybind11
```

or

```
py -m pip install pybind11
```

2. At the top of main.cpp, include pybind11.h:

```
#include <pybind11/pybind11.h>
```

3. At the bottom of main.cpp, use the PYBIND11_MODULE macro to define the entry point to the C++ function:

```
namespace py = pybind11;

PYBIND11_MODULE(test, m) {
    m.def("main_func", &main, R"pbdoc(
        pybind11 simple example.
    )pbdoc");

#ifdef VERSION_INFO
    m.attr("__version__") = VERSION_INFO;
#else
    m.attr("__version__") = "dev";
#endif
}
```

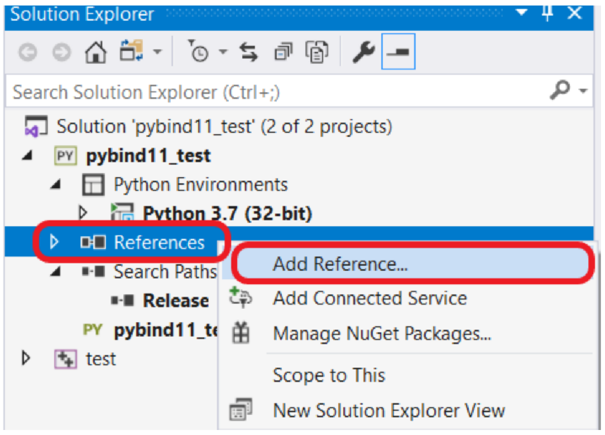
4. Set the target configuration to **Release** and build the C++ project to verify your code.

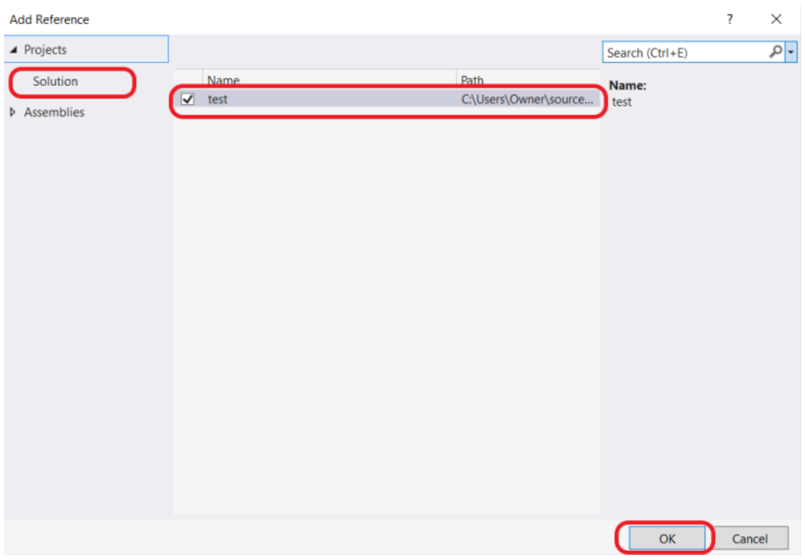
   The C++ module may fail to compile for the following reasons:

   - Unable to locate Python.h (**E1696: cannot open source file "Python.h"** and/or **C1083: Cannot open include file: "Python.h": No such file or directory**): verify that the path in **C/C++** > **General** > **Additional Include Directories** in the project properties points to your Python installation's include folder. See the table in Step 6.
   - Unable to locate Python libraries: verify that the path in **Linker** > **General** > **Additional Library Directories** in the project properties points to the Python installation's libs folder.See the table in Step 6.
   - Linker errors related to target architecture: change the C++ target's project architecture to match that of your Python installation. For example, if you're targeting x64 with the C++ project but your Python installation is x86, change the C++ project to target x86.

A.5 Make the DLL available to Python

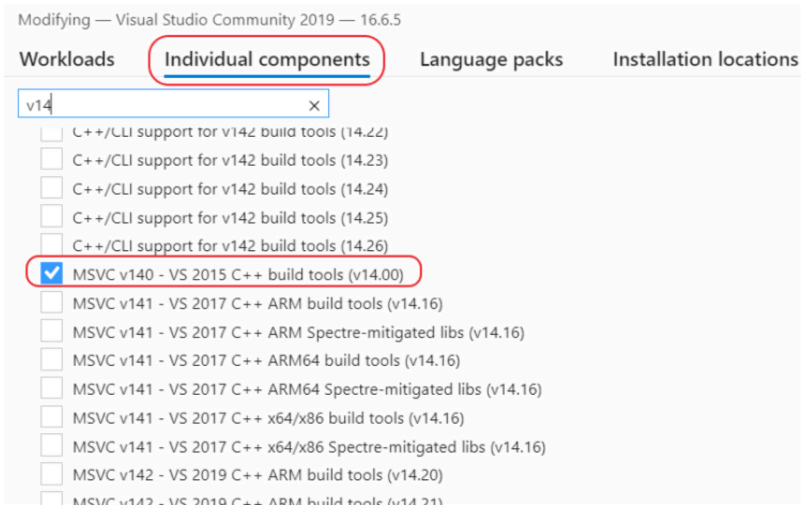
1. In **Solution Explorer**, right-click the **References** node in the Python project, and then select **Add Reference**. In the dialog that appears, select the **Projects** tab, select the **test** project, and then select **OK**.





	Name	Path
✓	test	C:\Users\Owner\source...

2. Run the Visual Studio installer, select **Modify**, select **Individual Components** > **Compilers, build tools, and runtimes** > **Visual C++ 2015.3 v140 toolset**. This step is necessary because Python (for Windows) is itself built with Visual Studio 2015 (version 14.0) and expects that those tools are available when building an extension through the method described here. (Note that you may need to install a 32-bit version of Python and target the DLL to Win32 and not x64.)



- [ ] C++/CLI support for v142 build tools (14.22)
- [ ] C++/CLI support for v142 build tools (14.23)
- [ ] C++/CLI support for v142 build tools (14.24)
- [ ] C++/CLI support for v142 build tools (14.25)
- [ ] C++/CLI support for v142 build tools (14.26)
- [✓] MSVC v140 - VS 2015 C++ build tools (v14.00)
- [ ] MSVC v141 - VS 2017 C++ ARM build tools (v14.16)
- [ ] MSVC v141 - VS 2017 C++ ARM Spectre-mitigated libs (v14.16)
- [ ] MSVC v141 - VS 2017 C++ ARM64 build tools (v14.16)
- [ ] MSVC v141 - VS 2017 C++ ARM64 Spectre-mitigated libs (v14.16)
- [ ] MSVC v141 - VS 2017 C++ x64/x86 build tools (v14.16)
- [ ] MSVC v141 - VS 2017 C++ x64/x86 Spectre-mitigated libs (v14.16)
- [ ] MSVC v142 - VS 2019 C++ ARM build tools (v14.20)
- [ ] MSVC v142 - VS 2019 C++ ARM build tools (v14.21)

3. Create a file named setup.py in the C++ project by right-clicking the project and selecting **Add** > **New Item**. Then select **C++ File (.cpp)**, name the file setup.py, and select **OK** (naming the file with the .py extension makes Visual Studio recognize it as Python despite using the C++ file template).

   Then, paste the following code into the setup.py file:

```python
import os, sys
from distutils.core import setup, Extension
from distutils import sysconfig

cpp_args = ['-std=c++11', '-stdlib=libc++',
'-mmacosx-version-min=10.7']

sfc_module = Extension(
'pybind11_test',
sources = ['main.cpp'],
# add pybind11 folder and Python include
folder
include_dirs=[r'pybind11/include',
r'C:\Users\Owner\AppData\Local\Programs\Python
\Python37-32\include'],
language='c++',
extra_compile_args = cpp_args,
)

setup(
    name = 'pybind11_test',
    version = '1.0',
    description = 'Simple pybind11 example',
    license      = 'MIT',
    ext_modules = [sfc_module],
)
```

4. The setup.py code instructs Python to build the extension using the Visual Studio 2015 C++ toolset when used from the command line.

   Open an elevated command prompt, navigate to **the folder that contains setup.py**, and enter the following command:

```
python setup.py install
```

A.6 Call the DLL from Python

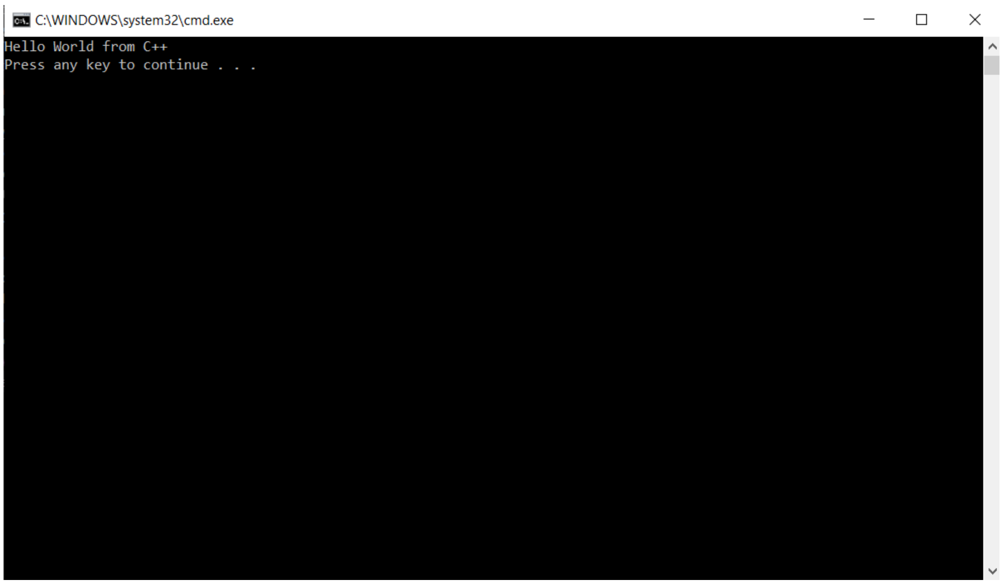
After DLL is made available to Python, we can now call the test.main_func() from Python code.

1. Add the following lines in your .py file to call methods exported from the DLL:

```python
import test

if __name__ == "__main__":
    test.main_func()
```

2. Run the Python program (**Debug** > **Start without Debugging**). The output should look like the following:

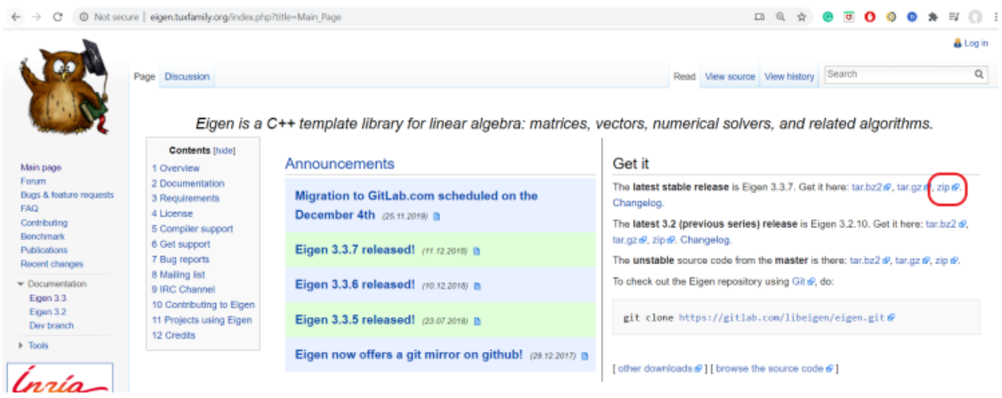


```
Hello World from C++
Press any key to continue . . .
```

A.7 Example: Store C++ Eigen Matrix as NumPy Arrays

This example requires the use of Eigen, a C++ template library. Due to its popularity, pybind11 provides transparent conversion and limited mapping support between Eigen and Scientific Python linear algebra data types.

1. If have not already, visit: `http://eigen.tuxfamily.org/index.php?title=Main_Page`. Click on **zip** to download the zip file.



Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

Announcements

Migration to GitLab.com scheduled on the December 4th (25.11.2019)

Eigen 3.3.7 released! (11.12.2018)

Eigen 3.3.6 released! (10.12.2018)

Eigen 3.3.5 released! (23.07.2018)

Eigen now offers a git mirror on github! (29.12.2017)

Get it

The **latest stable release** is Eigen 3.3.7. Get it here: tar.bz2, tar.gz, zip. Changelog.

The **latest 3.2 (previous series) release** is Eigen 3.2.10. Get it here: tar.bz2, tar.gz, zip. Changelog.

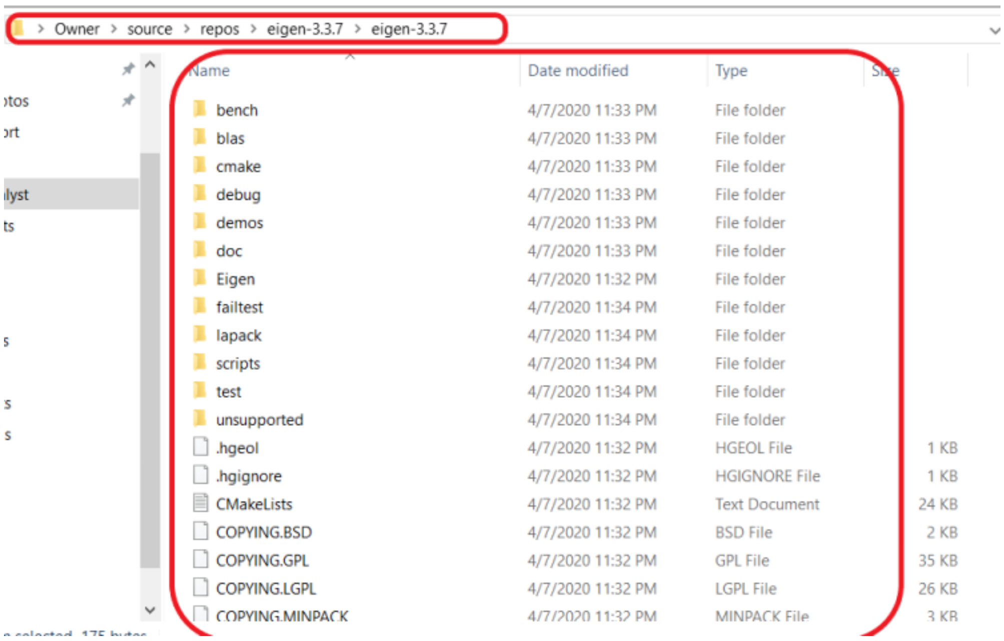
The **unstable** source code from the **master** is there: tar.bz2, tar.gz, zip.

To check out the Eigen repository using Git, do:

```
git clone https://gitlab.com/libeigen/eigen.git
```

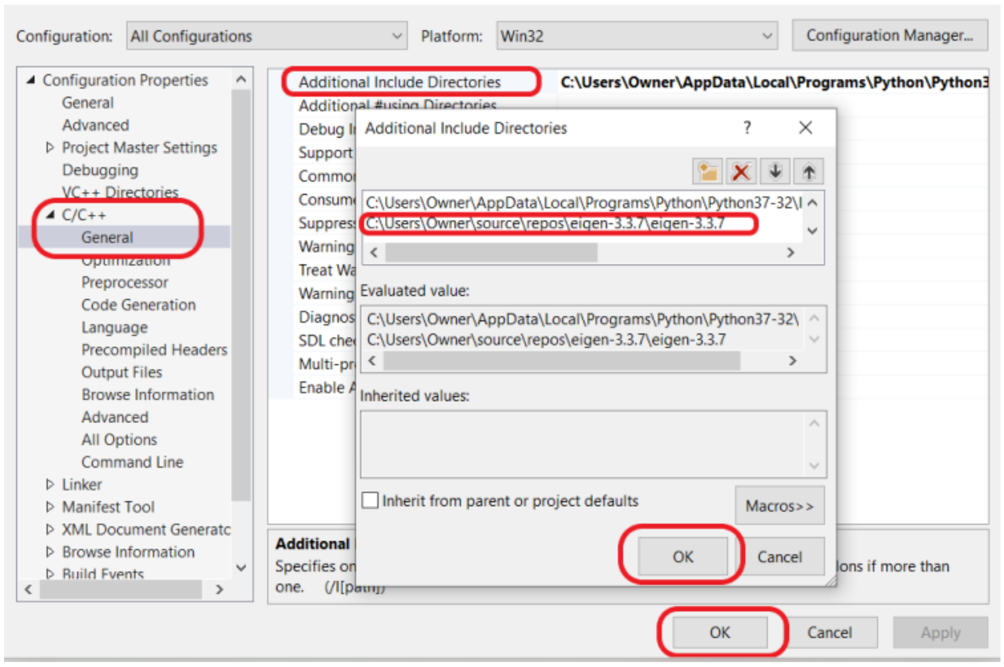
[other downloads] [browse the source code]

Once the download is complete, decompress the zip file in a suitable location. Note down the file path containing the folders as shown below:



Name	Date modified	Type	Size
bench	4/7/2020 11:33 PM	File folder	
blas	4/7/2020 11:33 PM	File folder	
cmake	4/7/2020 11:33 PM	File folder	
debug	4/7/2020 11:33 PM	File folder	
demos	4/7/2020 11:33 PM	File folder	
doc	4/7/2020 11:33 PM	File folder	
Eigen	4/7/2020 11:32 PM	File folder	
failtest	4/7/2020 11:34 PM	File folder	
lapack	4/7/2020 11:34 PM	File folder	
scripts	4/7/2020 11:34 PM	File folder	
test	4/7/2020 11:34 PM	File folder	
unsupported	4/7/2020 11:34 PM	File folder	
.hgeol	4/7/2020 11:32 PM	HGEOL File	1 KB
.hgignore	4/7/2020 11:32 PM	HGIGNORE File	1 KB
CMakeLists	4/7/2020 11:32 PM	Text Document	24 KB
COPYING.BSD	4/7/2020 11:32 PM	BSD File	2 KB
COPYING.GPL	4/7/2020 11:32 PM	GPL File	35 KB
COPYING.LGPL	4/7/2020 11:32 PM	LGPL File	26 KB
COPYING.MINPACK	4/7/2020 11:32 PM	MINPACK File	3 KB

2. In the **Solution Explorer**, right-click the C++ project, select **Properties**. Navigate to the **C/C++** > **General** tab, add the Eigen path to the **Additional Library Directories** Property, then select **OK**.



Additional Include Directories: C:\Users\Owner\AppData\Local\Programs\Python\Python3

Additional Include Directories

C:\Users\Owner\AppData\Local\Programs\Python\Python37-32\I
C:\Users\Owner\source\repos\eigen-3.3.7\eigen-3.3.7

Evaluated value:

C:\Users\Owner\AppData\Local\Programs\Python\Python37-32\
C:\Users\Owner\source\repos\eigen-3.3.7\eigen-3.3.7

Additional
Specifies on ... ions if more than one. (/I[path])

3. Right-click the C++ project and select **Build** to test your configurations (both **Debug** and **Release**).

4. Replace the code in main.cpp file with the following code:

```
# include <iostream>

// pybind11
#include <pybind11/pybind11.h>
#include <pybind11/eigen.h>
```

```


    // Eigen
    #include <Eigen/Dense>

    namespace py = pybind11;

    Eigen::Matrix3f example_mat(const int a,
    const int b, const int c) {
        Eigen::Matrix3f mat;
        mat << 5 * a, 1 * b, 2 * c,
               2 * a, 4 * b, 2 * c,
               1 * a, 3 * b, 3 * c;

        return mat;
    }

    Eigen::MatrixXd inv(Eigen::MatrixXd xs) {
        return xs.inverse();
    }

    double det(Eigen::MatrixXd xs) {
        return xs.determinant();
    }

    PYBIND11_MODULE(test, m) {
        m.def("example_mat", &example_mat);
        m.def("inv", &inv);
        m.def("det", &det);

    #ifdef VERSION_INFO
        m.attr("__version__") = VERSION_INFO;
    #else
        m.attr("__version__") = "dev";
    #endif
    }

```

Note that Eigen arrays are automatically converted to numpy arrays simply by including the pybind11/eigen.h header (see line 5 in 4). C++ function that has an Eigen return type can be directly use as a NumPy data type in Python.
Build the C++ project to check the code working correctly.

5. Include Eigen library's directory. Replace the code in setup.py with the following code:

```
    import os, sys
    from distutils.core import setup, Extension
```

```
from distutils import sysconfig

cpp_args = ['-std=c++11', '-stdlib=libc++', '-mmacosx-version-min=10.7']

sfc_module = Extension(
    'pybind11_test',
    sources = ['main.cpp'],
    # add pybind11 folder and Python include folder
    include_dirs = [r'pybind11/include',
    r'C:\Users\Owner\AppData\Local\Programs\Python\Python37-32\include',
    r'C:\Users\Owner\source\repos\eigen-3.3.7\eigen-3.3.7'],
    language = 'c++',
    extra_compile_args = cpp_args,
)

setup(
    name = 'pybind11_test',
    version = '1.0',
    description = 'Simple pybind11 example',
    license     = 'MIT',
    ext_modules = [sfc_module],
)
```

As before, in an elevated command prompt, navigate to the folder that contains setup.py, and enter the following command:

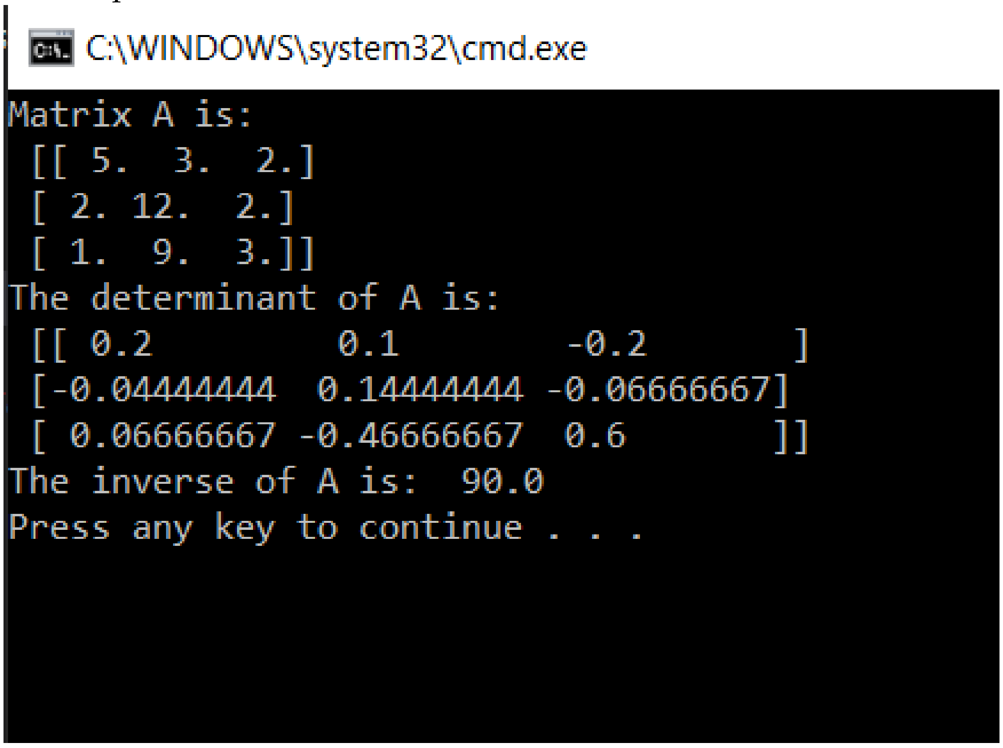
```
python setup.py install
```

6. Then, paste the following code in the Python file:

```
import test
import numpy as np

if __name__ == "__main__":
    A = test.example_mat(1,3,1)
    print("Matrix A is: \n" , A)
    print("The determinant of A is: \n", test.inv(A))
    print("The inverse of A is: ", test.det(A))
```

The output should be:



```
C:\WINDOWS\system32\cmd.exe
Matrix A is:
 [[ 5.   3.   2.]
 [ 2.  12.   2.]
 [ 1.   9.   3.]]
The determinant of A is:
 [[ 0.2         0.1        -0.2       ]
 [-0.04444444  0.14444444 -0.06666667]
 [ 0.06666667 -0.46666667  0.6       ]]
The inverse of A is:  90.0
Press any key to continue . . .
```

A.8 Example: Store Data Generated by Analytical Heston Model as NumPy Arrays

Only relevant part of the code will be displayed here. To request the code for the entire project, contact the author at cko22@bath.edu.

This example demonstrates how to use pybind11 for:

- C++ project with multiple files
- Store C++ Eigen Matrix as NumPy arrays

1. For multiple files C++ project, we need to include the source files in setup.py. This example also uses the Eigen library and so its directory will also be included. Paste the following code in setup.py:

```
    import os, sys
    from distutils.core import setup, Extension
    from distutils import sysconfig

    cpp_args = ['-std=c++11', '-stdlib=libc++', '-
mmacosx-version-min=10.7']

    sfc_module = Extension(
        'Data_Generation',
        sources = [r'module.cpp', r'Data.cpp', r'Heston.cpp', r'IntegrationScheme.cpp', r'IntegratorImp.cpp', r'NumIntegrator.cpp', r'pdetfunction.cpp', r'pfunction.cpp', r'Range.cpp'],
        include_dirs=[r'pybind11/include', r'C:\Users\Owner\AppData\Local\Programs\Python\Python37-32\include', r'C:\Users\Owner\source\repos\eigen-3.3.7\eigen-3.3.7'],
        language='c++',
        extra_compile_args = cpp_args,
    )

    setup(
        name = 'Data_Generation',
        version = '1.0',
        description = 'Python package with Data_Generation C++ extension (PyBind11)',
        license      = 'MIT',
        ext_modules = [sfc_module],
    )
```

Then, go to the folder that contains setup.py in command prompt, enter the following command:

```
python setup.py install
```

2. The code in module.cpp is:

```
//For analytic solution in case of Heston model
#include "Heston.hpp"
#include <ctime>
#include "Data.hpp"
#include <future>


//Inputting and Outputting
#include <iostream>
#include <vector>

// pybind11
#include <pybind11/pybind11.h>
#include <pybind11/eigen.h>
#include <pybind11/stl_bind.h>

// Eigen
//#include <C:\Users\Owner\source\repos\eigen-3.3.7\eigen-3.3.7\Eigen/Dense>
```

```
    #include <Eigen/Dense>

    namespace py = pybind11;

    struct CPPData {
    // Input Data
    std::vector<double> spot_price;
    std::vector<double> strike_price;
    std::vector<double> risk_free_rate;
    std::vector<double> dividend_yield;
    std::vector<double> initial_vol;
    std::vector<double> maturity_time;
    std::vector<double> long_term_vol;
    std::vector<double> mean_reversion_rate;
    std::vector<double> vol_vol;
    std::vector<double> price_vol_corr;

    // Output Data
    std::vector<double> option_price;
    };


    /*
        ....do something....
    */

    Eigen::MatrixXd module() {
        CPPData cppdata;
        simulation(cppdata);

        // Convert vector to eigen vector first
        // Input Data
        Map<Eigen::VectorXd> eigen_spot_price(
  cppdata.spot_price.data(), cppdata.spot_price.
  size());
        Map<Eigen::VectorXd> eigen_strike_price(
  cppdata.strike_price.data(), cppdata.
  strike_price.size());
        Map<Eigen::VectorXd> eigen_risk_free_rate(
  cppdata.risk_free_rate.data(), cppdata.
  risk_free_rate.size());
        Map<Eigen::VectorXd> eigen_dividend_yield(
  cppdata.dividend_yield.data(), cppdata.
  dividend_yield.size());
        Map<Eigen::VectorXd> eigen_initial_vol(
  cppdata.initial_vol.data(), cppdata.initial_vol
  .size());
```

```
     Map<Eigen::VectorXd> eigen_maturity_time(cppdata.maturity_time.data(), cppdata.maturity_time.size());
     Map<Eigen::VectorXd> eigen_long_term_vol(cppdata.long_term_vol.data(), cppdata.long_term_vol.size());
     Map<Eigen::VectorXd> eigen_mean_reversion_rate(cppdata.mean_reversion_rate.data(), cppdata.mean_reversion_rate.size());
     Map<Eigen::VectorXd> eigen_vol_vol(cppdata.vol_vol.data(), cppdata.vol_vol.size());
     Map<Eigen::VectorXd> eigen_price_vol_corr(cppdata.price_vol_corr.data(), cppdata.price_vol_corr.size());
     // Output Data
     Map<Eigen::VectorXd> eigen_option_price(cppdata.option_price.data(), cppdata.option_price.size());

     const int cols {11}; // No. of columns
     const int rows = SIZE * 4;
     // Define a matrix that store training data to be used in Python
     MatrixXd training(rows, cols);
     // Initialise each column using vectors
     training.col(0) = eigen_spot_price;
     training.col(1) = eigen_strike_price;
     training.col(2) = eigen_risk_free_rate;
     training.col(3) = eigen_dividend_yield;
     training.col(4) = eigen_initial_vol;
     training.col(5) = eigen_maturity_time;
     training.col(6) = eigen_long_term_vol;
     training.col(7) = eigen_mean_reversion_rate;
     training.col(8) = eigen_vol_vol;
     training.col(9) = eigen_price_vol_corr;
     training.col(10) = eigen_option_price;

     //     std::cout << training << std::endl;

     return training;
 };

 PYBIND11_MODULE(Data_Generation, m) {

     m.def("module", &module);
```

```

    #ifdef VERSION_INFO
        m.attr("__version__") = VERSION_INFO;
    #else
        m.attr("__version__") = "dev";
    #endif
    }
```

Note that the module function return type is Eigen. This can then be used directly as NumPy data type in Python.

3. The C++ function can then be called from Python as shown:

```
    import Data_Generation as dg
    import mysql.connector
    import pandas as pd
    import sqlalchemy as db
    import time

    ### This function calls analytical Heston
  option pricer in C++ to simulation 5,000,000
  option prices and store them in MySQL database

    """
    ....do something....

    """

    if __name__ == "__main__":
        t0 = time.time()

        # A new table (optional)
        #SQL_clean_table()

        target_size = 5000000 # Final table size
        batch_size = 500000 # Size generated each
  iteration

        # Get the number of rows existed in the
  table
        r = SQL_setup(batch_size);

        # Get the number of iterations
        iter = int(target_size/batch_size) - int(r
  /batch_size)
        print("Iteration(s) required: ", iter)
        count =0

        print("Now Run C++ code: ")
```

```python
    for i in range(iter):
        count = count +1
        print("---------------------------------------------------")
        print("Current iteration: ", count)
        cpp = dg.module() # store data from C++

        """
        ....do something....

        """

    r1=SQL_setup(batch_size)
    print("No. of rows in SQL Classic Heston Table Now: ", r1)
    print("\n")
    t1 = time.time()
    # Calculate total time for entire program
    total = t1 - t0
    print("THE ENTIRE PROGRAM TAKES: ", round(total), " s TO COMPLETE")

```

Appendix B

Asymptotic Expansions for General Stochastic Volatility Models

B.1 Asymptotic Expansions

We present the asymptotic expansions for general Stochastic Volatility Models by (Lorig et al., 2019). Consider a strictly positive spot price S whose risk-neutral dynamics are given by

$$\begin{aligned} S &= e^X && && \text{(B.1)} \\ dX &= -\frac{1}{2}\sigma^2(t,X,Y)dt + \sigma(t,X,Y)dW_1, && X(0) = x \in \mathbb{R} && \text{(B.2)} \\ dY &= f(t,X,Y)dt + \beta(t,X,Y)dW_2, && Y(0) = y \in \mathbb{R} && \text{(B.3)} \\ \langle dW_1, dW_2 \rangle &= \rho(t,X,Y)dt, && |\rho| < 1. && \text{(B.4)} \end{aligned}$$

The drift of X is selected to be $-\frac{1}{2}\sigma^2$ so that $S = e^X$ is martingale.

Let $C(t)$ be the time t value of a European call option, matures at time T > t with payoff function $P(X(T))$. To value a European-style option using risk-neutrla pricing we must compute functions of the form

$$u(t,x,y) := \mathbb{E}[P(X(T))|X(t) = x, Y(t) = y]$$

It is well-known that the function u satisfised the Kolmogorov Backward equation (Lorig et al., 2019)

$$(\partial_t + \mathcal{A}(t))u(t,x,y) = 0, \quad u(T,x,y) = P(x) \tag{B.5}$$

where the operator $\mathcal{A}(t)$ is given explicitly by

$$\mathcal{A}(t) = a(t,x,y)(\partial_x^2 - \partial_x) + f(t,x,y)\partial_y + b(t,x,y)\partial_y^2 + c(t,x,y)\partial_x\partial_y \tag{B.6}$$

and where the functions a, b and c are defined as

$$\begin{aligned} a(t,x,y) &:= \frac{1}{2}\sigma^2(t,x,y) \\ b(t,x,y) &:= \frac{1}{2}\beta^2(t,x,y) \\ c(t,x,y) &:= \rho(t,x,y)\sigma(t,x,y)\beta(t,x,y) \end{aligned}$$

Appendix C

Deep Neural Networks Library in Python: Keras

We show the code snippet for computing ANN in *Keras*. The Python code for defining ANN architecture for rough Heston implied volatility, approximation

```
#%% Define the neural network architecture
import keras
from keras import backend as K
keras.backend.set_floatx('float64')
from keras.callbacks import EarlyStopping

# Construct the model
input_layer = keras.layers.Input(shape=(n,))

hidden_layer1 = keras.layers.Dense(80, activation='elu
    ')(input_layer)
hidden_layer2 = keras.layers.Dense(80, activation='elu
    ')(hidden_layer1)
hidden_layer3 = keras.layers.Dense(80, activation='elu
    ')(hidden_layer2)

output_layer = keras.layers.Dense(100, activation='
    linear')(hidden_layer3)

model = keras.models.Model(inputs=input_layer, outputs
    =output_layer)

# summarize layers
model.summary()

# User-defined metrics: R2 and Max Abs Error
# R2
def coeff_determination(y_true, y_pred):
    SS_res =  K.sum(K.square( y_true-y_pred ))
    SS_tot = K.sum(K.square( y_true - K.mean(y_true) )
    )
    return  (1 - SS_res/(SS_tot + K.epsilon()))
#Max Error
def max_error(y_true,y_pred):
    return (K.max(abs(y_true-y_pred)))

earlystop = EarlyStopping(monitor="val_loss",
                          min_delta=0,
                          mode="min",
                          verbose=1,
                          patience= 25)

# Prepares model for training
#opt = keras.optimizers.Adam(learning_rate=0.005)
model.compile(loss='mse', optimizer='adam', metrics=['
   mae', coeff_determination, max_error])
```

Bibliography

Alos, Elisa et al. (June 2017). "Exponentiation of Conditional Expectations Under Stochastic Volatility". In: URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2983180.

Andersen, Leif et al. (Jan. 2007). "Moment Explosions in Stochastic Volatility Models". In: *Finance and Stochastics* 11, pp. 29–50. DOI: 10.1007/s00780-006-0011-7.

Atkinson, Colin et al. (Jan. 2011). "Rational Solutions for the Time-Fractional Diffusion Equation". In: URL: https://www.researchgate.net/publication/220222966_Rational_Solutions_for_the_Time-Fractional_Diffusion_Equation.

Bayer, Christian et al. (Oct. 2018). "Deep calibration of rough stochastic volatility models". In: URL: https://arxiv.org/abs/1810.03399.

Black, Fischer et al. (May 1973). "The Pricing of Options and Corporate Liabilities". In: URL: https://www.cs.princeton.edu/courses/archive/fall09/cos323/papers/black_scholes73.pdf.

Carr, Peter and Dilip B. Madan (Mar. 1999). "Option valuation using the fast Fourier transform". In: *Journal of Computational Finance*. URL: https://www.researchgate.net/publication/2519144_Option_Valuation_Using_the_Fast_Fourier_Transform.

Chapter 2 Modeling Process: k-fold cross validation. https://bradleyboehmke.github.io/HOML/process.html. Accessed: 2020-08-22.

Comte, Fabienne et al. (Oct. 1998). "Long Memory In Continuous-Time Stochastic Volatility Models". In: *Mathematical Finance* 8, pp. 291–323. URL: https://zulfahmed.files.wordpress.com/2015/10/comterenault19981.pdf.

Cox, JOHN C. et al. (Jan. 1985). "A THEORY OF THE TERM STRUCTURE OF INTEREST RATES". In: URL: http://pages.stern.nyu.edu/~dbackus/BCZ/discrete_time/CIR_Econometrica_85.pdf.

Create a C++ extension for Python. https://docs.microsoft.com/en-us/visualstudio/python/working-with-c-cpp-python-in-visual-studio?view=vs-2019. Accessed: 2020-07-21.

Duffy, Daniel J. (Jan. 2018). *Financial Instrument Pricing Using C++, Second Edition*. John Wiley Sons. ISBN: 978-0-470-97119-2.

Duffy, Daniel J. et al. (2012). *Monte Carlo Frameworks: Building customisable high-performance C++ applications*. John Wiley Sons, Inc. ISBN: 9780470060698.

Dupire, Bruno (1994). "Pricing with a Smile". In: *Risk Magazine*, pp. 18–20.

Eigen: The Matrix Class. https://eigen.tuxfamily.org/dox/group__TutorialMatrixClass.html. Accessed: 2020-07-22.

Eldan, Ronen et al. (May 2016). "The Power of Depth for Feedforward Neural Networks". In: URL: https://arxiv.org/abs/1512.03965.

Euch, Omar El et al. (Sept. 2016). "The characteristic function of rough Heston models". In: URL: https://arxiv.org/pdf/1609.02108.pdf.

— (Mar. 2019). "Roughening Heston". In: URL: https://ssrn.com/abstract=3116887.

Fouque, Jean-Pierre et al. (Aug. 2012). "Second Order Multiscale Stochastic Volatility Asymptotics: Stochastic Terminal Layer Analysis Calibration". In: URL: https://arxiv.org/abs/1208.5802.

Gatheral, Jim (2006). *The volatility surface : a practitioner's guide*. John Wiley Sons, Inc. ISBN: 978-0-471-79251-2.

Gatheral, Jim et al. (Oct. 2014). "Volatility is rough". In: URL: https://arxiv.org/abs/1410.3394.

— (Jan. 2019). "Rational approximation of the rough Heston solution". In: URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3191578.

Hebb, Donald O. (1949). *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press; 1 edition (12 Jun. 2002). ISBN: 978-0805843002.

Heston, Steven (Jan. 1993). "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". In: URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.3204&rep=rep1&type=pdf.

Hinton, Geoffrey et al. (Feb. 2015). "Overview of mini-batch gradient descent". In: URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Hornik et al. (Mar. 1989). "Multilayer feedforward networks are universal approximators". In: URL: https://deeplearning.cs.cmu.edu/F20/document/readings/Hornik_Stinchcombe_White.pdf.

— (Jan. 1990). "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: URL: http://www.inf.ufrgs.br/~engel/data/media/file/cmp121/univ_approx.pdf.

Horvath, Blanka et al. (Jan. 2019). "Deep Learning Volatility". In: URL: https://ssrn.com/abstract=3322085.

Hurst, Harold E. (Jan. 1951). "Long-term storage of reservoirs: an experimental study". In: URL: https://www.jstor.org/stable/2982267#metadata_info_tab_contents.

Hutchinson, James M. et al. (July 1994). "A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks". In: URL: https://alo.mit.edu/wp-content/uploads/2015/06/A-Nonparametric-Approach-to-Pricing-and-Hedging-Derivative-Securities-via-Learning-Networks.pdf.

Ioffe, Sergey et al. (Feb. 2015). "UBatch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: URL: https://arxiv.org/abs/1502.03167.

Itkin, Andrey (Jan. 2010). "Pricing options with VG model using FFT". In: URL: https://arxiv.org/abs/physics/0503137.

Kahl, Christian et al. (Jan. 2006). "Not-so-complex Logarithms in the Heston model". In: URL: http://www2.math.uni-wuppertal.de/~kahl/publications/NotSoComplexLogarithmsInTheHestonModel.pdf.

Kingma, Diederik P. et al. (Feb. 2015). “Adam: A Method for Stochastic Optimization”. In: URL: `https://arxiv.org/abs/1412.6980`.

Kwok, YueKuen et al. (2012). *Handbook of Computational Finance, Chapter 21*. Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-17254-0.

Lewis, Alan (Sept. 2001). “A Simple Option Formula for General Jump-Diffusion and Other Exponential Levy Processes”. In: URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=282110`.

Liu, Shuaiqiang et al. (Apr. 2019a). “A neural network-based framework for financial model calibration”. In: URL: `https://arxiv.org/abs/1904.10523`.

— (Apr. 2019b). “Pricing options and computing implied volatilities using neural networks”. In: URL: `https://arxiv.org/abs/1901.08943`.

Lorig, Matthew et al. (Jan. 2019). “Explicit implied vols for multifactor local-stochastic vol models”. In: URL: `https://arxiv.org/abs/1306.5447v3`.

Mandara, Dalvir (Sept. 2019). “Artificial Neural Networks for Black-Scholes Option Pricing and Prediction of Implied Volatility for the SABR Stochastic Volatility Model”. In: URL: `https://www.datasim.nl/application/files/8115/7045/4929/1423101.pdf`.

McCulloch, Warren et al. (May 1943). “A Logical Calculus of The Ideas Immanent in Nervous Activity”. In: URL: `http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf`.

Mostafa, F. et al. (May 2008). “A neural network approach to option pricing”. In: URL: `https://www.witpress.com/elibrary/wit-transactions-on-information-and-communication-technologies/41/18904`.

Peters, Edgar E. (Jan. 1991). *Chaos and Order in the Capital Markets: A New View of Cycles, Prices, and Market Volatility*. John Wiley Sons. ISBN: 978-0-471-13938-6.

— (Jan. 1994). *Fractal Market Analysis: Applying Chaos Theory to Investment and Economics*. John Wiley Sons. ISBN: 978-0-471-58524-4. URL: `https://www.amazon.co.uk/Fractal-Market-Analysis-Investment-Economics/dp/0471585246`.

pybind11 Documentation. `https://pybind11.readthedocs.io/en/stable/advanced/cast/eigen.html`. Accessed: 2020-07-22.

Ruf, Johannes et al. (May 2020). “Neural networks for option pricing and hedging: a literature review”. In: URL: `https://arxiv.org/abs/1911.05620`.

Schmelzle, Martin (Apr. 2010). “Option Pricing Formulae using Fourier Transform: Theory and Application”. In: URL: `https://pfadintegral.com/docs/Schmelzle2010%20Fourier%20Pricing.pdf`.

Shevchenko, Georgiy (Jan. 2015). “Fractional Brownian motion in a nutshell”. In: *International Journal of Modern Physics: Conference Series* 36. URL: `https://www.worldscientific.com/doi/abs/10.1142/S2010194515600022`.

Stone, Henry (July 2019). “Calibrating Rough Volatility Models: A Convolutional Neural Network Approach”. In: URL: `https://arxiv.org/abs/1812.05315`.

Using the C++ eigen library to calculate matrix inverse and determinant. `http://people.duke.edu/~ccc14/sta-663-2016/18G_C++_Python_pybind11.`

html#Using-the-C++-eigen-library-to-calculate-matrix-inverse-and-determinant. Accessed: 2020-07-22.

Wilmott, Paul (2006). *Paul Wilmott on Quantitative Finance, Vol 3*. John Wiley Sons, Inc.; 2nd Edition. ISBN: 978-0-470-01870-5.