King's College London
Department of Mathematics
Submission Cover Sheet for Coursework

The following cover sheet must be completed and submitted with any dissertation, project, coursework essay or report submitted as part of formal assessment for degree in the Mathematics Department.

You are not required to write your name on your work

Candidate number (this is found on your student record account)	AC10082
Module Code and Title	7CCMFM50 MSC Financial Mathematics Project
Title of Project/Coursework	Deep Pricing in the CEV Model

Declaration

By submitting this assignment I agree to the following statements:
I have read and understand the King's College London Academic Honesty and Integrity Statement that I signed upon entry to this programme of study.
I declare that the content of this submission is my own work.
I understand that plagiarism is a serious examination offence, an allegation of which can result in action being taken under the College's Misconduct regulations.

Your work may be used as an example of good practice for future students to refer to. If chosen, your work will be made available either via KEATS or by paper copy. Your work will remain anonymous; neither the specific mark nor any individual feedback will be shared. Participation is entirely optional and will not affect your mark. If you consent to your submission being used in this way please add an X in the box to the right.	

Deep Pricing in the CEV Model

by

Ziwei Zhang

Department of Mathematics
King's College London
The Strand, London WC2R 2LS
United Kingdom

Email: k21026702@kcl.ac.uk
Tel: +44 (0)7529212625

1 September 2022

Report submitted in partial fulfilment of the requirements for the degree of MSc in Financial Mathematics in the University of London

Abstract

In chapter one, I introduced some mathematical background. In chapter two, I used finite difference method (both explicit method and Crank-Nicolson scheme) to pricing American put option and used SOR method to fasten the calculation process of Crank-Nicolson scheme. Then, I used neural network to pricing American put option and showed its effectiveness. In chapter 3, I used Bloomberg data to compare the calibration effect of European put option under CEV model, jump diffusion model, and Heston model, then I did deep calibration under Heston model. Also, I used a different machine learning method "XGBoost" to pricing American put option under CEV model and compare this result to chapter2 and used American put option to do calibration under this new method.
Keywords: Finite different method, SOR method, neural network pricing, calibration, deep calibration, XGBoost.

Acknowledgements

Thanks to Professor Markus.Riedle who provides me with instructions on how to format my dissertation and some suggestions on finite difference methods. Thanks to Dr.John.Armstrong for his helpful lecture note and Python code on Numerical and Computational Methods in Finance. Thanks to Dr.Blanka.Horvath for machining learning course which provides me basic knowledge for deep learning.

Contents

Introduction

Ideally, option pricing should follow two principles. One of these two principles is a fast calculation, and the other is to be able to perform well in the market. Analytic pricing formula can meet the requirement of fast calculation, and that's an important reason that Black-Scholes formula proposed by Black and Scholes in 1973[6] still attractive in calculations despite many more realistic models came up. However, Black-Scholes model can't fit the market well because it doesn't capture the negative correlation between the stock prices change and volatility changes. To fix this, John C.Cox[14] introduced a new option pricing model which is known as constant elasticity of variance(CEV) model. This model characterizes an inverse dependence between the stock price and volatility by simply make volatility proportional to the power of the stock price and he derived an analytic formula for European option pricing under this model. Following the same logic, a standard way of researching option pricing seems to be first introducing a more complicated model and then derive an analytic formula of option price under this model. For example, Merton (1976)[21] for jump-diffusion model and Heston(1993) for Heston model.

However, we can not always find analytic formula for option pricing. For American option pricing under CEV model, we can only use numerical way to approximate the theoretical value. For example, Carr et al.[19] shows we can use finite difference method for pricing American option. Also, for other more realistic model like rough volatility models[2], we can only use numerical way like Monte-Carlo simulation. Horvath et al[2] shows that using numerical way can be very slow and rise serious problems in calibration and this is a major bottleneck for calibration.

Luckily, neural network provides us a way to approximate theoretical value in a very fast way and universal approximation theory1.4.3(b) shows neural network can approximate any reasonable function to any accuracy. After neural networks have been widely used, **tractability**[2] is not as important as before to determine the popularity of a stochastic model because we can

use neural network as the approximation function to theoretical price and don't have to derive an analytic formula.
The dissertation is organised as follows. In chapter one, I introduced some mathematical background. In chapter two, I used finite difference method to pricing American put option and used neural network to pricing American put option. In chapter 3, I used Bloomberg data to compare the calibration effect of European put option under CEV model, jump diffusion model, and Heston model, then I did deep calibration under Heston model. Also, I used a different machine learning method "XGBoost" to pricing American put option under CEV model and compare this result to chapter2 and used American put option to do calibration under this new method.

Chapter 1

Mathematical and Financial Background

1.1 Black-Scholes model and Black-Scholes PDE

Black-Scholes model is one of the most famous model for financial workers and it stands as building blocks of other important stochastic models. Black-Scholes PDE characterizes the process by which any contract price associated with the underlying changes over time and the PDE is derived by a simple portfolio.

1.1.1 The Black-Scholes assumptions

Black-Schole model has the following assumptions[15].

- the stock price S_t follows a **geometry Brownian motion**, which is

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

  where μ is a constant represents expected return on stock per year and σ is a constant called **volatility** which measures the standard deviation of the return and W_t is **Brownian motion**(also called Wiener process). In Forde's lecture note[17], a continuous time stochastic process $(W_t)_{t>0}$ is standard one-dimensional Brownian motion if it satisfies the following 4 properties:
  (1)$W_0 = 0$.

(2) W has independent increments,i.e. $W_{t_2} - W_{t_1}$, $W_{t_3} - W_{t_2}$,...$W_{t_n} - W_{t_{n-1}}$ are independent for all $0 \leq t_1 < t_2 ... < t_n$.
(3) The increments are **Normally distributed**(definition of normal distribution can be found in Riedle etal[16]): $W_t - W_s \sim N(0, t-s)$ for all $0 \leq s \leq t$.
(4) W_t is continuous as a function of t **almost surely**(The rigorous definition of almost surely continuous is in [23]).

- The short selling of securities with full use of proceeds is permitted.
- There are no transaction costs or taxes. All securities are perfectly divisible.
- There are no dividends during the life of the derivative.
- There are no riskless arbitrage opportunities.
- Security trading is continuous.
- The risk-free rate of interest, r, is constant and the same for all maturities.

1.1.2 Black-Scholes PDE

Under above assumptions, denote $V = V(S_t, t)$ to be the value of a contract related to S_t, (For example, it can be the value of European call option and V is a function of S_t and t) then V satisfies the following **Black − Scholes PDE**:

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

The derivation of Black and Scholes PDE is by constructing a portfolio and use no arbitrage theory. [15](In the later chapter, I will give a derivation of the PDE under CEV model using the same idea)

1.2 Constant Elasticity of Volatilty (CEV) model

1.2.1 motivation for CEV model and volatility smile

In Black and Scholes model, the volatility term σ is assumed to be a constant over time, but this assumption doesn't capture the fact that there is a negative correlation between the stock prices change and volatility changes. Nowadays, volatility smile has long been recognized by quantitative finance worker and financial learner or even ordinary stock marketer. If we look at the apple equity market and calculated **implied volatility**(this concept will be discussed in detail in Part3), in the assumption of Black and Scholes model, it should give us a constant over different strike price K. However, as shown in **Figure1.1**, the implied volatility decrease as strike price K increase and this phenomenon is called **volatility smile**, which implies that Black Scholes stock price model can't fit the market data well. Constant Elasticity of Volatilty (CEV) model, introduced by COX[14], intends to fix this problem, and the idea is by simply make volatility proportional to the power of the stock price.
In financial industry CEV model has been widely used by practitioners and perform well in equity and commodity market[24]. This results from its simple assumption and good property and the discussion will be presented in the next two parts.

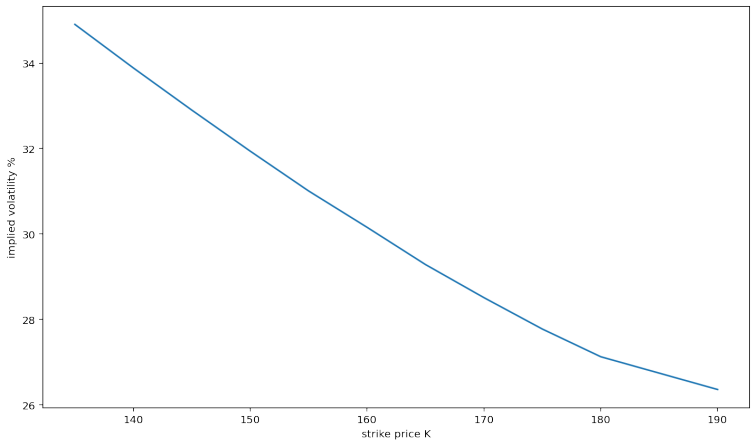


Figure 1.1: the volatility smile of Apple put option

1.2.2 The CEV Assumption

There are two assets in the CEV market model. The first is a risk-free asset that grows at a constant interest rate r. The second is a stock, the price of which is determined by the following stochastic differential equation at time t:

$$dS_t = \mu S_t dt + \sigma S_t^{\gamma} dW_t^1 \ . \tag{1.2}$$

Where μ and γ are the real-valued constants with $\gamma \in [0, 1]$ and W_t^1 is the Brownian motion. In this paper, the stock is assumed to follow the stochastic differential equation above until the first time when the stock price hit 0. We assume that the stock price is all 0 afterwards.

1.2.3 Properties of CEV model

PDE under CEV model
The derivation of PDE under CEV model is an essential step to calculate the price of American option by finite difference method in Chapter2. Following the same method as Hull[15], the PDE under CEV model can be derived as follows.
Denote $V = V(S, t)$ to be the value of a contract related to stock price. For example, it can be the value of European call option.
Use **Ito formula**[15] to V we have,

$$dV = (\frac{\partial V}{\partial S}\mu S + \frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\sigma^2 S^{2\gamma})dt + \frac{\partial V}{\partial S}\sigma S^{\gamma} dW_t^1. \tag{1.3}$$

Discrete (1.2) and (1.3), we have:

$$\Delta S = \mu S \Delta t + \sigma S^{\gamma} \Delta W_t^1 \ , \tag{1.4}$$

$$\Delta V = (\frac{\partial V}{\partial S}\mu S + \frac{\partial V}{\partial t} + \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\sigma^2 S^{2\gamma})\Delta t + \frac{\partial V}{\partial S}\sigma S^{\gamma} \Delta W_t^1 \ , \tag{1.5}$$

where $\Delta S, \Delta V$ means the change of S and V in the time interval Δt.

From the conclusion of Ito's lemmas[15], the two Brownian motion W_t^1 in (1.4) and (1.5) are the same. Therefore, we are able to construct a portfolio to eliminate it. The portfolio is to short this contract and long an amount $\frac{\partial V}{\partial S}$ of underlying S. Denote the value of this portfolio is Π, then we have

$$\Pi = -V + \frac{\partial V}{\partial S} S \ . \tag{1.6}$$

As the change of time Δt, the change value of this portfolio $\Delta\Pi$ satisfies:

$$\Delta\Pi = -\Delta V + \frac{\partial V}{\partial S}\Delta S \ . \tag{1.7}$$

Plug (1.4) and (1.5) into (1.7), we have:

$$\Delta\Pi = (-\frac{\partial V}{\partial t} - \frac{1}{2}\frac{\partial^2 V}{\partial S^2}\sigma^2 S^{2\gamma})\Delta t \ . \tag{1.8}$$

The equation above has no dW_t term, so it is a deterministic function of t, which means it is risk-less. In **risk neutral world**, by **no arbitrage argument**(details can be found in Forde's [17]), it should earn the same rate of return as the risk-free return. Therefore, we have:

$$\Delta\Pi = r\Pi\Delta t.$$

Substitute $\Delta\Pi$ by (1.8), we have

$$(\tfrac{\partial V}{\partial t} + \tfrac{1}{2}\tfrac{\partial^2 V}{\partial S^2}\sigma^2 S^{2\gamma})\Delta t = r(V - \tfrac{\partial V}{\partial S}S)\Delta t.$$

Therefore, we have the following PDE:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^{2\gamma}\frac{\partial^2 V}{\partial S^2} = rV \ . \tag{1.9}$$

The terminal density function under CEV model
Cox[14] derived the terminal density function S_T conditional on S_t under CEV model. This conclusion is important when pricing European option under CEV model and will be mentioned later. Under CEV model, the terminal density function S_T conditional on S_t and risk free rate r is

$$p(S_T, T; S_t, r) = (2-\gamma)k^{\frac{1}{2-\gamma}}(xz^{1-2\gamma})^{\frac{1}{2}(\frac{1}{2-\gamma})}I_{\frac{1}{2-\gamma}}(2(xz)^{\frac{1}{2}}) \ , \tag{1.10}$$

where

$$k = \tfrac{2r}{\sigma^2(2-\gamma)[e^{r(2-\gamma)(T-t)}-1]} \ ,$$

$$x = kS_t^{2-\gamma}e^{r(2-\gamma)(T-t)} \ ,$$

$$z = kS_T^{2-\gamma} \ ,$$

and I_q is the **modified Bessel function** of order q.[3]
Scaling relationship
The price of the American put option with strike K and maturity T has the following equation[13]:

$$P(S_0, K, T, r, \sigma, \gamma) = \frac{1}{\lambda} P(\lambda S_0, \lambda K, \alpha T, r/\alpha, \lambda^{1-\gamma}\alpha^{-\frac{1}{2}}\sigma, \gamma) \ .$$

The relationship will be discussed in detail in Chapter2.

1.3 Derivatives pricing

1.3.1 Feynman-Kac formula

Feynman-Kac formula[5] tells us, under suitable regularity and integrability conditions, the solution of the Black-Scholes PDE

$$\frac{\partial V}{\partial t} + \mu(S)\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2(S)\frac{\partial^2 V}{\partial S^2} = rV \tag{1.11}$$

with boundary condition

$$V(S, T) = f(S) \ , \tag{1.12}$$

can be expressed as

$$V(S, t) = e^{-r(T-t)} E^Q(f(S_T)|S_t = S) \ , \tag{1.13}$$

where S has dynamics starting from initial value $S_t = S$, and satisfies

$$dS_t = \mu(S_t)dt + \sigma(S_t)dW_t^Q \ . \tag{1.14}$$

W_t^Q is Brownian motion under **Q measure**[5] and f is the payoff function. $\mu(S_t)$ and $\sigma(S_t)$ are functions of S_t. For example, in Black and Scholes model $\mu(S_t) = rS_t$ and $\sigma(S_t) = \sigma S_t$.

1.3.2 European option pricing

The Black and Scholes model and CEV model mentioned above are in P measure[16]. Feynman-Kac formula tells us that we can price options in Q measure and use equation(1.13) to calculate the price of the option. **Girsanov's theorem**[5]

tells us the relationship between P measure and Q measure. Apply this theory, in Q measure, the CEV model is:

$$dS_t = rS_t dt + \sigma S_t^{\gamma} dW_t^Q \ , \tag{1.15}$$

where W_t^Q is the Brownian motion in measure Q. Denote K is the strike price and T is the time to maturity. For European call option, $f(S_T) = (S_T - K)^+$, where $(S_T - K)^+$ means the maximum value between $(S_T - K)^+$ and 0. Using equation(1.13) The price of European call option is :

$$V(S,t) = e^{-r(T-t)} E^Q((S_T - K)^+ | S_t = S) \ . \tag{1.16}$$

Using the terminal density function in equation(1.10) and calculate $V(S,t)$, we can get the price of European call option under CEV model, the exact calculation process is very difficult and can be found in COX.[14] In Hull's book[15], it gives us a concise formula for European options under CEV model. Denote C,P to be the price of European call and put option, when $0 < \gamma < 1$ they satisfies:

$$C = S_t[1 - \chi^2(a, b+2, c)] - Ke^{-r(T-t)}\chi^2(c, b, a) \ ,$$

$$P = Ke^{-r(T-t)}[1 - \chi^2(c, b, a)] - S_t\chi^2(a, b+2, c) \ ,$$

where

$$a = \frac{[Ke^{-r(T-t)}]^{2(1-\gamma)}}{(1-\gamma)^2 v}, b = \frac{1}{1-\gamma}, c = \frac{S^{2(1-\gamma)}}{(1-\gamma)^2 v} \ ,$$

with

$$v = \frac{\sigma^2}{2r(\gamma-1)}[e^{2r(\gamma-1)(T-t)} - 1] \ .$$

$\chi^2(m, n, l)$ is the probability of an event $P(M \leq m)$ under chi-squared distribution with noncentrality parameter n and l degrees of freedom.

1.3.3 Monte-Carlo simulation for CEV model

Monte-Carlo simulation for CEV model can be used to pricing European put option under CEV model. Follwing the **Euler scheme**[9], suppose we have N time steps, then we have:

$$S_{t+(i+1)\delta t} = S_{t+i\delta t} + rS_{t+i\delta t}\delta t + \sigma S_{t+i\delta t}^{\gamma}\sqrt{\epsilon_t}\delta t \ , \tag{1.17}$$

with initial value $S_t = S$, where $\delta t = \frac{T-t}{N}$ is the size of time step, and ϵ_t is the random number following the standard Gaussian distribution.($\epsilon_t \sim N(0,1)$). Following (1.17), we can simulate M(M is a large number usually above 10000) paths of stock price indexed by 1,2,3,...M. At time T, we know each path's value S_T^1, S_T^2,...S_T^M, according to equation(1.13), the European put option price is:

$$V(S,t) = e^{-r(T-t)} \frac{((K - S_T^1)^+ + (K - S_T^2)^+ ... + (K - S_T^M)^+)}{M} , \qquad (1.18)$$

where $(K - S_T^1)^+$ means the maximum value between $(K - S_T^1)$ and 0.

1.3.4 American option pricing

The American option gives holder the right to exercise it before time to maturity T. Intuitively, it's price should be higher than the price of European option with the same strike K and time to maturity T. The rigorous proof for this can be found in Wilmott et al.[20]. Because the price of the American option is higher, there must be certain values of S for which exercising the American option is optimal. Because if this is not the case, the Black and Scholes PDE would hold for all S and solving it yields the same price as the European option. However, at each time t, for each value of S, we don't know whether to exercise it or not. This kind of problem is known as **free boundary problem** . At each time t, a specific value of S denotes the boundary between two regions: on one side, the option should be held, and on the other, it should be exercised. We refer to this value, which changes over time, as the **optimum exercise price**, denoted by $S^*(t)$
According to Wilmott et al.[20], the American put option price has unique solution if it has following constraints:

- $V(S,t) \geq (K - S)^+$;
- the Black-Scholes PDE is replaced by inequality;
- $V(S,t)$ is a continuous function of S;
- $\frac{\partial V}{\partial S}$ is continuous.

In summary, according to Jing.et al[26]. the price of American put options under CEV models satisfies the following conditions.
In the exercise region $[0,T] \times [0, S^*(t)]$,

$$\frac{\partial V(S,t)}{\partial t} + rS\frac{\partial V(S,t)}{\partial S} + \frac{1}{2}\sigma^2 S^{2\gamma}\frac{\partial^2 V(S,t)}{\partial S^2} - rV(S,t) < 0 \ ,$$

$$V(S,t) = K - S = (K-S)^+ \ .$$

In the continuation region $[0,T] \times (S^*(t), +\infty)$,

$$\frac{\partial V(S,t)}{\partial t} + rS\frac{\partial V(S,t)}{\partial S} + \frac{1}{2}\sigma^2 S^{2\gamma}\frac{\partial^2 V(S,t)}{\partial S^2} - rV(S,t) = 0 \ ,$$

$$V(S,t) > (K-S)^+ \ .$$

The boundary conditions at $S^*(t)$ are that V and $\frac{\partial V}{\partial S}$ are continuous and satisfies:

$$V(S^*(t),t) = (K - S^*(t))^+, \frac{\partial V}{\partial S}(S^*(t),t) = -1. \tag{1.19}$$

Note: The exact calculation of V(S,t) is by using finite difference method which will be presented in detail in Part2.

1.3.5 Put call parity

Put call parity refers to a relationship between the value of call option and put option. When we are pricing, we are able to focus only on put option or call option and calculate the other one using this relationship. There are two forms of put-call parity from Jing.et al[26]. For European options, we have:

$$V_p(S,t) = V_c(S,t) + Ke^{-r(T-t)} - S0 \ ,$$

where $V_p(S,t)$ is the price of European put option and $V_c(S,t)$ is the price of European call option.
For American option, if the local volatility σ only depends on S, we have:

$$V_c(S,t;r,\sigma(S)) = V_p(K^2/S,t;r,\sigma(S))\frac{S}{K} \ ,$$

where $V_c(S,t;r,\sigma(S))$ is the price of American call option knowing r and $\sigma(S)$, and $V_p(K^2/S,t;r,\sigma(S))$ is the price of American put option with stock price K^2/S knowing r and $\sigma(S)$.

1.4 Machine learning and deep learning

1.4.1 Motivation

These years, machine learning has been widely used in all kinds of industry including artificial intelligence, data science and finance. In finance, there are a lot of quantitative work, for example, pricing derivatives, risk management, hedging and asset pricing. Many of these problems can be generalised as regression problem in the broad sense where we have many independent variables and some desired outputs. Take option pricing as an example, the independent variables can be current stock price, interest rates, volatility σ and different parameters in the underlying stochastic process and the output is the price of the option. These problems can be solved by supervised learning algorithms. A popular way is by using deep learning, which is using neural networks to implement supervised learning.
This method is appealing and may start a revolution in derivatives pricing area. In the past, tractability of stochastic models has been one of the most decisive qualities in determining their popularity[1] because it can be very time-consuming only use numerical ways like Mote-Carlo for approximating theoretical pricing and when we move to calibration part, this disadvantage can be amplified by many times(see more in part 3). It is this reason that many studies have been done to find a closed form solution to the derivative pricing under a particular stochastic model, from options price under Black and Scholes model[21], closed form formula for option price under CEV model[14] to closed form formula for option price under Heston model[22]. However, after the appearance of neural networks, tractability of stochastic models is not as important as before. After offline training to the neural networks, all parameters in neural networks are fixed and the process of using numerical methods to approximate theoretical price is replaced by using the fixed neural networks which is as fast as a closed form formula.

1.4.2 Definition of feedforward neural network

Now we introduced the definition of a feedforward neural network using the material in Horvath. et al[2].
Let $I, O, r \in N$, where I is the dimension of the input layer, O is the dimension of output layer and r is the number of all layer minis one. A function $\boldsymbol{f}$: $R^I \to R^O$ is a feedforward neural network with r-1 hidden layers, where there are $d_i \in N$ units in the i-th hidden layer for any $i = 1, 2...r-1$, and

activation functions $\boldsymbol{\sigma_i} : R^{d_i} \to R^{d_i}, i = 1, 2...r$, where $d_r = O$,if

$$\boldsymbol{f} = \boldsymbol{\sigma_r} \circ \boldsymbol{L_r} \circ ... \circ \boldsymbol{\sigma_1} \circ \boldsymbol{L_1},$$

where $\boldsymbol{L_i} : R^{d_i} \to R^{d_i}$, for any $i = 1, ...r$, is an affine function

$$\boldsymbol{L_i} = W^i \boldsymbol{x} + \boldsymbol{b}^i, \boldsymbol{x} \in R^{d_{i-1}} \ ,$$

parameterised by ***weight matrix*** $W^i = [W^i_{j,k}]_{j=1,2...d_1,k=1,...d_{i-1}} \in R^{d_i \times d_{i-1}}$ and ***bias vector*** $\boldsymbol{b^i} = (b^i_1 ... b^i_{d_i}) \in R^{d_i}$, with $d_0 = I$.
We can denote the class of such functions $\boldsymbol{f}$ by

$$\mathcal{N}_r(I, d_1, ..., d_{r-1}, O; \boldsymbol{\sigma_1}, ...\boldsymbol{\sigma_r}). \tag{1.20}$$

An example of feedforward neural networks is shown in Figure[2], where I is the dimension of input layer and O refers to the dimension of output layer. $\boldsymbol{L_i}$ is the linear transformation from the (i-1)th layer to the ith layer and $\boldsymbol{\sigma i}$ refers to the activation function of the ith layer.

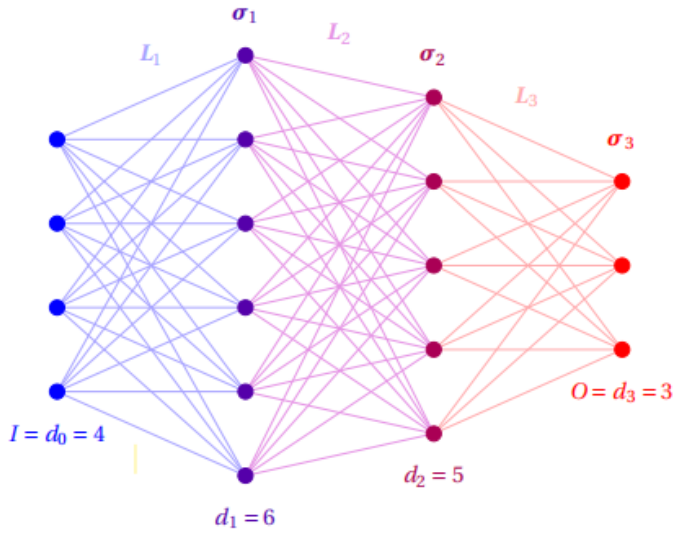


Figure 1.2: an example of neural networks(Horvath et al [2])

1.4.3 Properties of neural network

Here we introduce some important theories of neural network from Horvath et al.[2]. Before doing this, we introduce some notations and concepts. Denote

$C^k(R^n, R^m)$ to be the class of functions $R^n \to R^m$, whose partial derivatives exist and are continuous up to order k, $k \in N \cup \{\infty\}$.
(a) If $\boldsymbol{\sigma_i} \in C^{m_i}(R^{d_i}, R^{d_i})$ for any $i = 1, ...r$, then

$$\mathcal{N}_r(I, d_1, ..., d_{r-1}, O; \boldsymbol{\sigma_1}, ...\boldsymbol{\sigma_r}) \subset C^{min\{m_1,...m_r\}}(R^I, R^O) \ .$$

(b)(Universal approximation theory). Let $g : R \to R$ be a measurable function and satisfies:

- g is not a **polynominal function**,
- g is **bounded on any finite interval**,
- the closure of the set of all discontinuity points of g in R has zero **Lebesgue measure**,

Let $K \subset R^I$ be a **compact set**. For any $\epsilon > 0$, for any $u \in C(K, R)$, there exist $d \in N$ and $\boldsymbol{f} \in \mathcal{N}_2(I, d, 1; g, Id)$ such that:

$$\| u - f \|_{sup,K} < \epsilon \ ,$$

where Id is the identity function defined as:

$$Id(x) = x \text{ for } x \in R,$$

and $\| \ . \ \|_{sup,K}$ is the **sup morm**.
Note: For the meaning of concepts which are bolded, please refer to [2].
Remarks:(a) tells us that the **smoothness**[2] of activation function has a strong impact on the smoothness of neural network. (b)(Universal approximation theory) guarantees that neural networks have the ability to fit any reasonable function u to any accuracy. However, this theory doesn't tell us how many notes we need for a certain accuracy, nor tell us how the approximation function and original function looks like.

1.4.4 Optimise parameters in the neural networks

Hyperparameters refers to the number of hidden layers and the number of nodes in each hidden layer and these hyperparameters are chosen in advance by the architect of the neural network and don't need to be optimised . Actual

parameters are those parameters in each linear transformation. These parameters need to be optimised in order to make neural network approximates any reasonable function u as close as possible. Loss function l: $R^O \times R^O \to R^+$ is used in this process, where R^+ means the positive real number field, O means the dimension of output layer. Optimizing neural network $\boldsymbol{f}$ is then changed to minimising empirical risk $\mathcal{L}(\boldsymbol{f})$ which is defined as follows:
For training pairs $(\mathbf{x}^i, \mathbf{y}^i)$ i=1,2...N, $\mathbf{x} \in R^I$, $\mathbf{y} \in R^O$,

$$\mathcal{L}(\boldsymbol{f}) := \frac{1}{N} \sum_{i=1}^{N} l(\boldsymbol{f}(\mathbf{x}^i), \mathbf{y}^i) \ ,$$

where $\boldsymbol{f}(\mathbf{x}^i)$ is the neural network prediction of the i-th sample and $\mathbf{y}^i$ is the actual label of the i-th sample. l: $R^O \times R^O \to R^+$ is loss function measures the distance between two vectors.
In this paper, I choose root mean squared loss function which is defined as:

$$l(\hat{\mathbf{y}}, \mathbf{y}) = \sqrt{\frac{1}{O} \sum_{j=1}^{O} (\hat{\mathbf{y}}_j - \mathbf{y}_j)^2} \ ,$$

where $\mathbf{y}_j$ is the j-th element in vector $\mathbf{y}$.
Minimising empirical risk $\mathcal{L}(\boldsymbol{f})$ is by stochastic gradient descent (SGD). The idea of stochastic gradient descent is to divide the whole training set into many small subsets called minibatchs(indexed by 1,2,...k) and update parameters in neural network θ via:

$$\theta_{\mathbf{i}} = \theta_{\mathbf{i-1}} - \eta \bigtriangledown_\theta \mathcal{L}_{B_i}(\theta_{\mathbf{i-1}}), i = 1, 2...k \ ,$$

where $\bigtriangledown_\theta \mathcal{L}_{B_i}(\theta_{\mathbf{i-1}})$ is the gradient direction of $\mathcal{L}_{B_i}(\theta)$ when $\theta = \theta_{\mathbf{i-1}}$. The details about calculate gradient numerically is by using backpropagation, and these content can be found in Horvath et al.[1]

Chapter 2

Numerical Example

2.1 Finite difference methods for solving PDE in CEV model for American put option

2.1.1 Boundary conditions

As shown in (1.19), the whole domain for S is an infinite region. To use finite difference method, we need to set boundary conditions in order to iterate successfully. In other words, we want to limit the range of S in a finite interval $[S_{min}, S_{max}]$ and set the value of American put option on each boundary.
In the assumption of CEV model, if the stock price hit zero, then it remains 0 for all the subsequent time. Therefore, 0 is a nature bound for S_{min}. The problem is to find S_{max} sufficient enough to calculate all the American put option price in each grid that we are interested in. If S_t is much larger than strike price K, then from time t to time T, the stock price is very likely to be larger than K all the time, so we don't have the chance to exercise the option. Therefore, the price of American put option $V(S_t, t)$ is very close to 0 when S_t is much larger than strike K.
We notice that CEV model is just the Black-Scholes model when $\gamma = 1$, and we can derive that the change in the log of the stock price from time t to time T is normally distributed with standard deviation $\sigma\sqrt{T-t}$ as follows. Using Ito's formula to $logS_t$:

$$dlogS_t = (r - \tfrac{1}{2}\sigma^2)dt + \sigma dW_t \ ,$$

Integral both sides from t to T, we have

$$logS_T - logS_t = (r - \tfrac{1}{2}\sigma^2)(T-t) + \sigma(W_T - W_t) \ .$$

Therefore

$$logS_T \sim N(logS_t + (r - \tfrac{1}{2}\sigma^2)(T-t), \sigma^2(T-t)) \ .$$

We can ignore the term $(r - \frac{1}{2}\sigma^2)(T-t)$, because it is relatively small compared to $logS_t$ when $T-t$ is small and get:

$$logS_T \sim N(logS_t, \sigma^2(T-t)) \ .$$

From statistical view, the probability of $|logS_T - logS_t| > 5\sigma\sqrt{T-t}$ is very small. Therefore, we can approximately assume that $|logS_T - logS_t| \leq 5\sigma\sqrt{T-t}$. Then, if

$$logS_t - logK \geq 5\sigma\sqrt{T-t} \ , \tag{2.1}$$

we can get $logS_T - logK \geq 0$. From (2.1) and according to the discussion above, we can get

$$S_{max} = Ke^{5\sigma\sqrt{T}}. \tag{2.2}$$

After choosing S_{max}, we then able to list all the boundary conditions for finite difference method as follows.

- **Bottom boundary**:$V(0,t) = K - 0 = K$ because when $S = 0$, we can exercise it immediately.
- **Final boundary** :$V(S,T) = (K-S)^+$.
- **Top boundary** :$V(S_{max}, t) = 0$.

2.1.2 Explicit method and Crank-Nicolson scheme

Following Armstrong's method[9]. For the finite difference scheme, we need to discretize the domain. For S, we uniformly discretize it as $\{S_0 = S_{min}, S_1, ..., S_M = S_{max}\}$ and for t, we uniformly discretize it as $\{t_0 = 0, t_1, ...t_N = T\}$.
Then, we have

$$\delta S = \tfrac{S_{max} - S_{min}}{M} \ ,$$

and

$$\delta t = \frac{T}{N} \ .$$

Denote $V_t^i = V(S_i, t)$ and use the central difference estimate

$$\frac{\partial V}{\partial S}(S_i, t) \approx \frac{V(S_{i+1},t)-V(S_{i-1},t)}{2\delta S} = \frac{V_t^{i+1}-V_t^{i-1}}{2\delta S} \ ,$$

$$\frac{\partial^2 V}{\partial S^2}(S_i, t) \approx \frac{V(S_{i+1},t)-2V(S_i,t)+V(S_{i-1},t)}{\delta S^2} = \frac{V_t^{i+1}-2V^i+V_t^{i-1}}{\delta S^2} \ .$$

The PDE in CEV model is

$$\frac{\partial V}{\partial t} + rS_i\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^{2\gamma}\frac{\partial^2 V}{\partial S^2} - rV^i = 0 \ .$$

For each S_i in the grid (i=1,2,...M-1), plug it in the PDE above and we can get

$$\frac{dV^i}{dt} + \frac{1}{2}\sigma^2 S_i^{2\gamma}\frac{V^{i+1}-2V^i+V^{i-1}}{\delta S^2} + rS_i\frac{V^{i+1}-V^{i-1}}{2\delta S} - rV^i \approx 0 \ ,$$

rearranging the system of ODEs, we have

$$\frac{dV^i}{dt} \approx a_i V_t^{i-1} + b_i V_t^i + c_i V_t^{i+1} \ ,$$

with

$$a_i = -\frac{\sigma^2 S_i^{2\gamma}}{2(\delta S)^2} + \frac{rS_i}{2\delta S} \ ,$$

$$b_i = r + \frac{\sigma^2 S_i^{2\gamma}}{(\delta S)^2} \ ,$$

$$c_i = -\frac{\sigma^2 S_i^{2\gamma}}{2(\delta S)^2} - \frac{rS_i}{2\delta S} \ ,$$

and the boundary conditions

$$V_t^M = (K - S_M)^+ \ ,$$

$$V_t^0 = K - 0 = K \ .$$

Writing the ODE system in the form of matrix.

$$\frac{d\mathbf{V}}{dt} = \frac{d}{dt}\begin{pmatrix} V_t^1 \\ V_t^2 \\ V_t^3 \\ \vdots \\ V_t^{M-2} \\ V_t^{M-1} \end{pmatrix} \approx$$

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & b_{M-2} & b_{M-1} \\ 0 & 0 & 0 & 0 & \dots & a_{M-2} & a_{M-1} \end{pmatrix} \begin{pmatrix} V_t^1 \\ V_t^2 \\ V_t^3 \\ \vdots \\ V_t^{M-2} \\ V_t^{M-1} \end{pmatrix} + \begin{pmatrix} a_1 V_t^0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ c_{M-1} V_t^M \end{pmatrix}$$

$$\text{denote } \mathbf{\Lambda} = \begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & b_{M-2} & b_{M-1} \\ 0 & 0 & 0 & 0 & \dots & a_{M-2} & a_{M-1} \end{pmatrix}$$

$$\text{and } W_t = \begin{pmatrix} a_1 V_t^0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ c_{M-1} V_t^M \end{pmatrix}$$

Then, the ODE systems can be written as

$$\frac{d\mathbf{V}}{dt} \approx \mathbf{\Lambda} V_t + \mathbf{W_t}$$

Using backward difference, we can get the explicit way of finite difference scheme

$$\frac{\mathbf{V_t} - \mathbf{V_{t-\delta t}}}{\delta t} \approx \mathbf{\Lambda V_t} + \mathbf{W_t}$$

Rearranging it, we can get the recursion formula

$$\mathbf{V_{t-\delta t}} \approx (\mathbf{I} - \mathbf{\Lambda}\delta_t t)\mathbf{V_t} - \delta_t \mathbf{W_t} \tag{2.3}$$

where $\mathbf{I}$ is the identity matrix. Taking average of two estimators ($\mathbf{V_{t+\delta t}} + \mathbf{W_{t+\delta t}}$) and ($\mathbf{V_t} + \mathbf{W_t}$), we can derive the Crank-Nicolson scheme

$$\frac{\mathbf{V_{t+\delta t}} - \mathbf{V_t}}{\delta t} \approx \frac{1}{2}\mathbf{\Lambda}(\mathbf{V_{t+\delta t}} + \mathbf{V_t}) + \frac{1}{2}(\mathbf{W_{t+\delta t}} + \mathbf{W_t})$$

$$\mathbf{V_t} \approx (\mathbf{I} + \frac{1}{2}\mathbf{\Lambda})^{-1}((\mathbf{I} - \frac{1}{2}\mathbf{\Lambda})\mathbf{V_{t+\delta t}} - \frac{1}{2}\delta t(\mathbf{W_{t+\delta t}} + \mathbf{W_t})) \tag{2.4}$$

In Wilmott et al[20], if $V(S_t, t)$ is the price of American put option, the equation(2.3) and equation(2.4) become:

$$\mathbf{V_{t-\delta t}} \approx \mathbf{maximum}((\mathbf{I} - \mathbf{\Lambda}\delta_t t)\mathbf{V_t} - \delta_t \mathbf{W_t}, (\mathbf{K} - \mathbf{S})^+), \tag{2.5}$$

$$\mathbf{V_t} \approx \mathbf{maximum}((\mathbf{I} + \frac{1}{2}\mathbf{\Lambda})^{-1}((\mathbf{I} - \frac{1}{2}\mathbf{\Lambda})\mathbf{V_{t+\delta t}} - \frac{1}{2}\delta t(\mathbf{W_{t+\delta t}} + \mathbf{W_t})), (\mathbf{K} - \mathbf{S})^+), \tag{2.6}$$

where $(\mathbf{K} - \mathbf{S})^+ = ((K - S_0)^+, ...(K - S_M)^+)^T$.$((.)^T$ means the transpose of a vector). For vector $\mathbf{a} = (a_0, a_1, ...a_M)^T$ and vector $\mathbf{b} = (b_0, b_1, ...b_M)^T$, $\mathbf{maximum}(\mathbf{a}, \mathbf{b})$ is defined as: $(max(a_0, b_0), max(a_1, b_1), ...max(a_M, b_M))^T$, where $max(a_0, b_0)$, means the larger value between a_0 and b_0.

2.2 result of finite difference scheme

2.2.1 Result of option price and SOR method

Following the steps of 2.1, when using 10000 time steps and 100 stock prices steps, we can get the following result

S=1 K=0.95,T=1,r=0.02,$\sigma = 0.2, \gamma = 0.7$	Value	time
American put price by explicit scheme	0.04889	2.554s
American put price by Crank-Nicolson scheme	0.0489	12.262s

As shown above, there is not much difference in the put option price between explicit scheme and Crank-Nicolson scheme, but the Crank-Nicolson scheme took more than five times as long as explicit scheme. If we increase the time steps, the time difference between these two methods will become more and

more obvious. In my experiment, if I choose more than 200000 time steps, the Crank-Nicolson scheme will run out of memory in Python. This is because the Crank-Nicolson scheme iteration requires solving the inverse of a high-dimensional matrix. As we increase the number of steps, although we avoid directly solving the inverse of a high-dimensional matrix by solving a system of linear equations using Gaussian row elimination, or **LV factorization**[9], this is still very time-consuming.
In fact, Gaussian elimination is a direct method of solving a system of linear equations, and it gives us the totally accurate solution. However, in Crank-Nicolson scheme, the matrix $(\mathbf{I} + \frac{1}{2}\mathbf{\Lambda})$ is a **sparse matrix**[9]. Using iteration method can be a stable and fast way to deal with sparse matrix. **Gauss − Seidel** method and **Successive Over Relaxation**(**SOR**) **method** are two famous iteration methods. In my experiment, I choose the SOR method and the key idea of SOR methods are as follows.[12]. In order to solve

$$\mathbf{Ax} = \mathbf{b} \ .$$

We add a matrix term $\mathbf{Q}$, and solving the equation above is equivalent to solve:

$$\mathbf{Qx} = (\mathbf{Q} - \mathbf{A})\mathbf{x} + \mathbf{b} \ .$$

Using the idea of iteration[12], we can get ,

$$\mathbf{Qx}^{(k)} = (\mathbf{Q} - \mathbf{A})\mathbf{x}^{(k-1)} + \mathbf{b} \ .$$

Partition $\mathbf{A}$ to 3 parts, $\mathbf{L}, \mathbf{D}, \mathbf{U}$, where $\mathbf{L}$ is the lower triangle part of $\mathbf{A}$, $\mathbf{D}$ is the diagonal of $\mathbf{A}$, and $\mathbf{U}$ is the upper triangle part of $\mathbf{A}$ and set

$$\mathbf{Q} = \omega^{-1}(\mathbf{D} - \omega\mathbf{L})$$

$$\mathbf{G} = (\mathbf{D} - \omega\mathbf{L})^{-1}(\omega\mathbf{U} + (1 - \omega)\mathbf{D})$$

We can get the following iteration

$$(\mathbf{D} - \omega\mathbf{L})\mathbf{x}^{(k)} = \omega(\mathbf{U}\mathbf{x}^{(k-1)} + b) + (1 - \omega)\mathbf{D}\mathbf{x}^{(k-1)}$$

Here is the answer of Crank-Nicolson scheme using SOR.(in appendix I write both SOR function and Crank-Nicolson scheme using SOR)

S=1 K=0.95,T=1,r=0.02,$\sigma = 0.2, \gamma = 0.7$	Value	time
American put price by Crank-Nicolson SOR scheme	0.048858	12.262s

Although it doesn't reduce the time very much, iteration methods of solving a sparse system of linear equations can be very potential in implicit finite difference scheme. Many studies have been done in this part, and a very good way is projected successive overrelaxation (PSOR) method[20]

2.2.2 Exercise boundary and comparison between American and European put option

Following the steps in 2.1, we can plot the surface of the price of American put option and European put option in Figure2.1 and Figure2.2.
It maybe hard to see the difference between the two types of option in 3D plot. Therefore, we can plot these option prices at time 0 in Figure2.3. As shown in this picture, the price of the American put option is the same as K-S until S reaches the optimum exercise boundary at time 0 which is 0.695. Also, the price of the European option is below K-S and when S grows larger, it become very close to the price of the American put option. Therefore, this verifies that the price of the American option is always greater than the price of the European option, because American option has the right to exercise earlier. In this case, when S is small enough, American put option can have the right to exercise it and get the immediate K-S while European put option can only wait until maturity.
The optimum exercise price $S^*(t)$ is ploted in Figure2.5, this curve is also called exercise boundary. It is easy to show the exercise boundary at time T is just K, because at time T, when S is less than K, we should exercise the put option to get K-S and when S is greater than K we shouldn't exercise it. As shown from the picture, the boundary converges to K when $t \to T$. This makes sense, because as $t \to T$, there is less chance for S to vary, if S at time t is above K, it will very likely above K at time T, and if S is below K, it will very likely to below K at time T . Therefore, the exercise boundary $\to T$ when $t \to T$.
Look this picture closely, we can see the exercise boundary is jagged. This is because we discretize S and t when using finite difference method, and when we move from $\mathbf{V}_{(i+1)\delta t}$ to $\mathbf{V}_{i\delta t}$ (i=1,2...N) using (2.6) and use these two vectors subtract $(\mathbf{K} - \mathbf{S})(\mathbf{K} = (K, K, ...K)^T, \mathbf{S} = (S_0, S_1, ...S_M)^T$,$\mathbf{S}$ is a vector,$(...)^T$ means transpose) to get two new vectors, it maybe the same position in these two new vectors that gives the first positive value. A schematic is shown in Figure 2.4

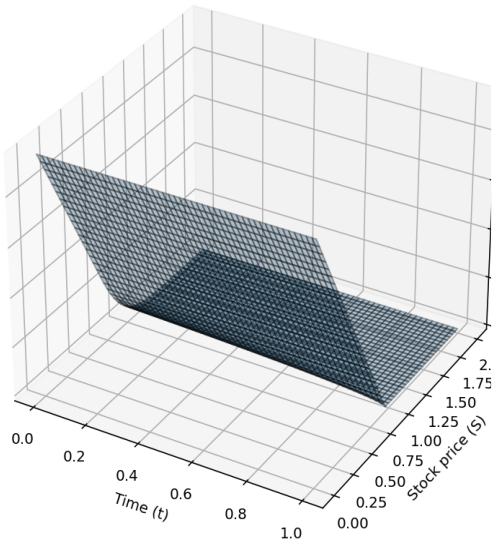


Figure 2.1: American put option price surface

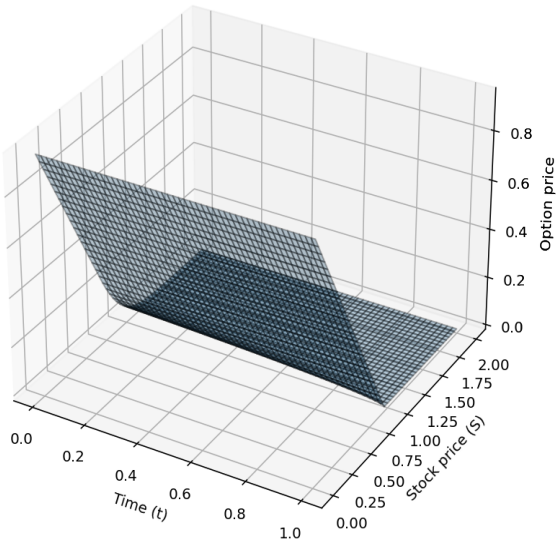


Figure 2.2: European put option price surface

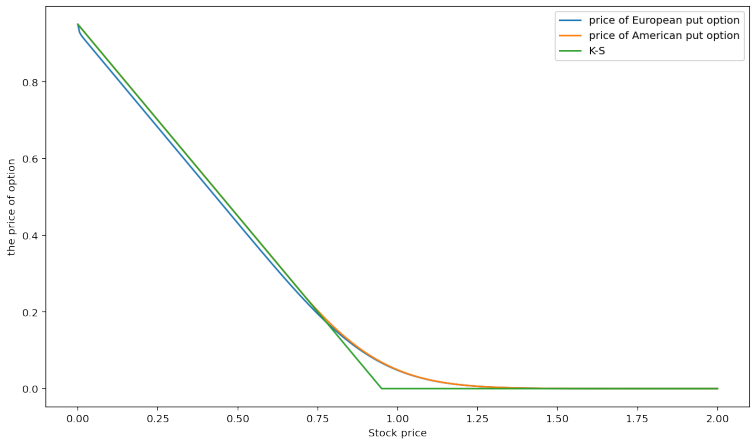


Figure 2.3: prices for different options at t=0

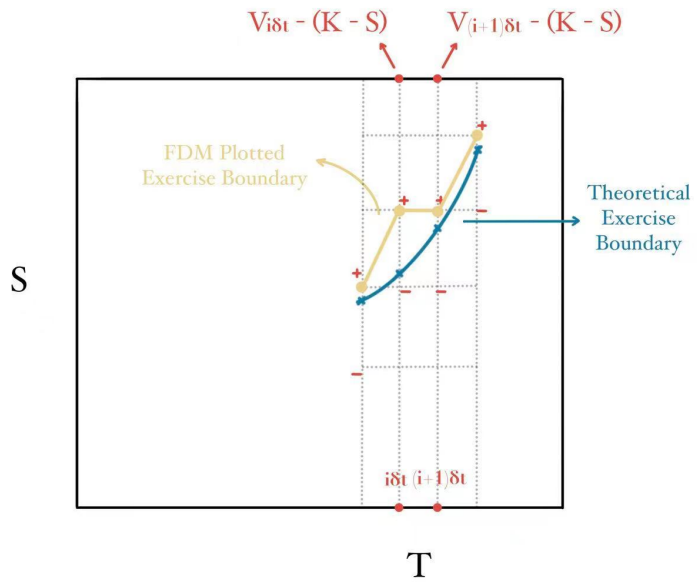


Figure 2.4: schemetic of exercise boundary

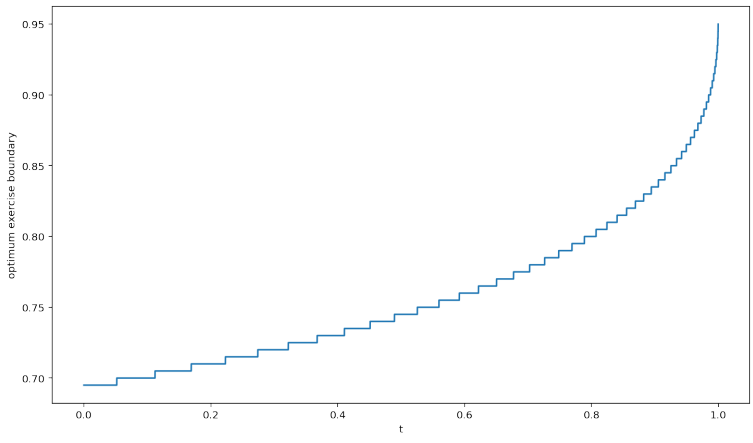


Figure 2.5: optimum exercise boundary for K=0.95

2.2.3 Test of FDM

In CEV model, we can not have the analytic formula of the price of American option. Also, the Monte Carlo simulation is not suitable for American option. The reason is that although Monte Carlo simulation can simulate different stock paths from time t to T, we don't know exercise boundary between t to T, therefore we don't know the value of American option on each simulated stock path because we don't know at each time point between t and T, we should exercise it or not.
Although we can't use Monte-Carlo for pricing American put option strictly, it can be a good way for testing under certain conditions. From Figure2.3, at time 0, as S grows, the price of European put option becomes more and more close to the price of American put option and when the stock price S is greater than 1, these two curves almost overlap. Therefore, we can use Monte-Carlo simulation to calculate the price of European put option in CEV model with a relatively large initial stock price S, for example, S=1.2. Under such assumptions, the price of European put option calculated by Monte-Carlo simulation should be very close to the price of American put option calculated by finite difference scheme.
Also, I use an app called options pricing suit[18] to test the price of American put option. The app is capable of calculating the price of American put option under Black and Scholes model. Setting $\gamma = 1$, the CEV model is just the Black and Scholes model so we can compare the price of American put option using finite difference method to the price calculated by the app. Here are the results.

S=1.2 K=0.95,T=1,r=0.02,$\sigma = 0.2, \gamma = 0.7$	Value
American put price by explicit scheme	0.00967
European put price by Monte-Carlo	0.00958

S=1.2,K=0.95,T=1,r=0.02,$\sigma = 0.2, \gamma = 1$	Value
American put price by explicit scheme	0.0104
American put price by app	0.0103

As can be seen from the table above, FDM works very well.

2.3 Two ways of using scaling relationship

So far, we already calculate the price of American put option in finite difference scheme. However, using finite difference scheme can be time-consuming due to the fact of iteration and that's the reason why we want to use deep pricing. In deep pricing, we want the neural networks to calculate the price of American put option by choosing any six parameters in the CEV model. The first step for deep pricing is to create the training set reasonably, and the value of finite difference scheme can be considered as theoretical value in training set.
We notice that in the CEV model, we have 6 parameters to choose. If we only use 10 values for each parameter and calculate its theoretical value as training set. We should run the finite difference scheme 10^6 times! Which is not tolerable in realistic. Luckily, we have a scaling relationship for the price of American option in the CEV model.
The scaling relationship is

$$P(S_0, K, T, r, \sigma, \gamma) = \frac{1}{\lambda} P(\lambda S_0, \lambda K, \alpha T, r/\alpha, \lambda^{1-\gamma}\alpha^{-\frac{1}{2}}\sigma, \gamma) \ . \tag{2.7}$$

If we rearrange the scaling relationship we can have the following form

$$\lambda P(S_0, K/\lambda, T, r\alpha, \alpha^{\frac{1}{2}}\sigma/\lambda^{1-\gamma}, \gamma) = P(\lambda S_0, K, \alpha T, r, \sigma, \gamma) \ . \tag{2.8}$$

Therefore, we have two methods of creating dataset. One is using (2.8) before the training of neural networks to create a dataset of 6 inputs and get the price of any 6 parameters directly. The other is using (2.7) and fix K and σ to create a datasets of 4 inputs and then use the scaling relationship to get the price of any 6 parameters after the training of neural network.

2.3.1 Method 1

Analysising the form of (2.8), if we fix K, r, σ, γ no matter what λ and α we choose, we can know $P(\lambda S_0, K, \alpha T, r, \sigma, \gamma)$ in a single finite difference scheme as long as λS_0 and αT are inside the boundary of finite difference scheme. Therefore, in order to calculate the left hand side of (2.8) what we need to do is just to find λS_0 and αT and plug it into the right hand side. The mathematical idea is below.
Suppose we want to calculate $P(s, k, t, r_0, e, g)$. Then we have,

$$\begin{cases} \mathrm{S}_0 = s \\ \mathrm{K}/\lambda = k \\ \mathrm{T}{=}\mathrm{t} \\ \mathrm{r}\alpha = r_0 \\ \alpha^{\frac{1}{2}}\sigma/\lambda^{1-\gamma} = e \\ \gamma = g \ . \end{cases}$$

Notice, $K, r, \sigma, \gamma, s, k, t, r_0, e, g$ are known. Solve these equations, we have

$$\begin{cases} \lambda S_0 = \frac{Ks}{k} \\ \alpha T = \frac{r_0}{r} t \ . \end{cases}$$

Also, for any s, k, r_0, t we need to ensure $\lambda S_0 < S_{max}$ and $\alpha T < 1$. Therefore, we have

$$\begin{cases} \frac{Ks}{k} < S_{max} \\ \frac{r_0 t}{r} < 1 \ . \end{cases}$$

These two inequalities can be satisfied when $K = K_{min}$ and $r = r_{max}$.
Plug λS_0 and αT to the right hand side of (2.8), we then be able to know $P(s, k, t, r_0, e, g)$.
Notice that in the scaling relationship, the sixth parameter γ is the same in both left and right hand side. Therefore, we have no benefit in scaling relationship if we want to change g. The only way we want to know the price for a different g is running another finite difference scheme with the same g.
Also, e is automatically determined by s, k, t, r_0, g.
Then, we know exactly how to create a data set of different six parameters:
1.Fix K, r, σ.
2.Choose a $\gamma = g$ and run a finite difference scheme.
3.Change any reasonable s, k, t, r_0 and use scaling method above to calculate e and get λ, α, and use (2.8) to get $P(s, k, t, r_0, e, g)$
4.change a different g and repeat 1 to 3.
After doing all these work, we then have a huge data set which has input s, k, t, r_0, e, g and output $P(s, k, t, r_0, e, g)$. The best advantage of this method is that we only use finite difference scheme once when we choose a γ, which greatly reduces the time of creating data set. However, the cost is that when we use the data set for training the neural networks, it's a six inputs problem and the more input parameters for training, the less accuracy it will be.
Notice, in method 1, if λS_0 and αT are not points on the grid of finite

difference scheme, we should use interpolation and λS_0 should be less than S_{max} and αT should be less than 1. In python, there are two interpolation function interp1d and interp2d. If only one of λS_0 and αT is outside the grid, we use one dimensional interpolation. Otherwise if both these values are outside the grid we use two dimensional interpolation.

2.3.2 Method 2

We notice that method 1 of creating data set has six inputs which increases the difficulty of neural networks training. We want to reduce the dimension of input. Therefore, we can use 4 dimensional inputs for training the neural networks and use scaling relationship after the training and get the price of American put option for any six parameters.
From equation (2.7), fix $K = \tilde{K}$ and $\sigma = \tilde{\sigma}$, if we can use neural networks to predict the value of any other 4 parameters , we then get the value of the left hand side. Using this equation we can get the right hand side for any six parameters. The mathematical idea is below.
Suppose we want to calculate $P(s, k, t, r_0, e, g)$. Then we have,

$$\begin{cases} \lambda S_0 = s \\ \lambda \tilde{K} = k \\ \alpha T = t \\ \mathrm{r}/\alpha = r_0 \\ \lambda^{1-\gamma}\alpha^{-\frac{1}{2}}\tilde{\sigma} = e \\ \gamma = g \ . \end{cases}$$

Solving these equations, we can get

$$\begin{cases} \mathrm{S}_0 = \frac{a\tilde{K}}{b} \\ \alpha = \frac{(\frac{b}{\tilde{K}})^{1-g}\tilde{\sigma}}{e} \\ \mathrm{r}{=}\mathrm{r}_0\alpha \\ \mathrm{T}{=}\mathrm{t}/\alpha \\ \gamma = g \ , \end{cases}$$

where $\tilde{K}$,$\tilde{\sigma}$ s, k, t, r_0, e, g are known.
Plugging S_0, T, r, γ in the left hand side of (2.7) and let the neural network predict this value, we then get the right hand side $P(s, k, t, r_0, e, g)$ for any

s, k, t, r_0, e, g. Here are the steps.
1.Fix $K = \tilde{K}$ and $\sigma = \tilde{\sigma}$, for each r and γ, use a finite difference scheme to calculate the price of different t and s and get $P(s, t, r_0, g)$ for different s, t, r_0, g.
2.Use data set from 1 for training neural networks.
3.Use the scaling relationship above to get $P(s, k, t, r_0, e, g)$.
This methods best advantage is that it reduces the input of neural networks from 6 dimensions to 4 dimensions. In 6 dimensional case, if we use 10 different values of each parameter, we then have 10^6 samples, while the same 1 million samples worth $\sqrt[4]{10^6} \approx 32$ for each parameter in 4 dimensional case. However, this methods use two loops for creating data set. For each r and γ, we should run a finite difference scheme, which require much more time when creating the data set.

2.4 Creating dataset

2.4.1 Supplement of special samples

In both method 1 and method 2, we need to select samples. For example, in method 1 step 3, when I mention change any reasonable s,k,t,r_0, I actually use the Cartesian product to generate samples. Cartesian product ensures the samples can uniformly cover each scenario of s,k,t,r_0 which is good for the neural networks to learn different scenarios. However, in my experiment, if we use this data set for training, the neural networks can only perform well when stock price S is far from strike K, which is meaningless because one can easily estimate the price will be approximately $(K - S)^+$. In order to fix it, we can add some specific samples where S is close to K.

2.4.2 Details about creating dataset

Finally I choose method 2 for creating the dataset because according to my experiment, it is more accurate. Here's how to build this dataset.

- Fix $K = \tilde{K} = 1$, $\sigma = \tilde{\sigma} = 0.15$(I choose this because when we use scaling relationship to the four set of parameters in the table (2.5.3). The 4 paramters S_0, r, T, γ we should put into the neural network are in a proper range.)

- Uniformly choose 21 values between [0,0.15] and assign these values to r. i.e. $r_0 = 0, r_1 = 0.0075, ...r_{20} = 0.15$. Uniformly choose 21 values between [0,1] and assign these values to γ. i.e. $\gamma_0 = 0, \gamma_1 = 0.05, ...\gamma_{20} = 1$. For each r_i (i=0,1,...20), for each γ_j (j=0,1,...20), **run a Crank – Nicolson scheme** ($S_{max} = 3$, T=2, S steps N=2000, t steps M=300, I choose $S_{max} = 3$ because according to (2.2), when I plug K=1 and $\sigma = 0.15$, I get $S_{max} \approx 3$)

- Uniformly choose 40 values between [0,2] for T. Uniformly choose 60 values between [0.7,1.3] for S. At [0,0.7] and [0.7,3], an additional 20 values are uniformly taken for S respectively.

- When the calculated American option price of a sample is below 1×10^{-5}, the sample is deleted.(I do this because when the option price is below 1×10^{-5}, it's worthless in real financial market and I don't want my neural network to learn too many examples like this)

In total, we have 1,412,417 samples for our dataset.

2.4.3 Scale the data

Scaling data is an important step before training neural networks. In this case, if we don't scale the data and use the theoretical value calculated by FDM for training. There will be many samples with value that is extremely small while also have many values that are relatively large. For example, if we look at the data set, the last column can have many values like 10^{-4} as well as values like 0.7. If we pass these values for training, it will confuse the neural networks. For example, if we use absolute loss function, and get the prediction for these two values like 0.0003 and 0.65. The percentage error is 200% and 7.14% while absolute error is 0.0002 and 0.05, but the neural networks will think the first prediction is better which will have a great negative impact on the accuracy of neural networks. In my experiment, I scale the data to the range of [-1,1]. Here are mathematical statement:

$$\tilde{x}^i = \frac{2x^i - (x^i_{max} - x^i_{min})}{(x^i_{max} - x^i_{min})} \ ,$$

where x^i means the i-th column of the original data set.

2.5 Neural networks training and results

2.5.1 Select the architecture of neural networks

After creating the dataset, we need to select the architecture of neural networks. Universal approximation theory in 1.4.3 (a) tells us that neural networks with one hidden layer can approximate any continuous function to any accuracy. However, this theory doesn't tell us how many notes we need for a certain accuracy, nor tell us how the approximation function and original function looks like. Unfortunately, there is no established theory that tells us how many layers and how many notes for each layer to pick, but in the book of Jeff Heaton[11], there are some empirically-derived rules-of-thumb, of these, the most commonly relied on is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'
Another way to choose the architecture is by using a neural network structure that has previously performed well for similar problems. In the paper 'Deep learning volatility'[1], Blanka Horvath used 4 hidden layers and 30 notes each layer for deep learning volatility. This structure can be a good starting point for deep pricing in the CEV model.
After several attempts, I choose to use 3 hidden layers with 30 nodes in each hidden layer. The schematic diagram of the neural network is shown in Figure 2.6. According to 1.4.3 (a), the smoothness of activation can impact the smoothness of neural network. To this reason, I choose the activation function to be **Elu** function which is defined as $\sigma_{Elu}(x) = \alpha(e^x - 1)$ because it's continuously differentiable.

2.5.2 Choose the loss function and update parameters

I choose the root mean square error loss function which is defined as follows:

$$l(\hat{y}_i, y_i) = \sqrt{(\hat{y}_i - y_i)^2} \ . \tag{2.9}$$

Optimise parameters in the neural network is by **adam optimizer**[10]. It is an extension to stochastic gradient descent mentioned in 1.4.4. I chose 100 epochs which means run the dataset 100 times and I used 10 percent of the dataset as the test set and 90 percent as the training set.

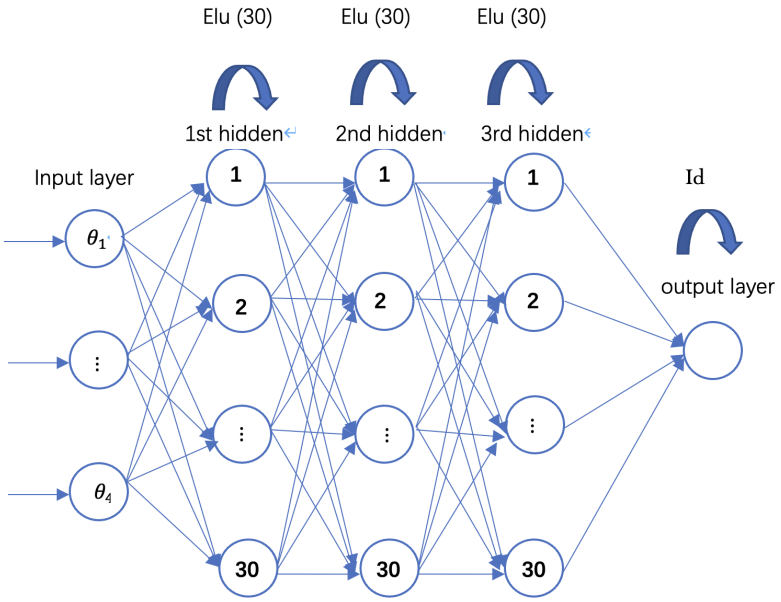


Figure 2.6: The neural network for CEV model

2.5.3 Result and test of accuracy

After training the neural network, we need to check if it is accurate enough. First we introduce **time value** to be the difference between the price of the option and the immediate exercise value. Setting $r = 0.02$, $\sigma = 0.1$, $T = 1$, and varying S0 between 0.8 and 1.5, we can plot two pictures showing the time value for finite difference method and time value for our neural network. Each picture contains 3 lines one for the case $\gamma = 0$, one for the case $\gamma = 0.5$ and one for the case $\gamma = 1$. **The pictures are shown in Figure 2.7 and in Figure 2.8.** Figure 2.7 gives us time value for FDM, and Figure 2.8 gives us time value for neural network. As shown from these two pictures, both figures demonstrate the characteristic that time value rises first and then falls as S0 increases. In addition, the "Time value" of neural network is slightly different from that of FDM when S0 is small, but the two are very close for other S0. This result preliminarily shows that our neural network is accurate.
The **table2.5.3** shows the prices of American put options for four different sets of parameters, where FD represents the price of American put option calculated by finite difference method and NN is the price of American put option calculated by our neural network. $e\%$ is relative percentage error

defined by:

$$e = |\frac{NN - FD}{FD} \times 100| \tag{2.10}$$

As shown from the table, e is below 1% except for the fourth value. In financial market, options have **bid price**(the price of selling) P_b and **ask price**(the price of buying) P_a. Using Apple put options data ("Source: Bloomberg Finance L.P."), $\frac{|P_a - P_b|}{((P_a + P_b)/2)}$ is around 3% to 5%. Therefore, we can assume that if e is lower than 3%, then the neural network has good accuracy.

S0	K	T	r	σ	γ	FD	NN	e%
1.1	1	1	0.05	0.1	0.9	0.003675	0.00370	0.68
1	1	1	0.05	0.1	0.9	0.024044	0.024028	0.06
0.9	1	0.5	0.02	0.1	0.5	0.09999	0.10006	0.07
2.6	2	1	0.1	0.2	0.8	0.002431	0.002477	1.89

Showing effectiveness

In the above process of calculating "time value", only S changes within a certain range(between 0.8 and 1.5). I hope that T can also vary within a certain range, so I use the following method to show the accuracy of the neural network.

- Set $r = 0.02$,$\sigma = 0.1, \gamma = 0.5$.
- For S0 varying from 0.8 to 1.6 and T varying from 0.1 to 2, If $|NN - FD| < 0.001$, then $e = |NN - FD| \times 100$, else, $e = |\frac{NN-FD}{FD} \times 100|$.
- Plot e for different S0 and T.
- Change $\gamma = 1$ and do the above steps again.

The second step 'If $|NN - FD| < 0.001$, then $e = |NN - FD| \times 100$' is necessary. When $|NN - FD| < 0.001$, the neural network already shows some good accuracy for this point, and if we blindly use relative percentage error, when FD is very close to 0, e tends to infinity.

Following the steps above, **The pictures are shown in Figure 2.9 and in Figure 2.10**. As shown from these pictures, generally, the neural network performs well, with error around 1%. However, for points with $T \in [1, 1.9]$ and $S0 \in [0.9, 1.16]$, the neural networks performs slightly inaccurate, with e between 5% and 8%.

Recall from 2.3.2, if we want to calculate $P(s, k, t, r_0, e, g)$, we use the scaling relationship to get $S0, T, r, \gamma$ which we should put in neural network for

prediction. If the calculated $S0 \in [0.9, 1.16]$ and $T \in [1, 1.9]$, we can expect the prediction is slightly inaccurate with e between 5% and 8%, otherwise, it is accurate with error between 1% and 2%.

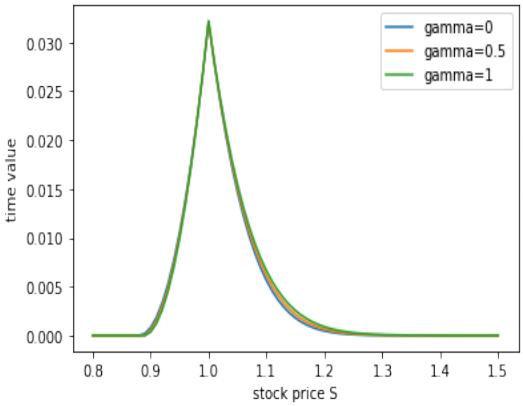


Figure 2.7: time value of FDM

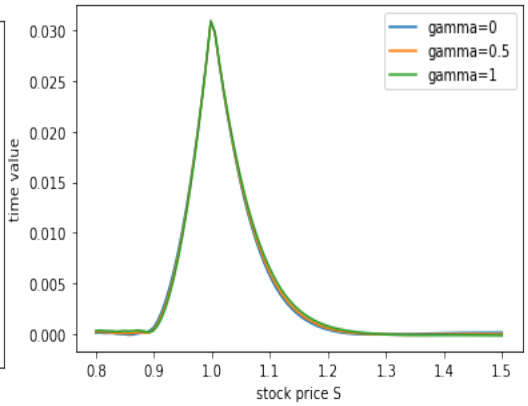


Figure 2.8: time value of neural network

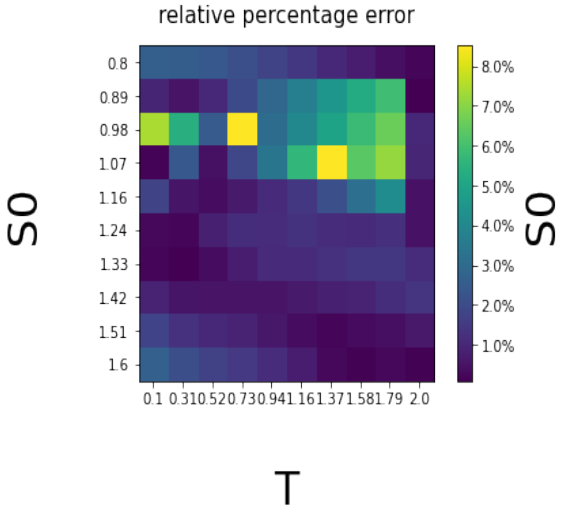


Figure 2.9: $\gamma = 0.5$

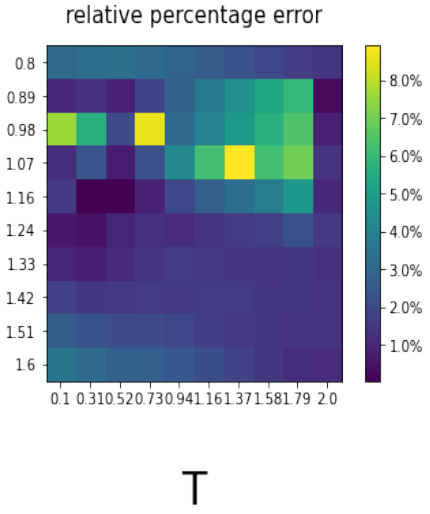


Figure 2.10: $\gamma = 1$

Chapter 3

Personal Contribution

This part is organized as follows.
First, I used Bloomberg data to compare the calibration effect of European put option under CEV model, jump diffusion model, and Heston model.
Second, I did deep calibration under Heston model as an example of how to do deep calibration.
At last, I used a different machine learning method "XGBoost" to pricing American put option under CEV model and compare this result to chapter2 and used American put option data from Bloomberg to do calibration under this new method.

3.1 Calibration and Deep calibration

3.1.1 Traditional calibration and its bottleneck

For a stochastic model $M(\theta)$(θ is a vector containing all the parameters we need to calibrate in the stochastic model. If $M(\theta)$ is Black and Scholes model, θ is only one dimensional and equal to σ), denote the theoretical option price under θ for strike K and maturity T is $P^{M(\theta)}(K,T)$ and the market price of the same type option is $P^{Mk}(K,T)$. Calibration intends to find θ which makes $P^{M(\theta)}(K,T)$ and $P^{Mk}(K,T)$ as close as possible for different K,T. In mathematical words, find θ which satisfies:

$$\theta = argmin_\theta \sum_{i=1}^{m}\sum_{j=1}^{n} \omega_{i,j}\delta(P^{M(\theta)}(K_i,T_j) - P^{Mk}(K_i,T_j)) \ , \qquad (3.1)$$

where $\omega_{i,j}$ is the weight for the option with strike K_i and time to maturity T_j, $\delta(a,b)$ measures the distance between a and b. For any function related of θ, denoted by $f(\theta)$, $argmin_\theta(f(\theta))$ means the θ which minimize $f(\theta)$. For the sake of simplification, in this part I choose $\omega_{i,j} = \frac{1}{n \times m}$ and δ to be the square of two-norm. For $a \in R$ and $b \in R$, it is defined by:

$$\delta(a,b) = (a-b)^2, \tag{3.2}$$

where R means real number field.
In many academic paper, option calibration is often achieved via implied volatility. Implied volatility is an very important concept in financial mathematics, it refers to a metric that captures the market's view of the likelihood of changes in a given security's price[7]. In mathematical words, market implied volatility is the root of the following equation:

$$P(S,K,T,r,IV^{Mk}(K,T)) = P^{Mk}(K,T) \ , \tag{3.3}$$

where $P^{Mk}(K,T)$ is the market price for the option with strike K and maturity T, and $P(S,K,T,r,\sigma)$ is the Black-Scholes formula for European option, in the case of call option it is given by:

$$P(S,K,T,r,\sigma) = SN(d1) + e^{-rt}KN(d2) \ , \tag{3.4}$$

where d1,d2 is

$$d1 = \frac{log\frac{S}{K} + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}} \ , \ d2 = d1 - \sigma\sqrt{T} \ .$$

Very similar, model implied volatility is the root of the following equation:

$$P(S,K,T,r,IV^{M(\theta)}(K,T)) = P^{M(\theta)}(K,T) \ . \tag{3.5}$$

In equation (3.3) , all the parameters $S,K,T,r,P^{Mk}(K,T)$ are known from the market except $IV^{Mk}(K,T)$ and the value $IV^{Mk}(K,T)$ which makes equation (3.3) holds is the market implied volatility.
In equation (3.5), $P^{M(\theta)}(K,T)$ is the theoretical price. In many stochastic models, there is no analytic formula for the theoretical price and this price is often calculated via numerical methods like Monte-Carlo simulation or finite difference methods. During the calibration process, each time the θ is updated, the numerical methods are used in the whole T,K surface. If the numerical methods need 2 seconds, and we need to calibrate on 84 options. The time for calculating all the options theoretical price is 168s. If the descent method for optimize (3.1) is more than 100 steps, then the whole time

for calibration is more than $1.68 \times 10^5 s$. This is far more time than can be tolerated. In order to solve this problem, we need a fast way to calculate theoretical price. One way is to choose a stochastic model which has an analytic formula or closed formula like Black Scholes formula. The other way is to put the approximation of theoretical price process in neural network, and once the parameters in neural network are settled, neural networks can quickly map θ to approximated prices $\tilde{P}^{M(\theta)}(K_i, T_i)$.

3.1.2 Deep calibration

In simple words, deep calibration means using neural networks as the option price approximation function and use this function in calibration. Hernandez[8] is one of the pioneer of deep calibration and he use neural networks to calibrate directly. Instead of using his method, I use the more practical and intuitive two-steps deep calibration way in Horvath et al[1].
In the first step, use neural network to learn the pricing map $\theta \to P^{M(\theta)}(K_i, T_j)$ and get $\tilde{P}^{M(\theta)}(K_i, T_j)$.
In the second step, do the optimisation

$$\theta = argmin_\theta \sum_{i=1}^{m} \sum_{j=1}^{n} \omega_{i,j} \delta(\tilde{P}^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j)) \ , \qquad (3.6)$$

Note : $\tilde{P}^{M(\theta)}(K_i, T_j)$ is the neural network calculated price under θ for options with strike K_i and maturity T_j. As shown in Figure 2, the function is fixed when all the parameters in linear transformations $L_1, L_2, ...L_r$ are fixed.

3.2 Calibrated models

3.2.1 CEV model

As shown in 1.15, in risk neutral measure Q, the stock price satisfies:

$$dS_t = rS_t dt + \sigma S_t^\gamma dW_t^Q \ ,$$

where r is the risk-free rate, σ is the volatility, $\gamma \in [0, 1]$ and W_t^Q is the Brownian motion in measure Q. Using the properties of CEV in 1.2.3, the closed form formula for the European put option price is:

$$P = Ke^{-r(T-t)}[1 - \chi^2(c, b, a)] - S_t\chi^2(a, b + 2, c) \ ,$$

where

$$a = \frac{[Ke^{-r(T-t)}]^{2(1-\gamma)}}{(1-\gamma)^2 v}, b = \frac{1}{1-\gamma}, c = \frac{S^{2(1-\gamma)}}{(1-\gamma)^2 v} \ ,$$

with

$$v = \frac{\sigma^2}{2r(\gamma-1)}[e^{2r(\gamma-1)(T-t)} - 1] \ .$$

3.2.2 Jump diffusion model

In jump diffusion model, the stock price at time t is assumed as

$$S_t = j_1^{N(t)}\tilde{S}_t \ ,$$

where $\tilde{S}_t$ follows an geometric Brownian motion,

$$d\tilde{S}_t = \tilde{S}_t(\tilde{\mu}dt + \sigma dW_t) \ ,$$

where j_1 is the jump size and $j_1 \in (0, 1)$, N(t) is the the total number of jumps up to time t. All jumps are of a fixed proportion of the stock price which means j_1 is a constant. N(t) is a random variable assumed to follow the Poisson distribution with parameter λt. In Q measure,

$$Q(N(t) = n) = \frac{(\lambda t)^n}{n!}e^{-\lambda t} \ .$$

In John's lecture notes [9], it gives the European call option price under this model. which is,

$$C(K, T, j_1, \lambda, \sigma) = \sum_{n=0}^{\infty} \frac{(j_1\lambda T)^n}{n!}e^{-j_1\lambda T}V(S_0, j_1^{-n}K, T, \tilde{\mu}, \sigma) \ .$$

where $V(S_0, j_1^{-n}K, T, \tilde{\mu}, \sigma)$ is the Black-Scholes call option price shown in (3.4) and $\tilde{\mu} = r + \lambda(1 - j_1)$.

3.2.3 Heston model

Heston model is introduced by Steven.Heston in 1993[22], in its assumption, the underlying volatility follows a stochastic process, and it allows arbitrary correlation between volatility and spot-asset return. In mathematical word,

$$dS_t = \mu S_t dt + \sqrt{v_t} S_t d(W_1)_t \ ,$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} d(W_2)_t \ ,$$

$$d(W_1)_t d(W_2)_t = \rho dt \ .$$

v_t is strictly positive if :

$$2\kappa\theta > \sigma^2 \ .$$

Under these assumptions, the price for European call option is:

$$C(S,t) = S_t P_1 - K e^{-r(T-t)} P_2 \ .$$

Using the property of put-call parity in 1.3.5, the price for European put option is:

$$P(S,t) = C(S,t) + K e^{-rT} - S_0 \ ,$$

where r is the interest rate, P_1 and P_2 are risk-neutral probabilities obtained by characteristic function f_j :

$$P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty Re[\frac{e^{-i\phi ln(K)} f_j}{i\phi}] d\phi \ ,$$

where

$$f_j = \exp(C_j + D_j v_t + i\phi ln S_t) \ ,$$

$$C_j = (r-q)\phi i(T-t) + \frac{\kappa\theta}{\sigma^2}[(b_j - \rho\sigma\phi i + d_j)(T-t) - 2ln(\frac{1-g_j e^{d_j(T-t)}}{1-g_j})] \ ,$$

$$D_j = \frac{b_j - \rho\sigma\phi i + d_j}{\sigma^2}(\frac{1-e^{dj(T-t)}}{1-g_j e^{d_j(T-t)}}) \ ,$$

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

$$d_j = \sqrt{(\rho\sigma\phi i)^2 - \sigma^2(2u_j\phi_i - \phi^2)} \ ,$$

where i is the imaginary unit and $u_1 = \frac{1}{2}, u_2 = -\frac{1}{2}, b_1 = \kappa + \lambda - \rho\sigma, b2 = \kappa + \lambda$, λ refers to the market price of volatility risk. In risk neutral world, $\lambda = 0$.

3.3 Monte-Carlo simulation for Heston

Consider the value calculated by the formulas above is theoretical value. Using Monte-Carlo simulation for Heston model has two purposes. One is to testify if the closed formula of Heston model is correct, and the other is to later compare whether the error between the neural network and the theoretical value is within the error between the Monte Carlo and the theoretical value.

3.3.1 Simulated two correlated Brownian motion

Heston model in 3.2.3 has two correlated Brownian motions, to simulate two correlated Brownian motions $(W_1)_t$ and $(W_2)_t$ with covariance matrix $\begin{bmatrix} t & \rho t \\ \rho t & t \end{bmatrix}$, we need to use **Cholesky decomposition**. [9]. Suppose we have n time steps and M scenarios, then the following steps show how to generate $(W_1)_t$ and $(W_2)_t$.
1. Generate a matrix of 2 rows and M columns containing independent standard normally distributed random numbers called N^i, where M is the number of scenarios we want to simulate.
2. Use Cholesky decomposition to matrix$\sum$ to get matrix L, where the matrix$\sum$ is just $\begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$without t, and L is the lower triangular matrix obtained from Cholesky decomposition.
3. $E^i = L \times N^i$.
4. Denote $\delta t = \frac{T}{n}$, then $(W_1)^k_{(i+1)\delta t} = (W_1)^k_{i\delta t} + \sqrt{\delta_t} E^i_{1k}$, $(W_2)^k_{(i+1)\delta t} = (W_2)^k_{i\delta t} + \sqrt{\delta_t} E^i_{2k}$ $(k = 1, 2...M; i = 0, 1...(n-1))$,
where $(W_1)^k_{(i+1)\delta t}$ means the value of W_1 at time point$(i+1)\delta t$ in the k-th scenario. E^i_{1k} represents the element in the first row, the k-th column of the matrix E^i.

3.3.2 Euler scheme for simulation

After simulating two correlated Brownian motion, we then use the following equations for simulating stock paths under Heston model:

$$v^k_{t+(i+1)\delta_t} = v^k_{t+i\delta_t} + \kappa(\theta - v_{t+i\delta_t})\delta_t + \sigma\sqrt{v_{t+i\delta_t}}((W_2)^k_{t+(i+1)\delta_t} - (W_2)^k_{t+i\delta_t}) \ ,$$

$$S^k_{t+(i+1)\delta t} = S^k_{t+i\delta t} + (rS^k_{t+i\delta t})\delta t + \sqrt{v_{t+i\delta t}}S^k_{t+i\delta t}((W_1)^k_{t+(i+1)\delta_t} - (W_1)^k_{t+i\delta_t}) \ , (k = 1, 2...M).$$

The initial value $S_t = S$. Similar to equation (1.18), the European put option price by Monte-Carlo simulation is :

$$V(S,t) = e^{-r(T-t)}\frac{((K - S^1_T)^+ + (K - S^2_T)^+ ... + (K - S^M_T)^+)}{M} \ . \qquad (3.7)$$

3.4 Data statement

In **3.5**, I use European put options with different strike K and time to maturity T for calibration, and the underlying is chosen to be Apple stock because Apple is one of the largest company in the world, the data set is in appendix called "APPL.xlsx". I didn't use all types of put options available from the market, but choose put options with strike prices around the stock price S0, because such options are the most heavily traded and most accurately priced. Therefore, the whole calibration process in this part is in a fixed grid of strike price K and time to maturity T, with

$$K = 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190 \ ;$$

$$T = 0.22, 0.3, 0.37, 0.47, 0.55, 0.62, 0.87 \ .$$

The implied volatility surface in this data is shown in Figure 3.5.1 where we can see the implied volatility decreases as strike price increase.
The dataset I used in **3.6** and is very similar to the dataset I used in **3.5**. However, there are two differences from the previous dataset. One difference is that I use American put option. The other difference is that it has the following K,T values:

$$K = 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200 \ ;$$

$$T = 0.16, 0.23, 0.31, 0.41, 0.48, 0.56 \ .$$

All data in this paper are from "Bloomberg Finance L.P."

3.5 Calibration for European put option

3.5.1 Measure the effectiveness of calibration

For different models, after optimising equation (3.1) and get $\theta = \theta^*$, we need to measure the effectiveness of calibration under θ^*. For calibration via price define the relative price percentage error as:

$$e^p_{i,j} = |\frac{P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j)}{P^{Mk}(K_i, T_j)}| \ . \tag{3.8}$$

For calibration via implied volatility it is defined as:

$$e^{iv}_{i,j} = |\frac{IV^{M(\theta)}(K_i, T_j) - IV^{Mk}(K_i, T_j)}{IV^{Mk}(K_i, T_j)}| \ . \tag{3.9}$$

3.5.2 CEV calibration

Calibration in CEV model intends to find $\theta = (\sigma, \gamma)^T$($(...)^T$ is the transpose of a vector) which satisfies:

$$\theta = argmin_\theta \sum_{i=1}^{12} \sum_{j=1}^{7} (P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2 \ .$$

Using the gradient based optimisation method **'SLSQP'**[9], θ_i is updated by:

$$\theta_i := \theta_{i-1} - \eta \bigtriangledown_\theta (\sum_{i=1}^{12} \sum_{j=1}^{7} (P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2)|\theta = \theta_{i-1} \ ,$$

where $P^{M(\theta)}(K_i, T_j)$ is given in 3.2.1. To test the closed form formula, I use Monte-Carlo simulation following the steps in 3.2.1. Here are the results:

S0=1 K=1,T=1,r=0.05,$\sigma = 0.1, \gamma = 0.9$	Value
corresponding 95% confidence interval by Monte-Carlo	(0.01920,0.02073)
European put option price by closed formula	0.01928

The closed formula price is within the Monte-Carlo interval. Based on equation(3.8), the calibration effectiveness of CEV model is shown in **Figure3.2**. As shown in this picture, the effectiveness of calibration becomes worse as K decrease, and when K=135, the relative error is around 40%. Generally, most relative errors are between 5% to 10%.
However, this may not because the model itself is bad. Options exchanges do not exactly price at the theoretical value derived from risk-neutral pricing; they price slightly higher than the theoretical price in order to make a profit. When the option price is high, this extra charge is relatively small, but in this case when S0=159.735 and strike price is much lower than S0, the theoretical price of put option is very small, and therefore amplifies the impact of the extra charge.

3.5.3 jump diffusion calibration

Calibration in jump diffusion model intends to find $\theta = (j_1, \lambda, \sigma)^T$ which satisfies:

$$\theta = argmin_\theta \sum_{i=1}^{12} \sum_{j=1}^{7} (P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2 \ .$$

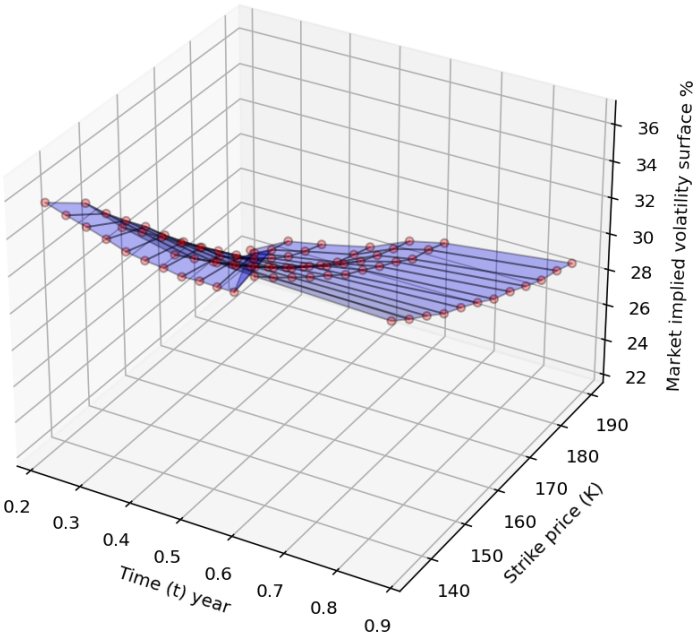
For jump diffusion model, I calibrate it via implied volatility, because in my experiment, the equation (3.5) will not give us infinity value under jump diffusion model. Calibration via implied volatility means finding θ that satisfies:

$$\theta = argmin_\theta \sum_{i=1}^{12} \sum_{j=1}^{7} (IV^{M(\theta)}(K_i, T_j) - IV^{Mk}(K_i, T_j))^2 \ .$$

Using the gradient based optimisation method **'SLSQP'**[9], θ_i is updated by:

$$\theta_i := \theta_{i-1} - \eta \bigtriangledown_\theta (\sum_{i=1}^{12} \sum_{j=1}^{7} (IV^{M(\theta)}(K_i, T_j) - IV^{Mk}(K_i, T_j))^2 | \theta = \theta_{i-1} \ ,$$

where $IV^{Mk}(K_i, T_j)$ is model implied volatility and $IV^{M(\theta)}(K_i, T_j)$ is market implied volatility and it is the root of the following equations:



1

Figure 3.1: implied volatility surface

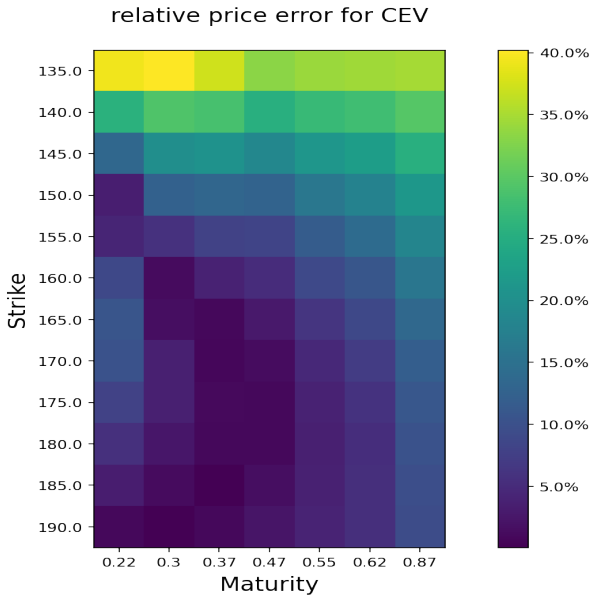


Figure 3.2: error for cev

$$P^{M(\theta)}(K_i, T_j) = P(K_i, T_j, j_1, \lambda, \sigma) \ ,$$

$$P(K_i, T_j, j_1, \lambda, \sigma) = \sum_{n=0}^{\infty} \frac{(j_1 \lambda T)^n}{n!} e^{-j_1 \lambda T} V(S_0, j_1^{-n} K_i, T_j, \tilde{\mu}, \sigma) \ ,$$

$$V(S, K_i, T_j, r, IV^{M(\theta)}(K_i, T_j)) = P^{M(\theta)}(K_i, T_j) \ ,$$

where $V(S_0, j_1^{-n} K, T, \tilde{\mu}, \sigma)$ is the Black-Scholes put option price shown above and $\tilde{\mu} = r + \lambda(1 - j_1)$.
Based on equation(3.9), the calibration effectiveness of CEV model is shown in **Figure3.3**.
As shown in this picture, except for the poor calibration results of some options near the edge of the K, T grid, the other points are all well calibrated, and most of the relative percentage errors are between 2% and 4%. The calibration effect for jump diffusion model is better than that in CEV model for Apple European put options.

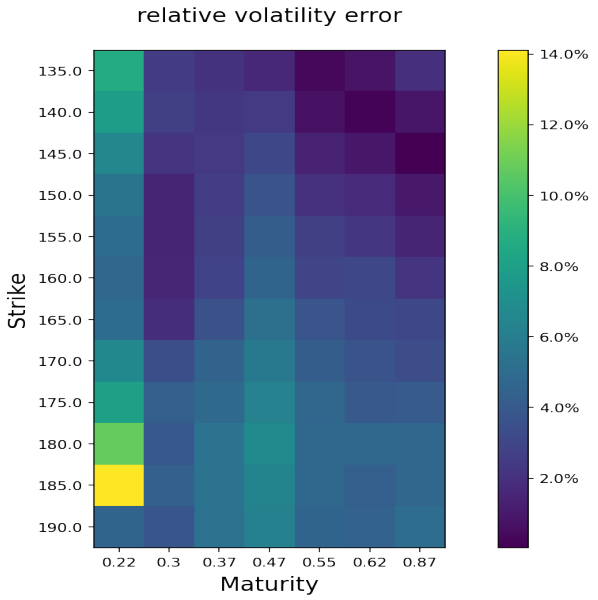


Figure 3.3: error for jump-diffusion

3.5.4 Heston calibration

For Heston model in measure Q, $\lambda = 0$. According to the formulas in 3.2.3, $\theta = (v_0, \kappa, \theta, \sigma, \rho)^T$. Similar to the previous steps, we are aiming to find θ which satisfies:

$$\theta = argmin_\theta \sum_{i=1}^{12} \sum_{j=1}^{7} (P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2 \ .$$

Using the gradient based optimisation method **'SLSQP'**[9], θ_i is updated by:

$$\theta_i := \theta_{i-1} - \eta \bigtriangledown_\theta (\sum_{i=1}^{12} \sum_{j=1}^{7} (P^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2) | \theta = \theta_{i-1}$$

Where $P^{M(\theta)}(K_i, T_i)$ is given in 3.31. Here, I choose to calibrate price directly rather than calibrate via implied volatility because the numerical way to solve this equation

$$V(S, K_i, T_i, r, IV^{M(\theta)}(K_i, T_j)) = P^{M(\theta)}(K_i, T_j)$$

"secant" may be unstable when the market price is small and give us infinity value. To ensure the price function $P^{M(\theta)}(K_i, T_j)$ given in 3.33 is writing correctly I use Monte-Carlo simulation under Heston model to test it. The details about Monte-Carlo simulation for Heston model is given in 3.3. The Monte-Carlo price function for Heston model is named price_m in appendix. Here is the result:

S0=60 K=50,T=1,r=0.04,v0=0.1,$\kappa = 2, \theta = 0.04, \sigma = 0.01, \rho = -0.7$	Value
corresponding 95% confidence interval by Monte-Carlo simulation	(1.4439,1.4880)
European put option price by closed formula	1.477

The closed formula price is within the Monte-Carlo 95% confidence interval. Based on equation(3.8), the calibration effect of Heston model is shown in **Figure3.4**. As shown in the picture, the effect of calibration is improved by Heston model, with most of the relative percentage errors between 1% and 2%. This means Heston model might better solve the problem of volatility smile. However, the reason that the model works well may not be because the stochastic process model is more market-friendly but simply because it has more choice of parameters. In CEV model, we only have 2 parameters r, γ to calibrate. In jump-diffusion model, we have 3 while in Heston model, we have 5. The more parameters we have, the more likely to perform better. However, we have to consider the computational problems and overfitting problems that arise when the model is too complex.

Figure 3.4: error for Heston

3.6 Deep calibration for Heston

3.6.1 Image based training

In the neural network of Part2, the input layer is four-dimensional and the output layer is one-dimensional. In the calibration of Heston model, it is necessary to calculate the theoretical price of the whole K, T grids and make it approximate to the market price through parameter adjustment. If we use the same way as Part 2, putting all the parameters $K, T, V0, \kappa, \theta, \sigma, \rho$ in the input layer and only putting theoretical price in output layer, it can cause both problems in neural networks accuracy and training time, because if we choose 10 points for each parameter, it will have 8.4×10^6 samples which is difficult for training. Also, this method does not fully exploit the structure of prices on the adjacent K,T grid because it treats K,T as isolated variables in the input layer.
To fix this problem, inspired by Horvath et al[1], a natural idea is to set the output layer to be the theoretical value of the option prices on all fixed K,T grids. In other words, K and T are no longer parameters in input layers and output layer has 84 dimensions which can be regarded as a image with fixed pixels.

3.6.2 Creating dataset

Creating dataset is the crucial step for training neural networks and the quality of dataset has great impact on neural networks's accuracy. In general, it is difficult to find the potential value range of each parameter and if the range is too wide, it will decrease the accuracy of neural network. However, in previous work 3.5.4, we already get the calibrated parameters $\theta_* = (v_*, \kappa_*, \sigma_*, v0_*, \rho_*)^T$, therefore, it makes sense to set the range of θ including θ_* because θ_* fit the market data well, and in later deep calibration part I want to cover this value.
In my experiment, I set the range of θ to be $(\frac{\theta_*}{2}, \frac{3\theta_*}{2})^T$. For each sample, I use a random way to pick parameters in the previous range. In python, numpy.random.rand() gives us a random number between (0,1). Take κ as an example, if we want to randomly select a κ between $(\frac{\kappa_*}{2}, \frac{3\kappa_*}{2})^T$. What we need to do is using numpy.random.rand() to generate a random number between (0,1) and use this number to times a constant a and plus a constant b. a and b follows the following equation:

$$a \times 0 + b = \frac{\kappa_*}{2}, a \times 1 + b = \frac{3\kappa_*}{2}$$

Solving the equations, we have $a = \kappa_*, b = \frac{\kappa_*}{2}$. For other parameters $\theta, \sigma, v0, \rho$, they follow the same way and they are independent of each other. The theoretical price under sample is calculated by closed function mentioned in 3.2.3. In my experiment, I generate 10000 samples for training.(The generation time of the dataset was very long, requiring more than four hours. I saved the resulting dataset in a txt file called 'Hestondata')

3.6.3 Select the architecture and training

After several attempts, I choose to use 3 hidden layers with 30 nodes in each hidden layer. The schematic diagram of the neural network is shown in Figure3.5. According to 1.4.3(a), the smoothness of activation can impact the smoothness of neural network. To this reason, I choose the activation function to be Elu function which is defined as $\sigma_{Elu}(x) = \alpha(e^x - 1)$ because it's continuously differentiable.
Input layer is 4 dimensional and output layer is 84 dimensional. The loss function is defined as:

$$l(\hat{\mathbf{y}}, \mathbf{y}) = \sqrt{\frac{1}{O}\sum_{j=1}^{O}(\hat{\mathbf{y}}_j - \mathbf{y}_j)^2} \ ,$$

where O is the dimension of output layer. In this case, $O = 84$.
Before training, we have to scale the data. The reason for scaling is mentioned in 2.4.3. In my experiment, I scale the data to the range of [-1,1]. Here are mathematical statement.

$$\tilde{x}^i = \frac{2x^i - (x^i_{max} - x^i_{min})}{(x^i_{max} - x^i_{min})}$$

where x^i means the i-th column of the original data set.
In training process, I choose 100 epochs, which means run the whole data set for 100 times and I use 10% of the whole dataset as test set. Because of the image-based training, the training process is significantly faster than Part2(only takes 2 minutes).

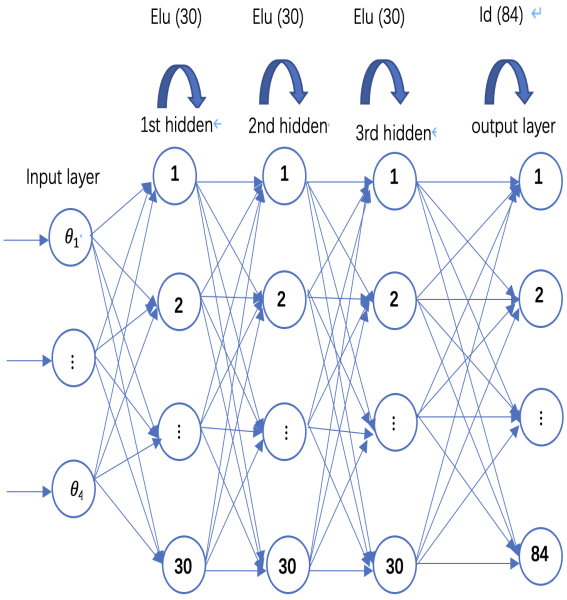


Figure 3.5: the neural network for Heston

3.6.4 Deep calibration under Heston model

After the training of neural network, we get the map $\theta \to \tilde{P}^{M(\theta)}(K_i, T_i)$
In the next step, do the optimisation:

$$\theta = argmin_\theta \sum_{i=1}^{12} \sum_{j=1}^{7} (\tilde{P}^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j))^2 \ ,$$

Using the gradient based optimisation method **'SLSQP'**[9], θ_i is updated by:

$$\theta_i = \theta_{i-1} - \eta \bigtriangledown_\theta (\sum_{i=1}^{12}\sum_{j=1}^{7}(\tilde{P}^{M(\theta)}(K_i,T_j) - P^{Mk}(K_i,T_j))^2)|\theta = \theta_{i-1} \quad (3.10)$$

3.6.5 Result

After training the neural network, we can get the map: $\theta \to \tilde{P}^{M(\theta)}(K_i,T_i)$, and using (3.10) we can get the optimum $\theta = \theta^{**}$. To see the accuracy of neural network, we define relative percentage price error between neural network's prediction and closed form Heston pricing formula as:

$$e^1_{i,j} = |\frac{\tilde{P}^{M(\theta)}(K_i,T_j) - P^{M(\theta)}(K_i,T_j)}{P^{M(\theta)}(K_i,T_j)}| \quad (3.11)$$

To see the effectiveness of deep calibration, use the equation(3.8) , fix $\theta = \theta^{**}$ and calculate $e^p_{i,j}$ for i=1,2...12 and j=1,2...7.
The neural network's effectiveness is shown in **Figure3.6**. As shown from the picture, the maximum error is 2% and most of errors are between 0.5% and 1%, which shows our neural network can approximate theoretical price $P^{M(\theta)}(K_i,T_j)$ very well.
The deep calibration effectiveness is shown in **Figure3.7** . As shown in this picture, except for the poor calibration results of some options near the left and top boundary of the K, T grid, the other points are well calibrated, and most of the relative percentage errors are between 5% and 10%.

3.7 XGBoost way for pricing and calibration for American put option under CEV model

3.7.1 Motivation

The problem of approximating theoretical price can be generalized as regression problem where we have input and output pairs. Supervised learning algorithms are particularly effective for dealing with regression problems. Neural networks perform well as one of these algorithms, but in the experiment of PART2, the training time of neural network is very long(almost an hour in Part 2) and its performance highly relied on the structure of dataset. In fact, the most troublesome work in Part 2 is creating a new dataset and

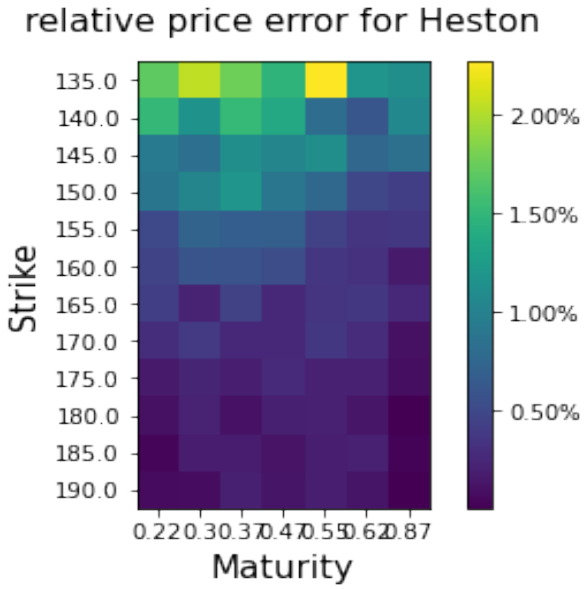


Figure 3.6: error between neural network and closed formula

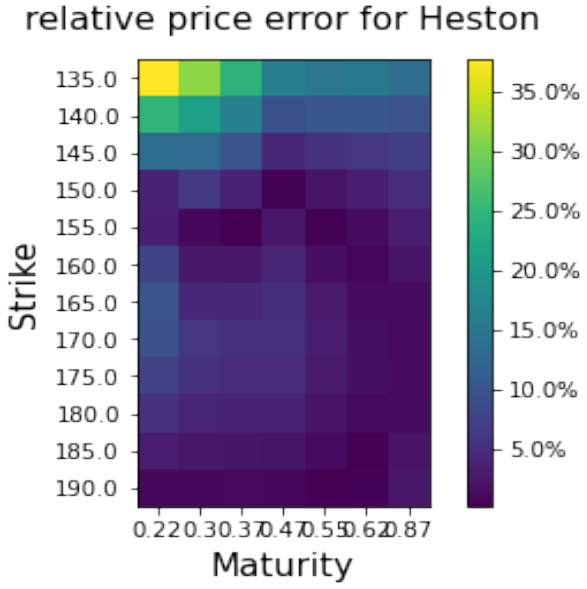


Figure 3.7: error for deep calibration

wait for an hour to see if the neural network performs well. However, there are other algorithms suitable for solving regression problems. XGBoost is a famous one. Using the same dataset in Part2, it reduced the training time of whole dataset from an hour to only 3 minutes.

3.7.2 The idea of XGBoost

XGBoost is a tree ensemble model consists of a set of classification and regression trees (CART) and it is widely used in classification and regression problems. CART tree is similar to decision tree but it assigns a score in each leaf. **Figure3.11** is an example of CART tree. Following the document of XGBoost package in python[25] The idea of XGBoost is as follows.
In mathematical words. XGBoost model can be written as:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i) \ ,$$

where x_i is the i-th sample and $f_k(x_i)$ is the sample scores get from the k-th CART tree. In mathematical word,

$$f_k(x) = w_{q(x)}, q : R^d \longrightarrow 1, 2, ...T \ ,$$

where $q(x)$ returns the index of leaf that x belongs to and w_i is the score of the i-th leaf and d is the dimension of x. The objective function is:

$$obj = \sum_{i=1}^{n} l(\hat{y}_i, y_i) + \sum_{k=1}^{k=K} \omega(f_k) \ ,$$

where l is the loss function, and $\omega(f_k)$ is the complexity of CART tree f_k, and it is defined as:

$$\omega(f) = \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \ .$$

The estimated value $\hat{y}_i$ is calculated in an iterative process:

$$\hat{y}_i^{(0)} = 0 \ ,$$

$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + f_1(x_i) \ ,$$

$$\dots$$

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) \ .$$

Plug $\hat{y}_i^{(t)}$ in the objective function and use Taylor expansion till order 2, we have:

$$obj^{(t)} \approx \sum_{i=1}^{n} [l(y_i, \hat{y_i^{(t-1)}}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + c \ ,$$

where

$$g_i = \frac{\partial}{\partial \hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \ ,$$

$$h_i = \frac{\partial^2}{\partial^2 \hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \ ,$$

c and $l(y_i, \hat{y_i^{(t-1)}})$ are constant terms. Rearrange it we have:

$$obj^{(t)} \approx \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \omega(f_t) + c1 \ ,$$

$$= \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 + c1 \ .$$

Notice that each leaf has the same score. Sum in terms of each leaf, we have:

$$obj^{(t)} \approx \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T + c1 \ ,$$

where $I_j = \{i | q(x_i) = j\}$ is the set of indices that belongs to the j-th leaf. Denote $G_i = \sum_{i \in I_j} g_i$ and $H_i = \sum_{i \in I_j} h_i$, we have:

$$obj^{(t)} \approx \sum_{j=1}^{T} [G_i w_j + \frac{1}{2} (H_i + \lambda) w_j^2] + \gamma T + c1 \ ,$$

where w_j is independent of each other, and $G_i w_j + \frac{1}{2}(H_i + \lambda) w_j^2$ is a quadratic form. Then the optimum score for the j-th leaf is

$$w_j^* = -\frac{G_j}{H_j + \lambda} \ .$$

Notice:The derivation above only talks about how to optimise scores assigned to each leaf in the base of knowing the structure of each CART tree f_k. The details about how to construct CART tree is beyond the scope of this dissertation. More details are in the paper of TianQi etal[4].

3.7.3 Implement of XGBoost

Dataset
I used the same dataset as the PART2 neural network in order to compare the predictions of the XGBoost and the neural network. The dataset has 5 columns, the first 4 columns are $S0, T, r, \gamma$ and the last column is the price of American put option, where $S0, T, r, \gamma$ is the input parameters and desired output is the price of American put option.
Use scaling relationship
After using XGBoost to learn the map $(S0, T, r, \gamma) \to \tilde{P}(S0, T, r, \gamma)$, use the scaling relationship in **2.3.2** to get the prediction for any $P(s, k, t, r_0, e, g)$, where s, k, t, r_0, e, g is the same notation in **2.3.2**.

3.7.4 The result and drawbacks of XGBoost

The 4 set of test values in Part2 are presented again here in order to compare the result of XGBoost to the result of neural network.

S0	K	T	r	σ	γ	FD	NN	XGBoost	$e_{NN}\%$	$e_{XG}\%$
1.1	1	1	0.05	0.1	0.9	0.00368	0.00370	0.00372	0.54	1.1
1	1	1	0.05	0.1	0.9	0.02404	0.02403	0.02403	0.04	0.04
0.9	1	0.5	0.02	0.1	0.5	0.10000	0.10006	0.10026	0.06	0.26
2.6	2	1	0.1	0.2	0.8	0.00243	0.002477	0.00257	1.89	5.8

As shown from the table, XGBoost prediction is quite accurate and there is only very little difference between the prediction of neural network and the prediction of XGBoost but it only takes 3 minutes to train the dataset which is very attractive.
To see the prediction effect on other input values. I plot three pictures showing the XGBoost predictions, finite different method values, and $(K - S)^+$ under three different $\gamma : 0, 0.5, 1$ when $K = 1, \sigma = 0.1, T = 1, r = 0.01$ As shown from these three pictures, XGBoost model is a good fit to the price curve of the American option when K and S are close, and this happens to be the case that the option pricer is most interested in, because if S is far from strike price K, it's easy to approximate the American put option price to be $(K - S)^+$.
Drawbacks: Till now, it seems to be a perfect alternative method to neural network. However, if we closely look at the pictures, the predictions of XGBoost are piece-wise constants. This is because the rule on the split node

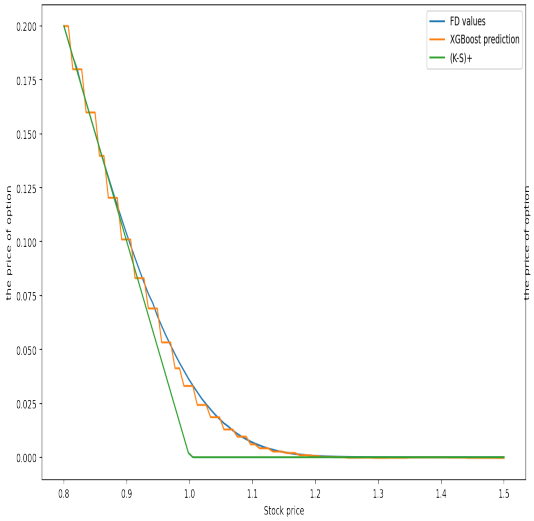


Figure 3.8: $\gamma = 0$

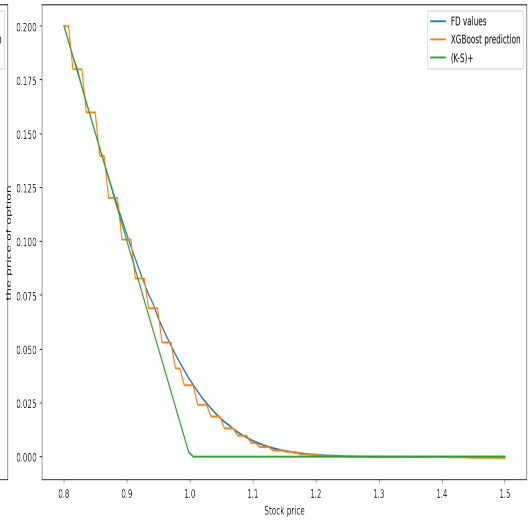


Figure 3.9: $\gamma = 0.5$

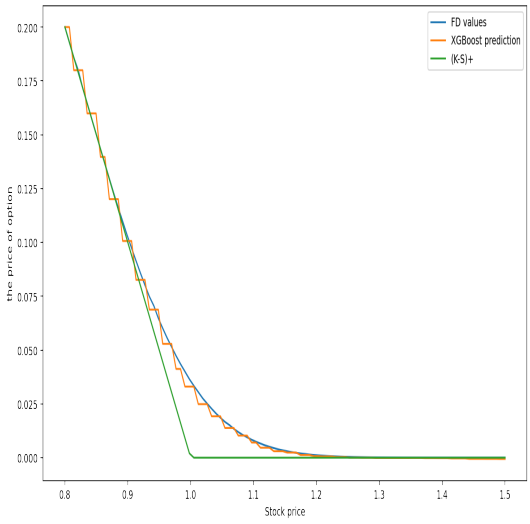


Figure 3.10: $\gamma = 1$

of the CART tree is based on whether a certain feature of the sample falls in a specified interval and each leaf has a specified score which makes the prediction not continuous. To show this, I plot The structure of CART tree in **Figure3.11**. Note that this is only part of the whole tree.

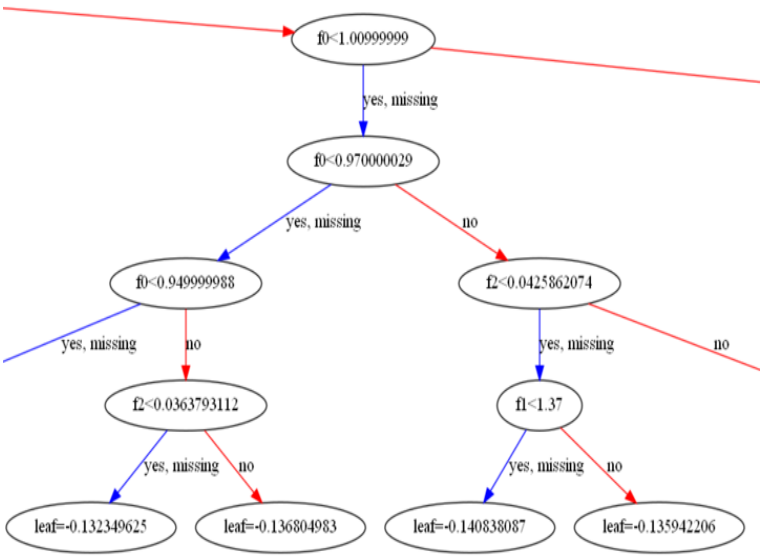


Figure 3.11: part of CART tree(fi is the $i-th$ column of a sample. In this tree, $f_0 = S0, f_1 = K, f_2 = T, f_3 = r$)

In Part 2, I plot the 'time value' for differnt γ in order to show whether neural networks predict well if stock price S is around K. In fact, 'time value' is just the prediction minis $(K-S)^+$, if I plot this picture, it can be quite tricky. Because it will amplifies the piece-wise constant characteristic of XGBoost predictions when S is far from K. Even this, it can still be tolerated. Because as I said before, option pricer care more about the cases when S is close to K and even the theoretical value of American put is K-S. The option exchange will price slightly higher than this value which downplays the error between the estimated value and the theoretical value in this case.
The real drawback comes from the smoothness of XGBoost prediction function. Piece-wise constant is not a convex function because it's not differentiable at jump points. Due to this fact, using this model for calibration can cause serious trouble when optimise

$$\theta = argmin_\theta \sum_{i=1}^{m} \sum_{j=1}^{n} \omega_{i,j} \delta(\tilde{P}^{M(\theta)}(K_i, T_j) - P^{Mk}(K_i, T_j)) \ ,$$

because gradient-based algorithm 'SLSQP' can not be used in this case. To fix this problem, I come up with a brutal way to do calibration using XGBoost model.

3.7.5 A brutal way of calibration using XGboost model

In financial market, calibration doesn't have to find the global optimal parameters, nor does it need to be accurate to many decimal places. It just needs to find a set of parameters so that its calculated theoretical option price and the market price are within an acceptable range. Considering about this, if the calculation of theoretical price is very fast and dimension of input parameters is low. Enumeration can be considered.
In the CEV model, only 2 parameters σ, γ need to be calibrated. And we already get the XGBoost prediction function which is very fast to calculate the prediction value. Therefore, the brutal way follows the following steps:
1.Define the range of parameters. In this case, $\sigma \in [0.1, 0.9]$,$\gamma \in [0, 1]$, denote $\sigma_0 = 0.1, \sigma_{max} = 0.9$, $\gamma_0 = 0.1, \gamma_{max} = 0.9$.
2.Take 10 points uniformly within the range of each parameter. $\sigma_1, \sigma_2 ... \sigma_{10}$, $\gamma_1, \gamma_2, ... \gamma_{10}$.
3.For each parameter pair, calculate the corresponding error between the theoretical price and market price, and find the pair σ_i, γ_j which gives the smallest error.
4.Set $\sigma_{i-1} = \sigma_0$, $\sigma_{i+1} = \sigma_{max}$, $\gamma_{j-1} = \gamma_0$, $\gamma_{j+1} = \gamma_{max}$ and repeat from 1 to 3.
Similar to the way in **3.5**, the effectiveness of calibration for XGBoost is shown in **Figure3.12**. As shown from the picture, when time to maturity is 0.48, the calibration effectiveness is very bad, while for other options with different time to maturity the error is between 20% to 40%. Compare this picture to **Figure3.2**, CEV model performs slightly bad for both European put options and American put option.

Figure 3.12: error for XGBoost calibration

Chapter 4

Conclusion

In chapter2, we implemented finite difference method and regarded the price calculated by it as theoretical price and create a dataset for neural network's training. However, creating a large dataset and calculate theoretical price for each sample is very time-consuming. In CEV model, if we don't know the scaling relationship, it would be very difficult to create a large dataset. Therefore, find some relationship like scaling relationship can be very helpful to create a dataset. Also, in chapter 3, we used imaged based training, which greatly reduced the training time for neural network. For similar deep calibration problem, we can use this idea. Lastly, I mentioned a different machining learning method XGBboost, which reduced the training time from an hour to only 3 minutes and used this model for calibration. I think this XGBoost method is potential for pricing options and need further research.

Appendix A

Python (MATLAB/C/R) code

Part2whole

September 1, 2022

[12]:
```
import scipy
import scipy.sparse.linalg
import scipy.interpolate
from itertools import product
from math import *
import numpy as np
import scipy.sparse
import matplotlib.pyplot as plt
#compute_abc, compute_W(a,b,c, V0, VM), compute_W(a,b,c, V0, VM) are based on␣
 ↪John.Armstrong's lecture notes
#https://keats.kcl.ac.uk/course/view.php?id=93365
def bottom_boundary_condition( K, T, S_min, r, t):
    return np.ones(t.shape)*K

def top_boundary_condition( K, T, S_max, r, t):
    return np.maximum(K-S_max,0)

def final_boundary_condition( K, T, S ):
    return np.maximum(K-S,0)

def compute_abc( K, T, sigma, r, S, dt, dS, beta ):
    a = -sigma**2 * S**(2*beta)/(2* dS**2 ) + r*S/(2*dS)
    b = r + sigma**2 * S**(2*beta)/(dS**2)
    c = -sigma**2 * S**(2*beta)/(2* dS**2 ) - r*S/(2*dS)
    return a,b,c

def compute_lambda( a,b,c ):
    return scipy.sparse.diags( [a[1:],b,c[:-1]],offsets=[-1,0,1],format='csr')

def compute_W(a,b,c, V0, VM):
    M = len(b)+1
    W = np.zeros(M-1)
    W[0] = a[0]*V0
    W[-1] = c[-1]*VM
    return W
#based on John.Armstrong's lecture notes
#https://keats.kcl.ac.uk/course/view.php?id=93365
```

```
def price_put_crank_nicolson( K, T, r, sigma, N, M, beta, S_min,␣
 ↪S_max,typeofoption="American"):
    dt = T/N
    dS = (S_max-S_min)/M
    S = np.linspace(S_min,S_max,M+1)
    t = np.linspace(0,T,N+1)
    V = np.zeros((N+1,M+1)) #...
    B=np.zeros(N)
    V[:,-1] = top_boundary_condition(K,T,S_max,r,t)
    V[:,0] = bottom_boundary_condition(K,T,S_max,r,t)
    V[-1,:] = final_boundary_condition(K,T,S)
    a,b,c = compute_abc(K,T,sigma,r,S[1:-1],dt,dS,beta)
    Lambda =compute_lambda(a,b,c) #...
    identity = scipy.sparse.identity(M-1)
    for i in range(N-1,-1,-1):
        Wt = compute_W(a,b,c,V[i,0],V[i,M])
        Wt_plus_dt = compute_W(a,b,c,V[i+1,0],V[i+1,M])
        if typeofoption=='American':
            V[i,1:M] = np.maximum(scipy.sparse.linalg.spsolve(identity+0.
 ↪5*Lambda*dt,(identity-0.5*Lambda*dt).dot(V[i+1,1:M]) -0.5*dt*(Wt_plus_dt +␣
 ↪Wt)),K-S[1:M])
            sign=(V[i-1,1:M]-(K-S[1:M]))>0
            for j in range(M-1):
                if sign[j]==True:
                    B[i-1]=(j+1)*dS+S_min
                    break
        elif typeofoption=='European':
            V[i,1:M] = scipy.sparse.linalg.spsolve(identity+0.
 ↪5*Lambda*dt,(identity-0.5*Lambda*dt).dot(V[i+1,1:M]) -0.5*dt*(Wt_plus_dt +␣
 ↪Wt))
        else:
            print('please enter a type from American or European')
    return V, t, S


###interpolation function
def findprice(S0,TTM,V,S_max,Tmax,N,M):
    col=S0/(S_max/M)
    row=(Tmax-TTM)/(Tmax/N)
    upcol=ceil(col)
    downcol=floor(col)
    uprow=ceil(row)
    downrow=floor(row)
    if (col-int(col))<0.1 and (row-int(row))>0.1:
        f1=scipy.interpolate.interp1d(np.array([downrow,uprow]), np.
 ↪array([V[downrow,int(col)],V[uprow,int(col)]]))
        P=f1(row)
```

```
    elif (col-int(col))>0.1 and (row-int(row))<0.1:
        f2=scipy.interpolate.interp1d(np.array([downcol,upcol]), np.
 ↪array([V[int(row),downcol],V[int(row),upcol]]))
        P=f2(col)
    elif (col-int(col))<0.1 and (row-int(row))<0.1:
        P=(V[int(row),int(col)])
    else:
        f3=scipy.interpolate.interp2d(np.array([downrow,uprow]),np.
 ↪array([downcol,upcol]),np.
 ↪array([[V[downrow,downcol],V[downrow,upcol]],[V[uprow,downcol],V[uprow,upcol]]]))
        P=f3(row,col)
    return float(P)
#based on John.Armstrong's lecture notes
#https://keats.kcl.ac.uk/course/view.php?id=93365
def price_put_explicit( K, T, r, sigma, N, M, beta, S_min, S_max):
    # Choose the shape of the grid
    dt = T/N
    dS = (S_max-S_min)/M
    S = np.linspace(S_min,S_max,M+1)
    t = np.linspace(0,T,N+1)
    V = np.zeros((N+1,M+1)) #...
    B=np.zeros(N)
    # Set the boundary conditions
    V[:,-1] = top_boundary_condition(K,T,S_max,r,t)
    V[:,0] = bottom_boundary_condition(K,T,S_max,r,t)
    V[-1,:] = final_boundary_condition(K,T,S) #...
    # Apply the recurrence relation
    a,b,c = compute_abc(K,T,sigma,r,S[1:-1],dt,dS,beta)
    Lambda =compute_lambda( a,b,c)
    identity = scipy.sparse.identity(M-1, format='csr')
    for i in range(N,0,-1):
        W = compute_W(a,b,c,V[i,0],V[i,M])
        # Use `dot` to multiply a vector by a sparse matrix
        V[i-1,1:M] = np.maximum((identity-Lambda*dt).dot( V[i,1:M] ) - W*dt,␣
 ↪K-S[1:M])
        sign=(V[i-1,1:M]-(K-S[1:M]))>0
        for j in range(M-1):
            if sign[j]==True:
                B[i-1]=(j+1)*dS+S_min
                break
    return V, t, S,B
#based on John.Armstrong's lecture notes
#https://keats.kcl.ac.uk/course/view.php?id=93365
def plot_option_price(V,t,S):
    M = len(S)-1
    S_zoom = S[0:int(M)]
    V_zoom=V[:,0:int(M)]
```

```
    t_mesh, S_mesh = np.meshgrid(t,S_zoom)
    ax = plt.axes(projection='3d')
    ax.plot_surface(t_mesh,S_mesh,V_zoom.T, alpha=0.3,edgecolor='k');
    ax.set_xlabel('Time (t)')
    ax.set_ylabel('Stock price (S)')
    ax.set_zlabel('Option price')
```

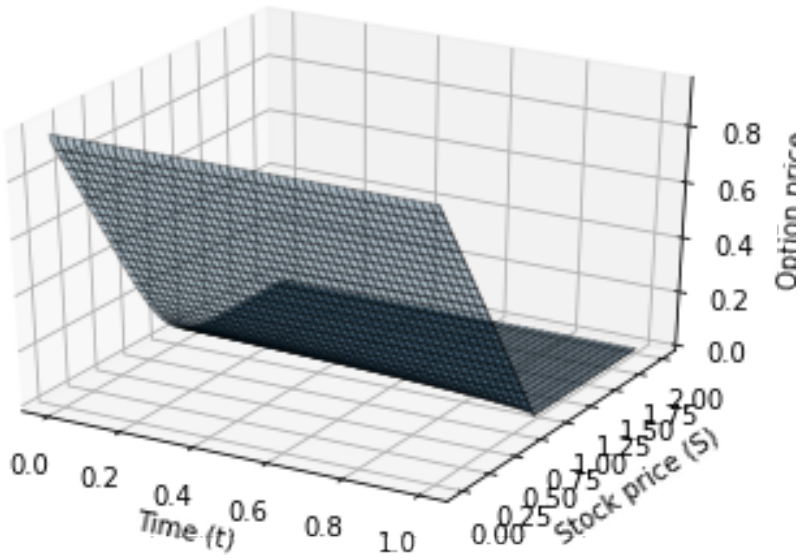
[4]:
```
def norm2(x,y):
    d=0
    for i in range(x.size):
        d=d+(x[i]-y[i])**2
    return d
##SOR method for solving a system of linear equations
def SOR(A,b,x,e,w,times):
    D=np.array(np.diag(np.diag(A)))
    L=np.triu(A,1)
    U=np.tril(A,-1)
    Sw=np.linalg.inv(D+w*L)@((1-w)*D-w*U)
    fw=w*(np.linalg.inv(D+w*L))@(b.T)
    x0=x
    x=Sw@x0.T+fw
    k=1
    while k< times:
        if norm2(x,x0)>e:
            x0=x
            x=Sw@x0.T+fw
            k=k+1
        else:
            break
    return x
def compute_lambda1( a,b,c ):
    array_a=np.diag(b)
    array=np.diag(a[1:])
    array1=np.diag(c[:-1])
    array_b=np.insert(array,0,values=np.zeros(b.size-1),axis=0)
    array_b=np.insert(array_b,b.size-1,values=np.zeros(b.size),axis=1)
    array_c=np.insert(array1,b.size-1,values=np.zeros(b.size-1),axis=0)
    array_c=np.insert(array_c,0,values=np.zeros(b.size),axis=1)
    matrix=array_a+array_b+array_c
    return matrix
def price_put_crank_nicolson_SOR( K, T, r, sigma, N, M, beta):
    x=np.zeros(M-1)
    e=pow(10,-6)
    times=100
    w=1
    dt = T/N
    S_min=0
```

```
    S_max=2
    dS = (S_max-S_min)/M
    S = np.linspace(S_min,S_max,M+1)
    t = np.linspace(0,T,N+1)
    V = np.zeros((N+1,M+1)) #...
    B=np.zeros(N)
    V[:,-1] = top_boundary_condition(K,T,S_max,r,t)
    V[:,0] = bottom_boundary_condition(K,T,S_max,r,t)
    V[-1,:] = final_boundary_condition(K,T,S)
    a,b,c = compute_abc(K,T,sigma,r,S[1:-1],dt,dS,beta)
    Lambda =compute_lambda1( a,b,c) #...
    QWE=np.ones(M-1)
    identity = np.diag(QWE)
    for i in range(N-1,-1,-1):
        Wt = compute_W(a,b,c,V[i,0],V[i,M])
        Wt_plus_dt = compute_W(a,b,c,V[i+1,0],V[i+1,M])
        V[i,1:M] = np.maximum(SOR((identity+0.5*Lambda*dt),(identity-0.
↪5*Lambda*dt).dot(V[i+1,1:M]) -0.5*dt*(Wt_plus_dt + Wt),x,e,w,times),K-S[1:M])
        sign=(V[i-1,1:M]-(K-S[1:M]))>0
        for j in range(M-1):
            if sign[j]==True:
                B[i-1]=(j+1)*dS+S_min
                break
    return V, t, S,B
```

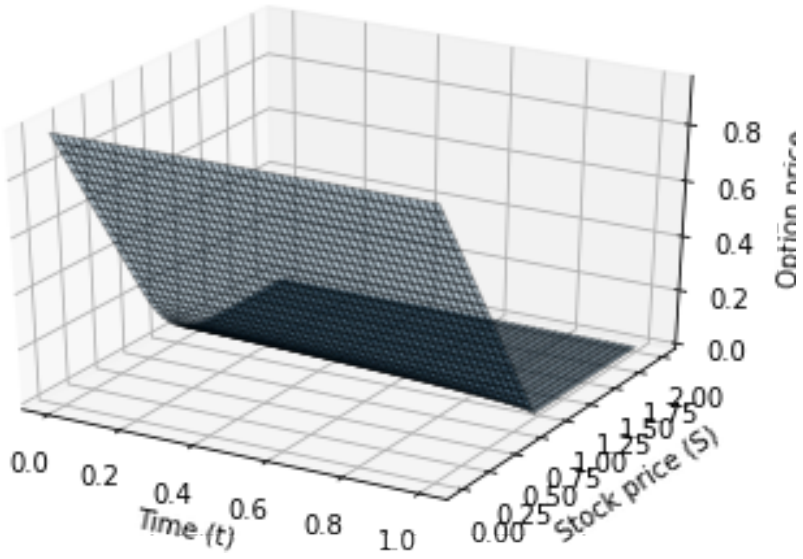
[6]:
```
#Use this to run SOR method
V, t, S=price_put_crank_nicolson(0.95, 1, 0.02, 0.2, 1000, 100, 1, 0, 2,␣
↪'American')
print(V[0,50])
V, t, S,B= price_put_crank_nicolson_SOR( 0.95, 1, 0.02, 0.2, 1000, 100, 1)
print(V[0,50])
del V
del t
del S
del B
```

```
0.04835745341940862
0.04783471808633215
```

[15]:
```
#Plot European put price surface
V2, t2, S2=price_put_crank_nicolson(0.95, 1, 0.02, 0.2, 800, 400, 0.
↪7,0,2,'European')
V, t, S,B=price_put_explicit( 0.95, 1, 0.02, 0.2, 10000, 400, 0.7,0,2)
plot_option_price(V2,t2,S2)
```



[16]:
```
#Plot American put price surface
plot_option_price(V,t,S)
```



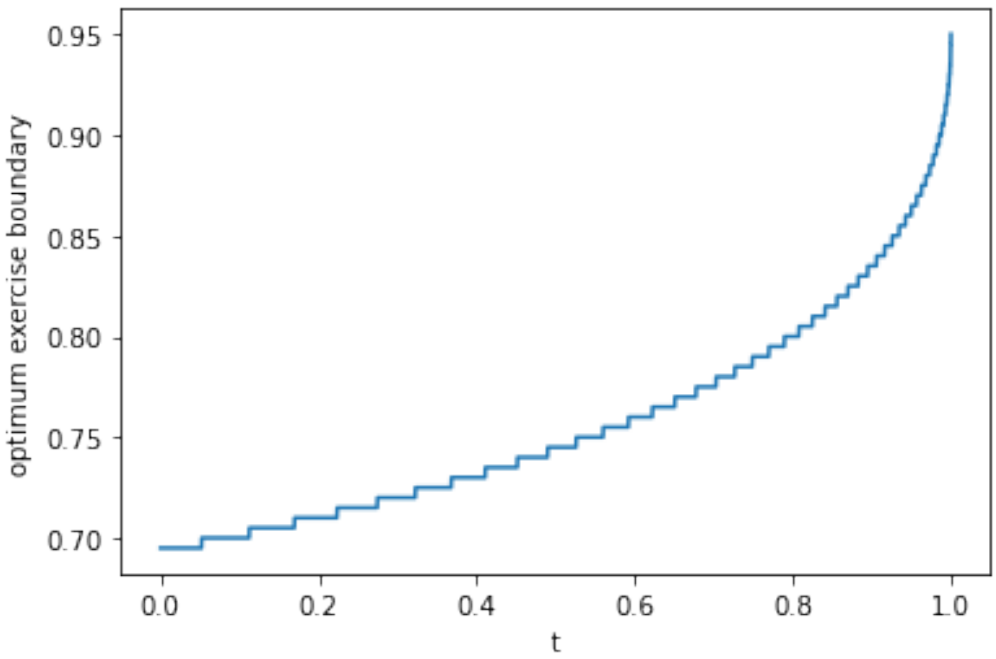
[17]:
```
#test for fdm
def montecarlotest(S,T,K,N,M,r,sigma,gamma):
    A=np.zeros((N+1,M+1))
    A[:,0]=S
```

```
    dt=T/M
    for i in range(1,M+1):
        dW=np.random.randn(N+1)*(dt**0.5)
        A[:,i]=A[:,i-1]+r*A[:,i-1]*dt+sigma*(A[:,i-1]**gamma)*(dW.T)
    return np.maximum((K-A[:,M]),0).mean()*np.exp(-r*T)
print(montecarlotest(1.2,1,0.95,100000,100,0.02,0.2,0.7))
print(V[0,240])
```

```
0.00955597394255625
0.009670418153185721
```

[18]:
```
##plot exercise boundary
plt.plot(t[0:10000],B)
plt.xlabel("t")
plt.ylabel("optimum exercise boundary")
print(B[0])
```

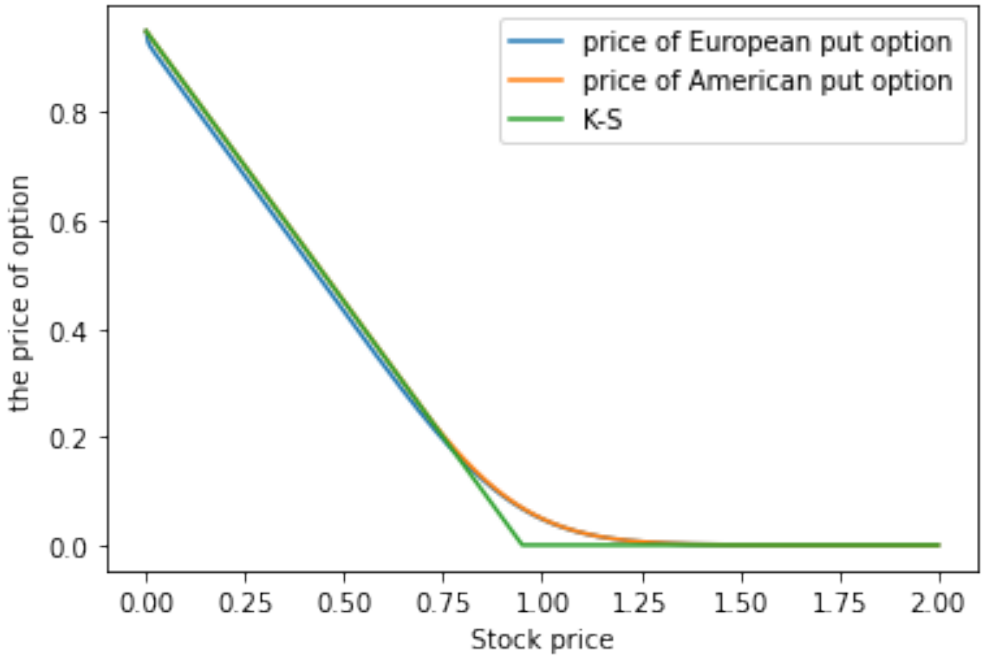
```
0.6950000000000001
```



[19]:
```
#plot 2D prices
plt.plot(S[0:401],V2[0,:],label='price of European put option')
plt.plot(S[0:401],V[0,:],label='price of American put option')
plt.plot(S[0:401],np.maximum(0.95-S,0),label='K-S')
plt.xlabel("Stock price")
plt.ylabel("the price of option")
```

```
plt.legend('A fan diagram of geometric Brownian motion')
plt.legend()
```

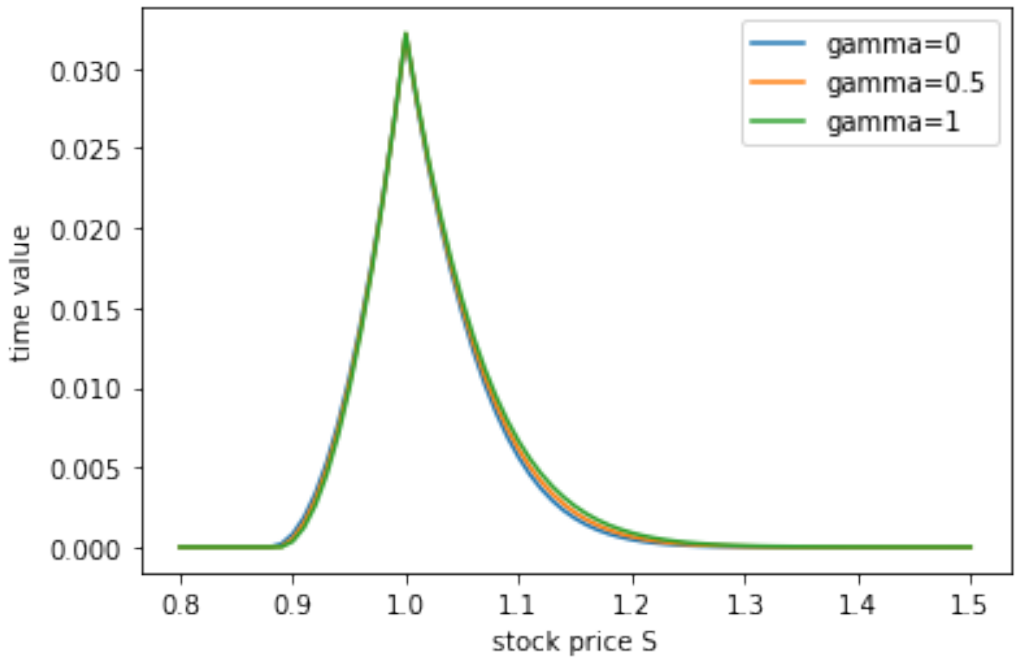
[19]: <matplotlib.legend.Legend at 0x7f9060ab9cd0>



[0]:
```
#setting parameters and plot time value of fdm
ss=np.linspace(0.7,1.3,61)
sslow=np.linspace(0.5,0.7,21)
ssup=np.linspace(1.3,1.9,21)
ss=np.concatenate((ss,sslow))
ss=np.concatenate((ss,ssup))
tt=np.linspace(0,2,41)
S_min=0
S_max=3
T=2
Tmax=2
N=2000
M=300
fixK=1
fixsigma=0.15
K=1
r=0.02
sigma=0.1
gamma0=0
gamma05=0.5
```

```
gamma1=1
SS0=np.linspace(0.8,1.5,71)
V00, t00, S00= price_put_crank_nicolson(  K, T, r, sigma, N, M, gamma0, S_min,␣
 ↪S_max,typeofoption="American")
V05, t05, S05= price_put_crank_nicolson(  K, T, r, sigma, N, M, gamma05, S_min,␣
 ↪S_max,typeofoption="American")
V10, t10, S10= price_put_crank_nicolson(  K, T, r, sigma, N, M, gamma1, S_min,␣
 ↪S_max,typeofoption="American")
FD=np.zeros((3,71))
for i in range(71):
  FD[0,i]=findprice(SS0[i],1,V00,S_max,Tmax,N,M)
  FD[1,i]=findprice(SS0[i],1,V05,S_max,Tmax,N,M)
  FD[2,i]=findprice(SS0[i],1,V10,S_max,Tmax,N,M)
plt.plot(SS0,FD[0,:]-np.maximum(1-SS0,0),label='gamma=0')
plt.plot(SS0,FD[1,:]-np.maximum(1-SS0,0),label='gamma=0.5')
plt.plot(SS0,FD[2,:]-np.maximum(1-SS0,0),label='gamma=1')
plt.xlabel('stock price S')
plt.ylabel('time value')
plt.legend()
```

[0]: <matplotlib.legend.Legend at 0x7f3f7174df10>



[0]:
```
#creating dataset: don't run this page if you don't need to create dataset but␣
 ↪use downloaded one.
```

```
def createdata(ss,tt,n):
  with open('data.txt',"w") as f:
    rr=np.linspace(0,0.15,n)
    ggmama=np.linspace(0,1,n)
    for i in range(n):
      for j in range(n):
        r=rr[i]
        gamma=ggmama[j]
        V, t, S= price_put_crank_nicolson( fixK, T, r, fixsigma, N, M, gamma,␣
 ↪S_min, S_max,typeofoption="American")
        print(i,j)
        for p in range(ss.shape[0]):
          s=ss[p]
          for q in range(tt.shape[0]):
            t=tt[q]
            print(s,t,r,gamma,findprice(s,t,V,S_max,Tmax,N,M),file=f)
    f.close()
    return 0
createdata(ss,tt,20)
with open('data.txt',"r") as f:
  validdata = np.loadtxt(f)
```

[0]:

[0]:
```
#if you have already downloaded a dataset, run first two commands
with open('datanew.txt',"r") as f:
  validdata = np.loadtxt(f)


#delete some samples with extreme small price.
count=0
for i in range(validdata.shape[0]):
  if validdata[i,4]>0.00001:
    count=count+1
validdata1=np.zeros((count,5))
row=0
for k in range(validdata.shape[0]):
  if validdata[k,4]>0.00001:
    validdata1[row,:]=validdata[k,:]
    row=row+1
print(count)
```

```
1412417
```

[0]:
```
#scale the dataset
#scale method 2
xx=validdata1[:,0:4]
```

```
yy=validdata1[:,4]
print(yy.shape)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    xx, yy, test_size=0.1, random_state=42)

x_train_transform=np.zeros((X_train.shape[0],X_train.shape[1]))
x_test_transform=np.zeros((X_test.shape[0],X_test.shape[1]))
for i in range(4):
    x_train_transform[:,i]=(2*X_train[:,i]-(X_train[:,i].max()-X_train[:,i].
 ↪min()))/(X_train[:,i].max()-X_train[:,i].min())
    x_test_transform[:,i]=(2*X_test[:,i]-(X_test[:,i].max()-X_test[:,i].min()))/
 ↪(X_test[:,i].max()-X_test[:,i].min())
y_train_transform=(2*y_train-(y_train.max()-y_train.min()))/(y_train.
 ↪max()-y_train.min())
y_test_transform=(2*y_test-(y_test.max()-y_test.min()))/(y_test.max()-y_test.
 ↪min())
```

```
(1412417,)
```

[0]:
```
#construct the loss function
def ziweifunction(y_true,y_predict):
    if y_true<0.01:
        a=(abs(y_true-y_predict)/y_true)/5
    else:
        a=abs(y_true-y_predict)
    return a
def ziweifunction2(y_true,y_predict):
    a=abs((y_true-y_predict)/y_true+0.0001)*100
    return a
def newpercent(y_true,y_predict):
    a=log(abs((y_true-y_predict**2)))
    return a
from keras import backend as K
def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))
```

[0]:
```
#training the neural network. Change epochs to 100 to improve accuracy
import tensorflow.keras as keras
g_hat = keras.Sequential([
    keras.layers.Dense(30, activation="elu", input_shape=(4,)),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(1, activation="linear")
]
)
```

```
g_hat.summary()
g_hat.compile(optimizer="adam", loss=root_mean_squared_error)
g_hat.fit(x_train_transform, y_train_transform, batch_size=80,␣
 ↪epochs=70,validation_data = (x_test_transform,y_test_transform))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 30)                150

 dense_1 (Dense)             (None, 30)                930

 dense_2 (Dense)             (None, 30)                930

 dense_3 (Dense)             (None, 30)                930

 dense_4 (Dense)             (None, 1)                 31

=================================================================
Total params: 2,971
Trainable params: 2,971
Non-trainable params: 0
_________________________________________________________________
Epoch 1/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0096 -
val_loss: 0.0031
Epoch 2/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0047 -
val_loss: 0.0050
Epoch 3/70
15890/15890 [==============================] - 39s 2ms/step - loss: 0.0039 -
val_loss: 0.0048
Epoch 4/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0034 -
val_loss: 0.0029
Epoch 5/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0030 -
val_loss: 0.0018
Epoch 6/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0028 -
val_loss: 0.0031
Epoch 7/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0026 -
val_loss: 0.0024
Epoch 8/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0024 -
```

```
val_loss: 0.0029
Epoch 9/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0023 -
val_loss: 0.0031
Epoch 10/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0021 -
val_loss: 0.0022
Epoch 11/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0020 -
val_loss: 0.0022
Epoch 12/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0019 -
val_loss: 0.0021
Epoch 13/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0019 -
val_loss: 0.0011
Epoch 14/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0018 -
val_loss: 0.0025
Epoch 15/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0017 -
val_loss: 0.0015
Epoch 16/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0017 -
val_loss: 0.0021
Epoch 17/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0016 -
val_loss: 0.0015
Epoch 18/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0016 -
val_loss: 0.0011
Epoch 19/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0016 -
val_loss: 0.0014
Epoch 20/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0016 -
val_loss: 0.0023
Epoch 21/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0015 -
val_loss: 0.0015
Epoch 22/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0015 -
val_loss: 0.0016
Epoch 23/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0015 -
val_loss: 0.0018
Epoch 24/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0014 -
```

```
val_loss: 0.0010
Epoch 25/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0014 -
val_loss: 0.0010
Epoch 26/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0014 -
val_loss: 0.0021
Epoch 27/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0014 -
val_loss: 0.0012
Epoch 28/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0014 -
val_loss: 9.2027e-04
Epoch 29/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0014 -
val_loss: 0.0024
Epoch 30/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0014 -
val_loss: 0.0014
Epoch 31/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0013 -
val_loss: 0.0011
Epoch 32/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0013 -
val_loss: 0.0018
Epoch 33/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0013 -
val_loss: 0.0025
Epoch 34/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0013 -
val_loss: 0.0024
Epoch 35/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0013 -
val_loss: 0.0023
Epoch 36/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0013 -
val_loss: 0.0013
Epoch 37/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0013 -
val_loss: 0.0012
Epoch 38/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0013 -
val_loss: 0.0013
Epoch 39/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0013 -
val_loss: 8.5706e-04
Epoch 40/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
```

```
val_loss: 0.0012
Epoch 41/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 0.0019
Epoch 42/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0012 -
val_loss: 0.0012
Epoch 43/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 0.0017
Epoch 44/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 8.4213e-04
Epoch 45/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0012 -
val_loss: 0.0012
Epoch 46/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 8.5680e-04
Epoch 47/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 0.0011
Epoch 48/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 0.0021
Epoch 49/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0012 -
val_loss: 0.0016
Epoch 50/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0012 -
val_loss: 0.0018
Epoch 51/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 0.0013
Epoch 52/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0012 -
val_loss: 0.0018
Epoch 53/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 0.0013
Epoch 54/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 6.8464e-04
Epoch 55/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0011
Epoch 56/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
```

```
val_loss: 0.0019
Epoch 57/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 0.0028
Epoch 58/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0010
Epoch 59/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0011 -
val_loss: 0.0017
Epoch 60/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0012 -
val_loss: 0.0014
Epoch 61/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0012
Epoch 62/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 5.4420e-04
Epoch 63/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0011 -
val_loss: 6.9131e-04
Epoch 64/70
15890/15890 [==============================] - 37s 2ms/step - loss: 0.0011 -
val_loss: 7.7571e-04
Epoch 65/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0019
Epoch 66/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0015
Epoch 67/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0011 -
val_loss: 7.3007e-04
Epoch 68/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 0.0011
Epoch 69/70
15890/15890 [==============================] - 36s 2ms/step - loss: 0.0011 -
val_loss: 8.6830e-04
Epoch 70/70
15890/15890 [==============================] - 38s 2ms/step - loss: 0.0011 -
val_loss: 4.9591e-04
```

```
[0]: <keras.callbacks.History at 0x7f3f01f613d0>
```

[0]:
```
#scaling relationship function
def scalingfunction(a,b,c,d,e,f,K,sigma):
    lamb=b/K
    S0=a/lamb
    gamma=f
    alpha=(((lamb**(1-gamma))*sigma)/e)**2
    r=d*alpha
    T=c/alpha
    return np.array((S0,T,r,gamma)),lamb
#get predictions
ans1,lamb1=scalingfunction(1.1,1,1,0.05,0.1,0.9,1,0.15)
ans2,lamb2=scalingfunction(1,1,1,0.05,0.1,0.9,1,0.15)
ans3,lamb3=scalingfunction(0.9,1,0.5,0.02,0.1,0.5,1,0.15)
ans4,lamb4=scalingfunction(2.6,2,1,0.1,0.2,0.8,1,0.15)
A=np.array([ans1,ans2,ans3,ans4])
for i in range(4):
    A[:,i]=(2*A[:,i]-(X_train[:,i].max()-X_train[:,i].min()))/(X_train[:,i].
 ↪max()-X_train[:,i].min())
prediction=(g_hat.predict(A)*(y_train.max()-y_train.min())+(y_train.
 ↪max()-y_train.min()))/2
print(prediction[0]*lamb1,prediction[1]*lamb2,prediction[2]*lamb3,prediction[3]*lamb4)
```

```
[0.00388429] [0.02433939] [0.10014172] [0.00256228]
```

[0]:
```
#FDM value
V9,t9,s9=price_put_crank_nicolson( 1, T, 0.05, 0.1, N, M, 0.9, S_min,␣
 ↪S_max,typeofoption="American")
V10,t10,s10=price_put_crank_nicolson( 1, T, 0.02, 0.1, N, M, 0.5, S_min,␣
 ↪S_max,typeofoption="American")
V11,t11,s11=price_put_crank_nicolson( 2, T, 0.1, 0.2, N, M, 0.8, S_min,␣
 ↪S_max,typeofoption="American")
print(findprice(1.
 ↪1,1,V9,S_max,Tmax,N,M),findprice(1,1,V9,S_max,Tmax,N,M),findprice(0.9,0.
 ↪5,V10,S_max,Tmax,N,M),findprice(2.6,1,V11,S_max,Tmax,N,M))
```

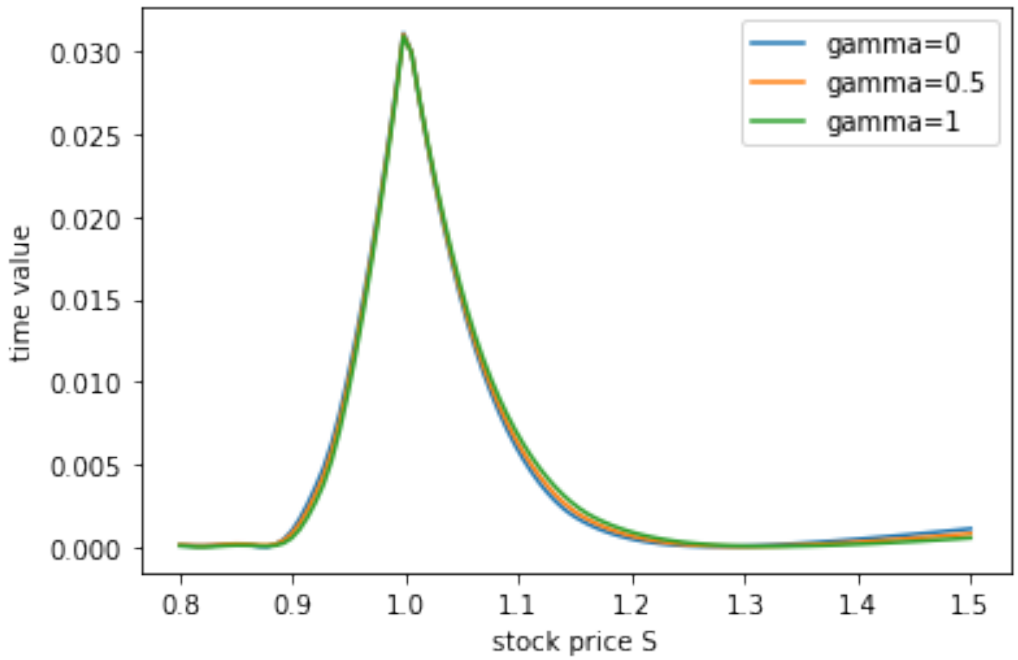
```
0.003739863919322287 0.024278618567804368 0.09999999999999998
0.002443116884701212
```

[0]:
```
#plot time value of neural network
def NN_CEV(S0,K,T,r,sigma,gamma,fixK,fixsigma):
  para,lamb=scalingfunction(S0,K,T,r,sigma,gamma,fixK,fixsigma)
  for i in range(4):
    para[i]=(2*para[i]-(X_train[:,i].max()-X_train[:,i].min()))/(X_train[:,i].
 ↪max()-X_train[:,i].min())
  prediction=g_hat.predict(np.array([para]))[0][0]
  prediction=(prediction*(y_train.max()-y_train.min())+(y_train.max()-y_train.
 ↪min()))/2
```

```
  prediction=prediction*lamb
  return prediction
print(NN_CEV(1.1,1,1,0.05,0.1,0.9,fixK,fixsigma),NN_CEV(1,1,1,0.05,0.1,0.
  ↪9,fixK,fixsigma),NN_CEV(0.9,1,0.5,0.02,0.1,0.5,fixK,fixsigma),NN_CEV(2.
  ↪6,2,1,0.1,0.2,0.8,fixK,fixsigma))
SS0=np.linspace(0.8,1.5,100)
pred0=np.zeros((3,100))
for i in range(100):
  pred0[0,i]=NN_CEV(SS0[i],1,1,0.02,0.1,0,fixK,fixsigma)
  pred0[1,i]=NN_CEV(SS0[i],1,1,0.02,0.1,0.5,fixK,fixsigma)
  pred0[2,i]=NN_CEV(SS0[i],1,1,0.02,0.1,1,fixK,fixsigma)
plt.plot(SS0,pred0[0,:]-np.maximum(1-SS0,0),label='gamma=0')
plt.plot(SS0,pred0[1,:]-np.maximum(1-SS0,0),label='gamma=0.5')
plt.plot(SS0,pred0[2,:]-np.maximum(1-SS0,0),label='gamma=1')
plt.xlabel('stock price S')
plt.ylabel('time value')
plt.legend()
```

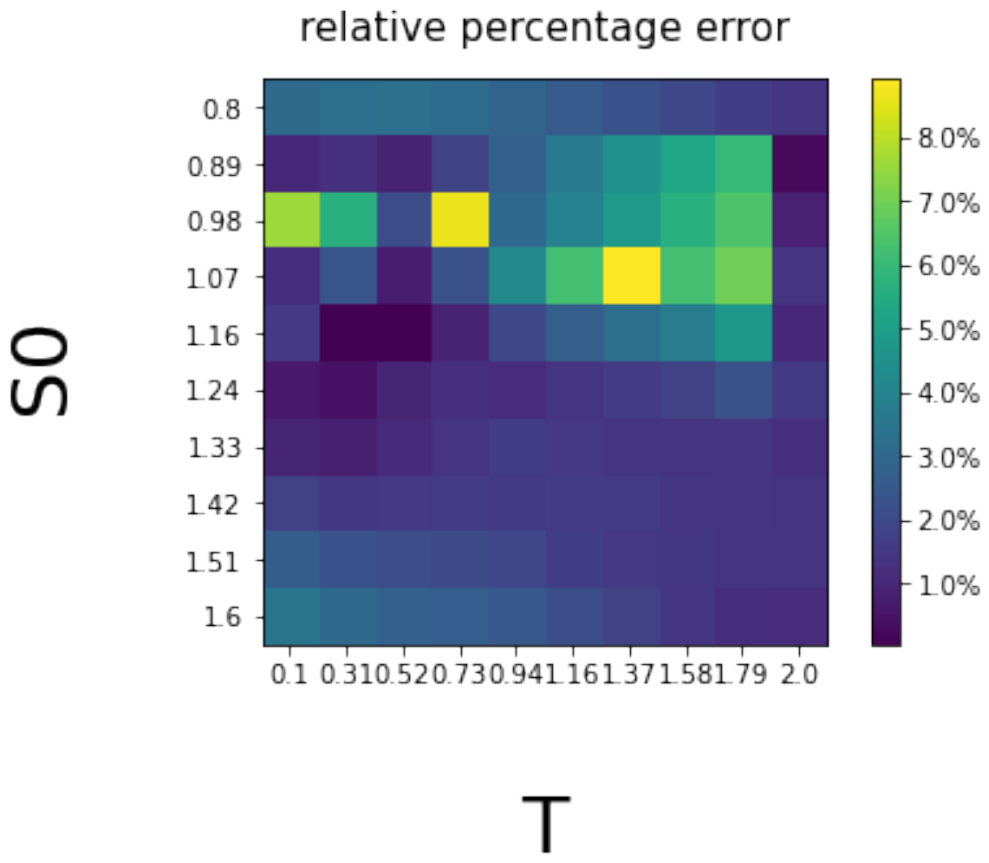
```
0.0038842885336169974 0.024339390941465755 0.1001417219032214
0.0025622908731434135
```

[0]: <matplotlib.legend.Legend at 0x7f3f01f496d0>



[0]:
```
#showing effectiveness change gamma to get another picture
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
SS0=np.linspace(0.8,1.6,10)
TT=np.linspace(0.1,2,10)
r=0.02
K=1
def NN_prices(SS0,K,TT,r,sigma,gamma):
  prices=np.zeros((SS0.shape[0],TT.shape[0]))
  for i in range(SS0.shape[0]):
    for j in range(TT.shape[0]):
      prices[i,j]=NN_CEV(SS0[i],K,TT[j],r,sigma,gamma,fixK,fixsigma)
  return prices
def FDM_prices(SS0,TT,V):
  prices0=np.zeros((SS0.shape[0],TT.shape[0]))
  for i in range(SS0.shape[0]):
    for j in range(TT.shape[0]):
      prices0[i,j]=findprice(SS0[i],TT[j],V,S_max,Tmax,N,M)
  return prices0
def rela(prices,prices0):
  rela=np.zeros((prices.shape[0],prices.shape[1]))
  for i in range(prices.shape[0]):
    for j in range(prices.shape[1]):
      if abs(prices[i,j]-prices0[i,j])<0.001:
        rela[i,j]=abs(prices[i,j]-prices0[i,j])*100
      else:
        rela[i,j]=abs(prices[i,j]-prices0[i,j])/(prices0[i,j])
  return rela
prices=NN_prices(SS0,K,TT,r,sigma,gamma1)
prices0=FDM_prices(SS0,TT,V10)
rela1=rela(prices,prices0)
ax=plt.subplot(1,1,1)
plt.title("relative percentage error",fontsize=15,y=1.04)
plt.imshow(rela1*100)
plt.colorbar(format=mtick.PercentFormatter())
ax.set_xticks(np.linspace(0,len(TT)-1,len(TT)))
ax.set_xticklabels(np.round(TT,2))
ax.set_yticks(np.linspace(0,len(SS0)-1,len(SS0)))
ax.set_yticklabels(np.round(SS0,2))
plt.xlabel("T",fontsize=30,labelpad=40)
plt.ylabel("S0",fontsize=30,labelpad=40)
```

[0]:
```
Text(0, 0.5, 'S0')
```


relative percentage error
S0
T
0.8
0.89
0.98
1.07
1.16
1.24
1.33
1.42
1.51
1.6
0.1 0.31 0.52 0.73 0.94 1.16 1.37 1.58 1.79 2.0
8.0%
7.0%
6.0%
5.0%
4.0%
3.0%
2.0%
1.0%

[0]:

[0]:

[0]:

[0]:

1

2

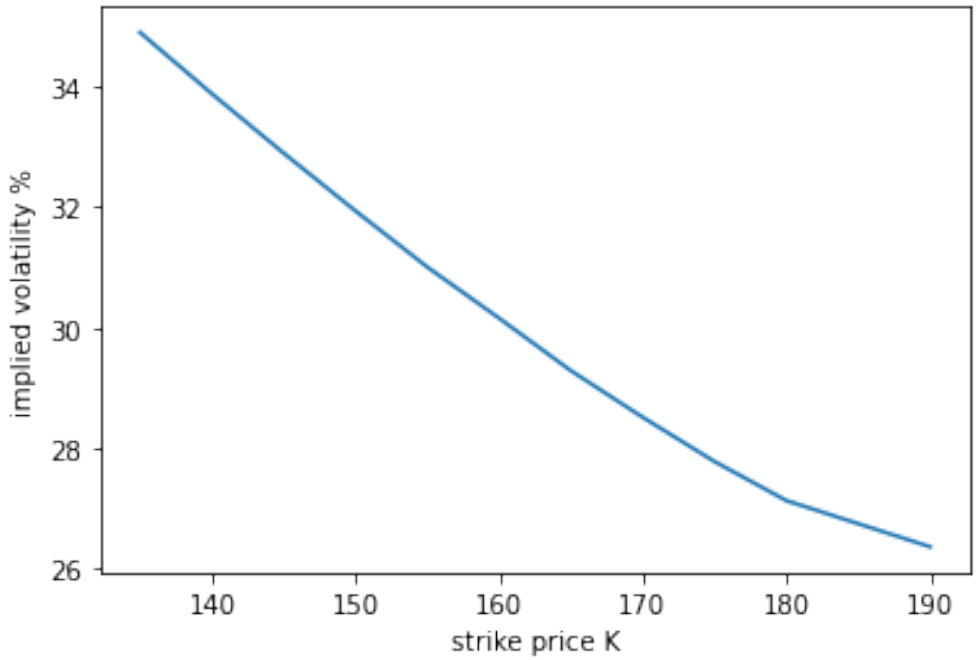
[0]:

jump_calibration

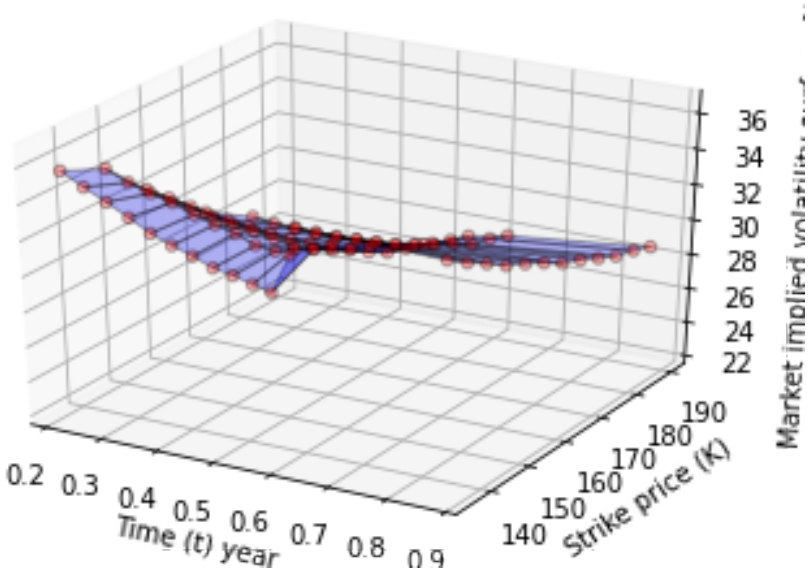
September 1, 2022

[2]:
```
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
import matplotlib.pyplot as plt
#import data and plot volatility smile
SURFACE = pd.read_excel('SURFACE.xlsx', header=0)
prices0 = np.array(pd.read_excel('prices.xlsx', header=0))
V=np.array(SURFACE)
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
plt.plot(K,V[:,3])
plt.xlabel("strike price K")
plt.ylabel("implied volatility %")
```

[2]:
```
Text(0, 0.5, 'implied volatility %')
```



[3]:
```
#plot implied volatility surface
def plot_iv_surface(V,t,K):
    K_zoom = K
    V_zoom=V
    t_mesh, K_mesh = np.meshgrid(t,K_zoom)
    ax = plt.axes(projection='3d')
    ax.plot_surface(t_mesh,K_mesh,V_zoom, alpha=0.3,edgecolor='k',color="blue");
    ax.scatter(t_mesh,K_mesh,V_zoom, alpha=0.3,edgecolor='k',color="red");
    ax.set_xlabel('Time (t) year')
    ax.set_ylabel('Strike price (K)')
    ax.set_zlabel('Market implied volatility surface %')
plot_iv_surface(V,t,K)
```



[4]:
```
#All Black and Scholes formula code based on John.Armstrong's lecture notes
#https://keats.kcl.ac.uk/course/view.php?id=93365
import scipy.stats
def N(x):
    return scipy.stats.norm.cdf(x)
def compute_d1_and_d2( S, t, K, T, r, sigma):
    tau = T-t
    d1 = 1/(sigma*np.sqrt(tau))*(np.log(S/K) + (r+0.5*sigma**2)*tau)
    d2 = d1 - sigma*np.sqrt(tau)
    return d1,d2
def black_scholes_call_price(S, t, K, T, r, sigma):
    d1, d2 = compute_d1_and_d2(S,t,K,T,r,sigma)
    return S*N(d1) - np.exp(-r*(T-t))*K*N(d2)
def black_scholes_put_price(S, t, K, T, r, sigma):
    d1, d2 = compute_d1_and_d2(S,t,K,T,r,sigma)
    return -S*N(-d1) + np.exp(-r*(T-t))*K*N(-d2)
import scipy.optimize
def compute_implied_volatility( V_KT, S0, K, T, r ):
    def f( sigma ):
        return black_scholes_put_price(S0,0,K,T,r,sigma)-V_KT
    sol = scipy.optimize.root_scalar(f, x0=0.01, x1=1.0, method='secant')
    return sol.root
S0=159.375
r=0.0638
sigma=0.35
gamma=1
compute_implied_volatility( 1.71, 159.375, 135, 0.22, 0.0638)
```

[4]: 0.35411798617295875

[5]:
```
#jump diffusion model calibration

# The function jump_diffusion_put_price is based on John.Armstrong's lecture
 ↪notes
#https://keats.kcl.ac.uk/course/view.php?id=93365.
def jump_diffusion_put_price( S, K, T, r, sigma, lbda, j):
    S = np.array(S)
    term = np.ones(S.shape)
    total = np.zeros(S.shape)
    n = 0
    mu_tilde = r + lbda*(1-j)
    while np.any(abs(term)>1e-7*abs(total)):
        V = black_scholes_put_price(S,0,
j**(-n)*K,T,mu_tilde, sigma)
        term = ((j*lbda*T)**n)/factorial(n) *np.exp( -j*lbda*T) * V
        total = total + term
        n = n+1
    return total
def iv_surface_jd( S0, r, sigma, lbda, j, K, t ):
    #K,t are vectors contain all grid points
    iv=np.zeros([12,7])
    prices = np.zeros([12,7])
    for m in range(12):
        for n in range(7):
            prices[m,n]=jump_diffusion_put_price( S0, K[m], t[n], r, sigma,
 ↪lbda, j)
    for p in range(12):
        for q in range(7):
            iv[p,q]=compute_implied_volatility(prices[p,q],S0,K[p],t[q],r)
    return iv,prices
def price_surface_jd( S0, r, sigma, lbda, j, K, t ):
    #K,t are vectors contain all grid points
    prices = np.zeros([12,7])
    for m in range(12):
        for n in range(7):
            prices[m,n]=jump_diffusion_put_price( S0, K[m], t[n], r, sigma,
 ↪lbda, j)
    return prices
```

[6]:
```
from math import *
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
def error2( sigma, lbda, j ):
    iv,prices = iv_surface_jd( S0, r, sigma, lbda, j, K, t )
```

```
    error=0
    for i in range(12):
        for j in range(7):
            error=error+(iv[i,j]-V[i,j]/100)**2
    return error
def error_price( sigma, lbda, j ):
    prices = price_surface_jd( S0, r, sigma, lbda, j, K, t )
    error=0
    for m in range(12):
        for n in range(7):
            error=error+(prices[m,n]-prices0[m,n])**2
    return error

def objective( x ):
    sigma = x[0]
    lbda = x[1]
    j = x[2]
    return error2( sigma, lbda, j)
S0=159.735
```

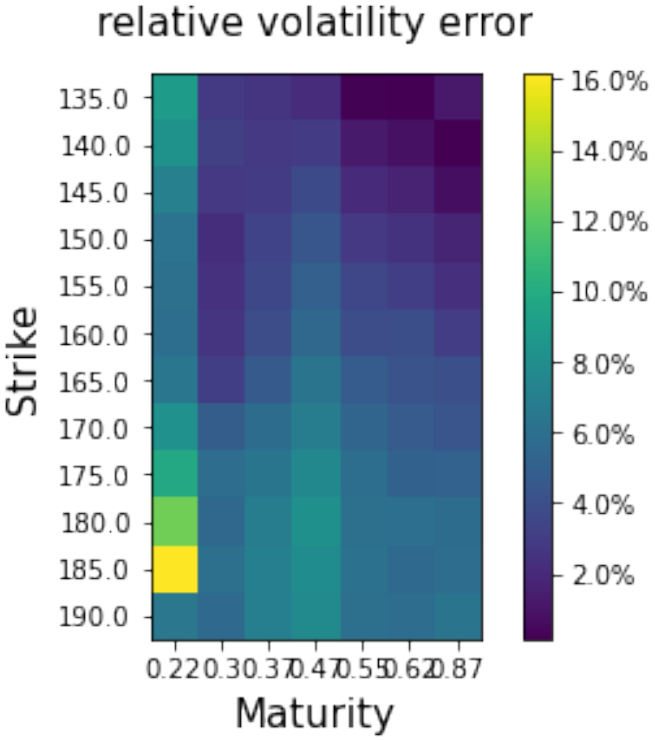
[0]:
```
#this is very slow
params_guess = np.array([0.36, 1, 0.90])
res = scipy.optimize.minimize(objective,params_guess,method='SLSQP')
sigma=res.x[0]
lbda=res.x[1]
j=res.x[2]
```

[10]:
```
```

[11]:
```
#calibration effectiveness of jump-diffusion model
iv,prices=iv_surface_jd( S0, r, sigma, lbda, j, K, t )
rela=abs(iv-V/100)/(V/100)
ax=plt.subplot(1,1,1)
plt.title("relative volatility error",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())

ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(t)
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

[11]:
```
Text(0, 0.5, 'Strike')
```


relative volatility error
Strike
135.0
140.0
145.0
150.0
155.0
160.0
165.0
170.0
175.0
180.0
185.0
190.0
0.22
0.30
0.37
0.47
0.55
0.62
0.87
Maturity
16.0%
14.0%
12.0%
10.0%
8.0%
6.0%
4.0%
2.0%

CEV_Heston_calibration

September 1, 2022

[1]:
```
import scipy
import scipy.interpolate
import scipy.sparse.linalg
from itertools import product
from math import *
import numpy as np
import scipy.sparse
import matplotlib.pyplot as plt
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
import scipy.stats
```

[2]:
```
#this function is based on github https://github.com/KNFO-MIMUW/Heston_model/
 ↪tree/master/project
import numpy as np
from scipy.integrate import quad

# Heston put price
def Heston_put_price(S0, v0, K, T, r, q, kappa, theta, sigma, rho, lmbda):
    p1 = p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, 1)
    p2 = p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, 2)
    call=S0 * np.exp(-q*T) * p1 - K * np.exp(-r*T) * p2
    put=call+K*np.exp(-r*T) - S0*np.exp(-q*T)
    return put

# Heston probability
def p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, j):
    integrand = lambda phi: np.real(np.exp(-1j * phi * np.log(K)) \
                                    * f_Heston(phi, S0, v0, T, r, q, kappa,␣
 ↪theta, sigma, rho, lmbda, j) \
                                    / (1j * phi))
    integral = quad(integrand, 0, 100)[0]
    return 0.5 + (1 / np.pi) * integral
```

```
# Heston characteristic function
def f_Heston(phi, S0, v0, T, r, q, kappa, theta, sigma, rho, lmbda, j):

    if j == 1:
        u = 0.5
        b = kappa + lmbda - rho * sigma
    else:
        u = -0.5
        b = kappa + lmbda

    a = kappa * theta
    d = np.sqrt((rho * sigma * phi * 1j - b)**2 - sigma**2 * (2 * u * phi * 1j␣
 ↪- phi**2))
    g = (b - rho * sigma * phi * 1j + d) / (b - rho * sigma * phi * 1j - d)
    C = (r - q) * phi * 1j * T + (a / sigma**2) \
            * ((b - rho * sigma * phi * 1j + d) * T - 2 * np.log((1 - g * np.
 ↪exp(d * T))/(1 - g)))
    D = (b - rho * sigma * phi * 1j + d) / sigma**2 * ((1 - np.exp(d * T)) / (1␣
 ↪- g * np.exp(d * T)))

    return np.exp(C + D * v0 + 1j * phi * np.log(S0))
```

[3]:
```
#Mote-Carlo-test for Heston:
import scipy
import scipy.stats
from math import *
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
def simulated_W1_W2(n_paths,n_steps,rho) :
    sigma3=np.array([[1,rho],[rho,1]])
    L=np.linalg.cholesky(sigma3)
    W1=np.zeros([n_paths,n_steps+1])
    W1[:,0]=0
    W2=np.zeros([n_paths,n_steps+1])
    W2[:,0]=0
    dt=T/n_steps
    np.random.seed(0)
    for i in range(n_steps):
        epsilon = L@np.random.randn( 2, n_paths )
        W1[:,i+1]=W1[:,i]+sqrt(dt)*(epsilon[0,:].T)
        W2[:,i+1]=W2[:,i]+sqrt(dt)*(epsilon[1,:].T)
    return W1,W2
def simulate_stocktwiddle_paths(kappa, theta, rho, v0, sigma, S0, r, T,␣
 ↪n_steps, n_paths):
    W1,W2=simulated_W1_W2(n_paths,n_steps,rho)
    v = np.zeros([n_paths, n_steps + 1])
```

```
    v[:, 0] = v0
    Stwiddle = np.zeros([n_paths, n_steps + 1])
    Stwiddle[:,0]=S0
    dt = T / n_steps
    times = np.linspace(0, T, n_steps + 1)
    for i in range(0, n_steps):
        v[:, i + 1] = v[:, i] + kappa * (theta-v[:,i]) * dt + sigma*(v[:,i]**(0.
↪5)) * (W2[:, i + 1] - W2[:, i])
        Stwiddle[:, i + 1] = Stwiddle[:, i] + (r*Stwiddle[:, i]) * dt +  (v[:
↪,i]**(0.5))*Stwiddle[:, i]* (W1[:, i + 1] - W1[:, i])
    return Stwiddle, times
def price_m(kappa, theta, rho, v0, sigma, S0, r, T, KK, n_steps, n_paths,␣
↪option_type):
    Stwiddle, times=simulate_stocktwiddle_paths(kappa, theta, rho, v0, sigma,␣
↪S0, r, T, n_steps, n_paths)
    if option_type=="call":
        payoff=np.maximum(Stwiddle[:,-1]-KK,0)
    elif option_type=="put":
        payoff=np.maximum(KK-Stwiddle[:,-1],0)
    else:
        print("error, enter a valid type call or put")
    p = 95
    alpha = scipy.stats.norm.ppf((1 - p / 100) / 2)
    price = np.mean(payoff)*np.exp(-r*T)
    sigma_sample = np.exp(-r * T) * np.std(payoff)
    lower = price + alpha * sigma_sample / np.sqrt(n_paths)
    upper = price - alpha * sigma_sample / np.sqrt(n_paths)
    return price, np.array((lower,upper))
```

[0]:

[4]:
```
T=1
S0=60
K=50
r=0.04
q=0
v0=0.1
rho=-0.7
kappa=2
theta=0.04
sigma=0.01
lmbda=0
print(Heston_put_price(S0, v0, K, T, r, q, kappa, theta, sigma, rho, lmbda))
print(price_m(kappa, theta, rho, v0, sigma, S0, r, T, K, 100, 100000, 'put'))
```

```
1.4777478129182313
(1.4659484771522722, array([1.44388714, 1.48800982]))
```

[5]:
```
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
excel_data = pd.read_excel('APPL.xlsx', header=0)
SURFACE = pd.read_excel('SURFACE.xlsx', header=0)
prices0 = np.array(pd.read_excel('prices.xlsx', header=0))
V=np.array(SURFACE)
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
```

[6]:
```
# Parameters
       # maturity
S0=159.375
r=0.0638    # risk-free interest rate
q = 0     # dividend rate
v0 = 0.04    # initial variance
rho = -0.7   # correlation between Brownian motions
kappa = 2    # mean reversion rate
theta = 0.04 # Long term mean of variance
sigma = 0.3  # volatility of volatility
lmbda = 0    # market price of volatility risk
```

[0]:
```
```

[0]:
```
```

[7]:
```
def price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho, lmbda):
    #K,t are vectors contain all grid points
    prices=np.zeros([12,7])
    for i in range(12):
        for j in range(7):
            prices[i,j]=Heston_put_price(S0, v0, K[i], t[j], r, q, kappa,␣
 ↪theta, sigma, rho, lmbda)
    return prices
def error_price( v0,kappa,theta,sigma,rho ):
    prices = price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho,␣
 ↪lmbda)
    error=0
    for m in range(12):
        for n in range(7):
            error=error+(prices[m,n]-prices0[m,n])**2
    return error
def objective( x ):
    v0 = x[0]
```

```
    kappa = x[1]
    theta = x[2]
    sigma= x[3]
    rho= x[4]
    return error_price( v0,kappa,theta,sigma,rho )
```

[8]:
```
def data_Heston(n):
    data=np.zeros((n,89))
    for i in range(n):
        data[i,0]=(np.random.rand())*0.5+0.1
        data[i,1]=np.random.rand()*2
        data[i,2]=np.random.rand()*0.08
        data[i,3]=np.random.rand()
        data[i,4]=np.random.rand()
        data[i,5:89]=price_surface_Heston(S0, data[i,0], K, t, r, q, data[i,1],␣
 ↪data[i,2], data[i,3], data[i,4], lmbda).reshape((1,84))
    return data
```

[0]:
```
```

[9]:
```
S0=159.735
params_guess = np.array([0.04, 1, 0.04, 0.3, -0.7])
res = scipy.optimize.minimize(objective,params_guess,method='SLSQP')
assert res.success
```

```
/tmp/ipykernel_541/3965259002.py:34: RuntimeWarning: overflow encountered in exp
  * ((b - rho * sigma * phi * 1j + d) * T - 2 * np.log((1 - g * np.exp(d *
T))/(1 - g)))
/tmp/ipykernel_541/3965259002.py:34: RuntimeWarning: invalid value encountered
in cdouble_scalars
  * ((b - rho * sigma * phi * 1j + d) * T - 2 * np.log((1 - g * np.exp(d *
T))/(1 - g)))
/tmp/ipykernel_541/3965259002.py:35: RuntimeWarning: overflow encountered in exp
  D = (b - rho * sigma * phi * 1j + d) / sigma**2 * ((1 - np.exp(d * T)) / (1 -
g * np.exp(d * T)))
/tmp/ipykernel_541/3965259002.py:35: RuntimeWarning: invalid value encountered
in cdouble_scalars
  D = (b - rho * sigma * phi * 1j + d) / sigma**2 * ((1 - np.exp(d * T)) / (1 -
g * np.exp(d * T)))
/tmp/ipykernel_541/3965259002.py:17: IntegrationWarning: The occurrence of
roundoff error is detected, which prevents
  the requested tolerance from being achieved.  The error may be
  underestimated.
  integral = quad(integrand, 0, 100)[0]
/tmp/ipykernel_541/3965259002.py:17: IntegrationWarning: The integral is
probably divergent, or slowly convergent.
  integral = quad(integrand, 0, 100)[0]
```

```
/tmp/ipykernel_541/3965259002.py:17: IntegrationWarning: The maximum number of
subdivisions (50) has been achieved.
  If increasing the limit yields no improvement it is advised to analyze
  the integrand in order to determine the difficulties.  If the position of a
  local difficulty can be determined (singularity, discontinuity) one will
  probably gain from splitting up the interval and calling the integrator
  on the subranges.  Perhaps a special-purpose integrator should be used.
  integral = quad(integrand, 0, 100)[0]
/tmp/ipykernel_541/3965259002.py:17: IntegrationWarning: The algorithm does not
converge.  Roundoff error is detected
  in the extrapolation table.  It is assumed that the requested tolerance
  cannot be achieved, and that the returned result (if full_output = 1) is
  the best which can be obtained.
  integral = quad(integrand, 0, 100)[0]
/tmp/ipykernel_541/3965259002.py:37: RuntimeWarning: overflow encountered in exp
  return np.exp(C + D * v0 + 1j * phi * np.log(S0))
/tmp/ipykernel_541/3965259002.py:14: RuntimeWarning: invalid value encountered
in cdouble_scalars
  integrand = lambda phi: np.real(np.exp(-1j * phi * np.log(K)) \
/tmp/ipykernel_541/1337715046.py:13: RuntimeWarning: overflow encountered in
double_scalars
  error=error+(prices[m,n]-prices0[m,n])**2
/tmp/ipykernel_541/3965259002.py:14: RuntimeWarning: overflow encountered in
cdouble_scalars
  integrand = lambda phi: np.real(np.exp(-1j * phi * np.log(K)) \
```

[10]:
```
v0=res.x[0]
kappa=res.x[1]
theta=res.x[2]
sigma=res.x[3]
rho=res.x[4]
print(v0,kappa,theta,sigma,rho)
```

```
0.027105267973522224 8.56482315711637 0.1617134409913782 2.86446472001963
-0.22117048338196574
```

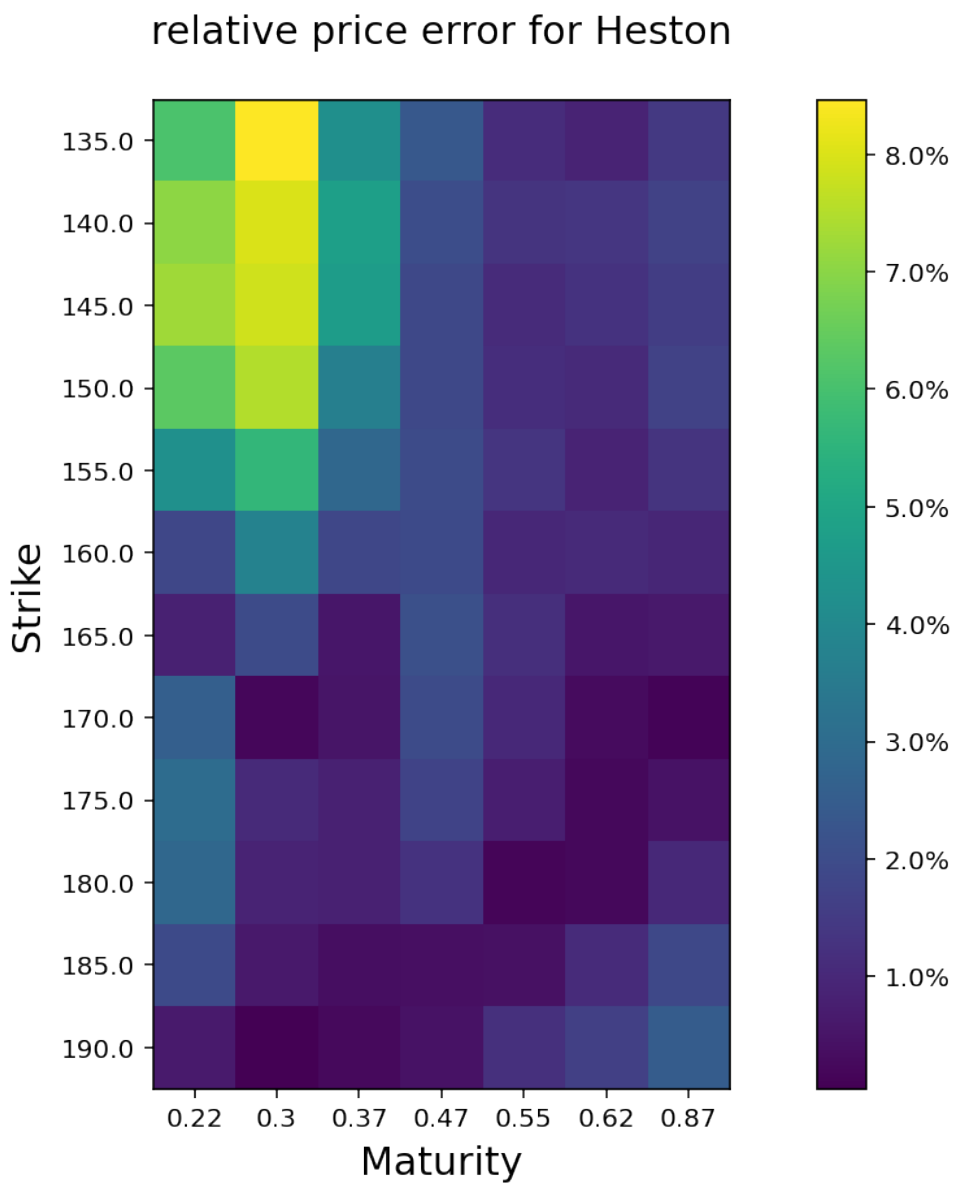
[11]:
```
prices=price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho, lmbda)
rela=abs(prices-prices0)/prices0
ax=plt.subplot(1,1,1)
plt.title("relative price error for Heston",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())

ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(t)
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
```

```
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

[11]: Text(0, 0.5, 'Strike')

[11]:



[15]:
```
#CEV closed formula
import numpy as np
import scipy.stats
def CEVabc(S0,K,T,r,sigma,gamma,typeofoption='put'):
    v=((sigma**2)*(np.exp(2*gamma*r*T-2*r*T)-1))/(2*r*(gamma-1))
```

```
    a=((K*np.exp(-r*T))**(2-2*gamma))/(((1-gamma)**2)*v)
    b=1/(1-gamma)
    c=S0**(2-2*gamma)/(((1-gamma)**2)*v)
    call=S0*(1-scipy.stats.ncx2.cdf(a,b+2,c))-K*np.exp(-r*T)*scipy.stats.ncx2.
 ↪cdf(c,b,a)
    put=K*np.exp(-r*T)*(1-scipy.stats.ncx2.cdf(c,b,a))-S0*scipy.stats.ncx2.
 ↪cdf(a,b+2,c)
    if typeofoption=='put':
        return put
    elif typeofoption=='call':
        return call
    else:
        print('enter call or put as type')
#test for CEV closed formula
def montecarlotest(S,K,T,r,sigma,gamma,N,M):
    A=np.zeros((N+1,M+1))
    A[:,0]=S
    dt=T/M
    for i in range(1,M+1):
        dW=np.random.randn(N+1)*(dt**0.5)
        A[:,i]=A[:,i-1]+r*A[:,i-1]*dt+sigma*(A[:,i-1]**gamma)*(dW.T)
    p = 95
    alpha = scipy.stats.norm.ppf((1 - p / 100) / 2)
    payoffs=np.maximum((K-A[:,M]),0)
    price = np.maximum((K-A[:,M]),0).mean()*np.exp(-r*T)
    sigma_sample = np.exp(-r * T) * np.std(payoffs)
    lower = price - alpha * sigma_sample / np.sqrt(N)
    upper = price + alpha * sigma_sample / np.sqrt(N)
    return price, np.array((upper,lower))
def price_surface_CEV(S0, K, t, r, sigma, gamma):
    #K,t are vectors contain all grid points
    prices=np.zeros([12,7])
    for i in range(12):
        for j in range(7):
            prices[i,j]=CEVabc(S0,K[i],t[j],r,sigma,gamma,typeofoption='put')
    return prices
def iv_surface_CEV(S0, K, t, r, sigma, gamma):
    #K,t are vectors contain all grid points
    prices=np.zeros([12,7])
    iv=np.zeros([12,7])
    for i in range(12):
        for j in range(7):
            prices[i,j]=CEVabc(S0,K[i],t[j],r,sigma,gamma,typeofoption='put')
    for m in range(12):
        for n in range(7):
            iv[m,n]=compute_implied_volatility( prices[m,n], S0, K[m], t[n], r )
    return iv
```

```
def error_price( sigma,gamma ):
    prices = price_surface_CEV(S0, K, t, r, sigma, gamma)
    error=0
    for m in range(12):
        for n in range(7):
            error=error+(prices[m,n]-prices0[m,n])**2
    return error
def error_iv( sigma,gamma ):
    iv = iv_surface_CEV(S0, K, t, r, sigma, gamma)
    error=0
    for m in range(12):
        for n in range(7):
            error=error+(iv[m,n]-iv0[m,n]/100)**2
    return error
def objective( x ):
    sigma = x[0]
    gamma = x[1]
    return error_price(sigma,gamma )
```

[16]:
```
print(CEVabc(1,1,1,0.05,0.1,0.9,typeofoption='put'))
print(montecarlotest(1,1,1,0.05,0.1,0.9,10000,100))
```

```
0.019279179851649286
(0.019426122604079235, array([0.01868235, 0.02016989]))
```

[49]:
```
S0=159.735
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
r=0.068
iv0=np.array(pd.read_excel('SURFACE.xlsx', header=0))
params_guess = np.array([0.3,0.9])
res = scipy.optimize.minimize(objective,params_guess,method='SLSQP')
assert res.success
```

```
/tmp/ipykernel_458/787799655.py:5: RuntimeWarning: overflow encountered in exp
  v=((sigma**2)*(np.exp(2*gamma*r*T-2*r*T)-1))/(2*r*(gamma-1))
```

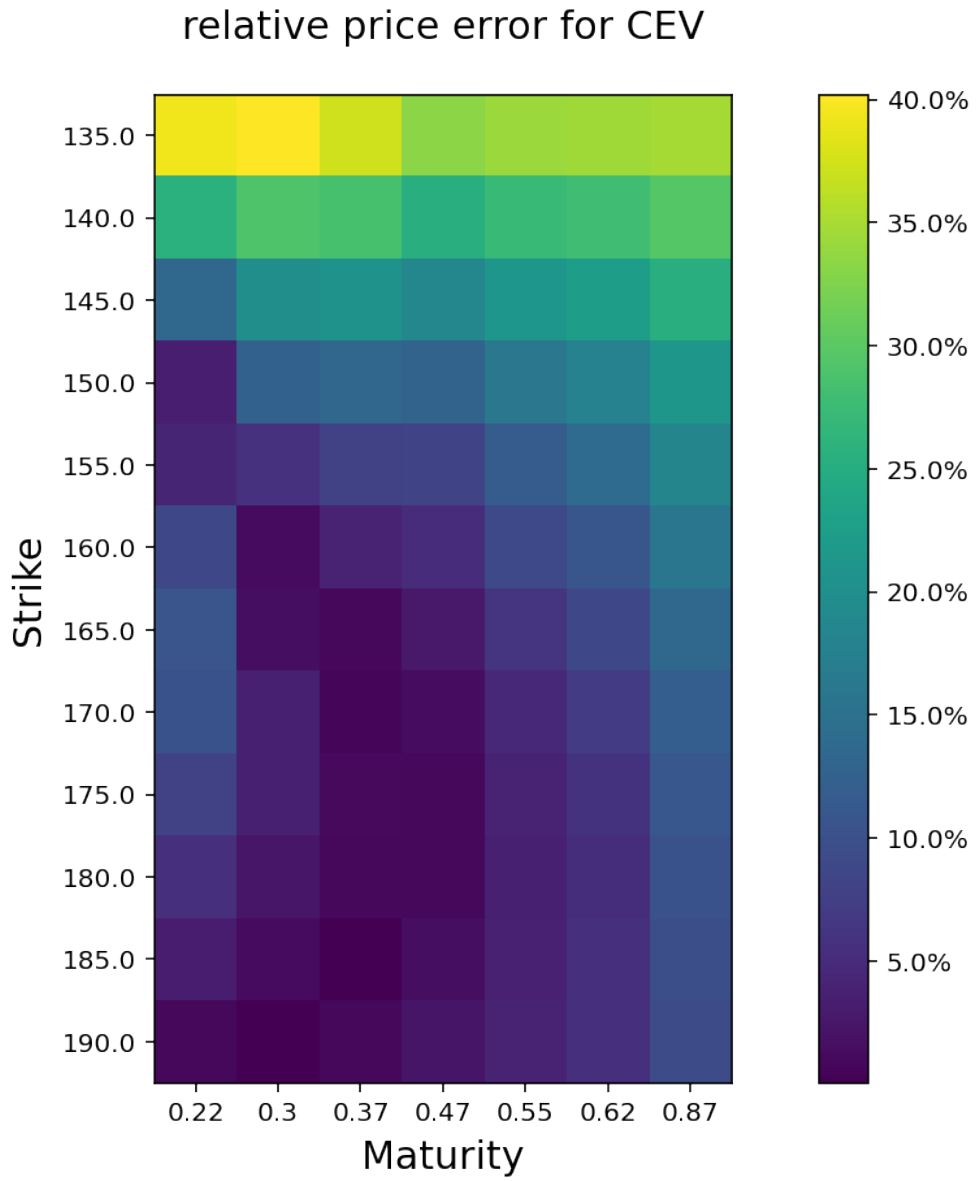
[50]:
```
print(res.x[0],res.x[1])
sigma=res.x[0]
gamma=res.x[1]
prices=price_surface_CEV(S0, K, t, r, sigma, gamma)
rela=abs(prices-prices0)/prices0
ax=plt.subplot(1,1,1)
plt.title("relative price error for CEV",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())
```

```
ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(t)
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

```
0.3101784721823033 0.9993143907455962
```

[50]: `Text(0, 0.5, 'Strike')`

[50]:



[0]:

deep_calibration_Hestonold

September 1, 2022

[0]:
```
import numpy as np
from scipy.integrate import quad
import scipy
import scipy.interpolate
import scipy.sparse.linalg
from itertools import product
from math import *
import numpy as np
import scipy.sparse
import matplotlib.pyplot as plt
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
import scipy.stats
q=0
lmbda=0
r=0.0638
S0=159.735
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
#Heston closed form pricing function is based on github https://github.com/
 ↪KNFO-MIMUW/Heston_model/tree/master/project
# Heston put price
def Heston_put_price(S0, v0, K, T, r, q, kappa, theta, sigma, rho, lmbda):
    p1 = p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, 1)
    p2 = p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, 2)
    call=S0 * np.exp(-q*T) * p1 - K * np.exp(-r*T) * p2
    put=call+K*np.exp(-r*T) - S0*np.exp(-q*T)
    return put

# Heston probability
def p_Heston(S0, v0, K, r, q, T, kappa, theta, sigma, rho, lmbda, j):
    integrand = lambda phi: np.real(np.exp(-1j * phi * np.log(K)) \
```

```
                                    * f_Heston(phi, S0, v0, T, r, q, kappa,␣
↪theta, sigma, rho, lmbda, j) \
                                    / (1j * phi))
    integral = quad(integrand, 0, 100)[0]
    return 0.5 + (1 / np.pi) * integral

# Heston characteristic function
def f_Heston(phi, S0, v0, T, r, q, kappa, theta, sigma, rho, lmbda, j):

    if j == 1:
        u = 0.5
        b = kappa + lmbda - rho * sigma
    else:
        u = -0.5
        b = kappa + lmbda

    a = kappa * theta
    d = np.sqrt((rho * sigma * phi * 1j - b)**2 - sigma**2 * (2 * u * phi * 1j␣
↪- phi**2))
    g = (b - rho * sigma * phi * 1j + d) / (b - rho * sigma * phi * 1j - d)
    C = (r - q) * phi * 1j * T + (a / sigma**2) \
            * ((b - rho * sigma * phi * 1j + d) * T - 2 * np.log((1 - g * np.
↪exp(d * T))/(1 - g)))
    D = (b - rho * sigma * phi * 1j + d) / sigma**2 * ((1 - np.exp(d * T)) / (1␣
↪- g * np.exp(d * T)))

    return np.exp(C + D * v0 + 1j * phi * np.log(S0))
def price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho, lmbda):
    #K,t are vectors contain all grid points
    prices=np.zeros([12,7])
    for i in range(12):
        for j in range(7):
            prices[i,j]=Heston_put_price(S0, v0, K[i], t[j], r, q, kappa,␣
↪theta, sigma, rho, lmbda)
    return prices
def error_price( v0,kappa,theta,sigma,rho ):
    prices = price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho,␣
↪lmbda)
    error=0
    for m in range(12):
        for n in range(7):
            error=error+(prices[m,n]-prices0[m,n])**2
    return error
def objective( x ):
    v0 = x[0]
    kappa = x[1]
```

```
    theta = x[2]
    sigma= x[3]
    rho= x[4]
    return error_price( v0,kappa,theta,sigma,rho )
```

[0]:
```
#create dataset
def data_Heston(n):
    data=np.zeros((n,89))
    for i in range(n):
        data[i,0]=(np.random.rand())*0.04+0.01
        data[i,1]=np.random.rand()*7+3
        data[i,2]=np.random.rand()*0.16+0.08
        data[i,3]=np.random.rand()*0.5
        data[i,4]=(np.random.rand()-0.5)*2
        data[i,5:89]=price_surface_Heston(S0, data[i,0], K, t, r, q, data[i,1],␣
 ↪data[i,2], data[i,3], data[i,4], lmbda).reshape((1,84))
        print(i)
    return data
dataset=data_Heston(10000)
np.savetxt('Hestondata',dataset)
```

[0]:
```
#scale method 2
xx=dataset[:,0:5]
yy=dataset[:,5:]
print(yy.shape)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    xx, yy, test_size=0.1, random_state=42)

x_train_transform=np.zeros((X_train.shape[0],X_train.shape[1]))
x_test_transform=np.zeros((X_test.shape[0],X_test.shape[1]))
for i in range(5):
    x_train_transform[:,i]=(2*X_train[:,i]-(X_train[:,i].max()-X_train[:,i].
 ↪min()))/(X_train[:,i].max()-X_train[:,i].min())
    x_test_transform[:,i]=(2*X_test[:,i]-(X_test[:,i].max()-X_test[:,i].min()))/
 ↪(X_test[:,i].max()-X_test[:,i].min())
y_train_transform=np.zeros((y_train.shape[0],y_train.shape[1]))
y_test_transform=np.zeros((y_test.shape[0],y_test.shape[1]))
for i in range(84):
  y_train_transform[:,i]=(2*y_train[:,i]-(y_train[:,i].max()-y_train[:,i].
 ↪min()))/(y_train[:,i].max()-y_train[:,i].min())
  y_test_transform[:,i]=(2*y_test[:,i]-(y_test[:,i].max()-y_test[:,i].min()))/
 ↪(y_test[:,i].max()-y_test[:,i].min())
```

```
(10000, 84)
```

[0]:
```
#construct the loss function
def ziweifunction(y_true,y_predict):
    if y_true<0.01:
        a=(abs(y_true-y_predict)/y_true)/5
    else:
        a=abs(y_true-y_predict)
    return a
def ziweifunction2(y_true,y_predict):
    a=abs((y_true-y_predict)/y_true+0.0001)*100
    return a
def newpercent(y_true,y_predict):
    a=log(abs((y_true-y_predict**2)))
    return a
from keras import backend as K
def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))
```

[0]:
```
#image-based training
import tensorflow.keras as keras
g_hat = keras.Sequential([
    keras.layers.Dense(30, activation="elu", input_shape=(5,)),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(30, activation="elu"),
    keras.layers.Dense(84, activation="linear")
]
)
g_hat.summary()
g_hat.compile(optimizer="adam", loss=root_mean_squared_error)
g_hat.fit(x_train_transform, y_train_transform, batch_size=80,␣
 ↪epochs=100,validation_data = (x_test_transform,y_test_transform))
```

[0]:
```
```

[32]:
```
#get neural network prediction function
def NN_heston(v0,kappa,theta,sigma,rho):
  para=np.array([[v0,kappa,theta,sigma,rho]])
  for i in range(5):
    para[0][i]=(2*para[0][i]-(X_train[:,i].max()-X_train[:,i].min()))/(X_train[:
 ↪,i].max()-X_train[:,i].min())
  prediction=g_hat.predict(para)[0]
  for j in range(84):
    prediction[j]=(prediction[j]*(y_train[:,j].max()-y_train[:,j].
 ↪min())+(y_train[:,j].max()-y_train[:,j].min()))/2
  prediction=prediction.reshape(12,7)
  return prediction
```

[29]:

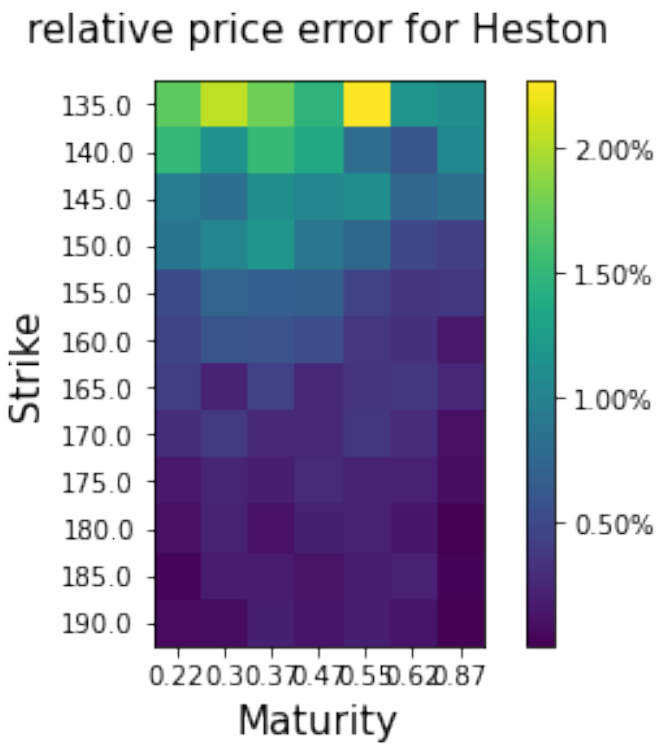
[33]:
```
#show the effectiveness of neural network and deep calibration
import pandas as pd
prices0 = np.array(pd.read_excel('prices.xlsx', header=0))
def error_NN(v0,kappa,theta,sigma,rho):
  error=0
  prices=NN_heston(v0,kappa,theta,sigma,rho)
  for i in range(12):
    for j in range(7):
      error=error+(prices[i,j]-prices0[i,j])**2
  return error
def objective( x ):
    v0 = x[0]
    kappa = x[1]
    theta = x[2]
    sigma= x[3]
    rho= x[4]
    return error_NN( v0,kappa,theta,sigma,rho )
params_guess = np.array([0.027,8.56,0.1617,0.3,-0.22])
res = scipy.optimize.minimize(objective,params_guess,method='SLSQP')
assert res.success
v0=res.x[0]
kappa=res.x[1]
theta=res.x[2]
sigma=res.x[3]
rho=res.x[4]
print(v0,kappa,theta,sigma,rho)
q=0
lmbda=0
r=0.0638
S0=159.735
K=np.linspace(135,190,12)
t=np.array([80,108,136,171,199,227,318])
t=np.round(t/365,2)
prices=NN_heston(v0,kappa,theta,sigma,rho)
prices1=price_surface_Heston(S0, v0, K, t, r, q, kappa, theta, sigma, rho,␣
 ↪lmbda)
rela=abs(prices-prices1)/prices1
ax=plt.subplot(1,1,1)
plt.title("relative price error for Heston",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())
ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(t)
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
```

```
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

```
0.04852283606452406 8.56 0.12846271730558342 0.2720433444741522 -0.22
```

[33]:
```
Text(0, 0.5, 'Strike')
```

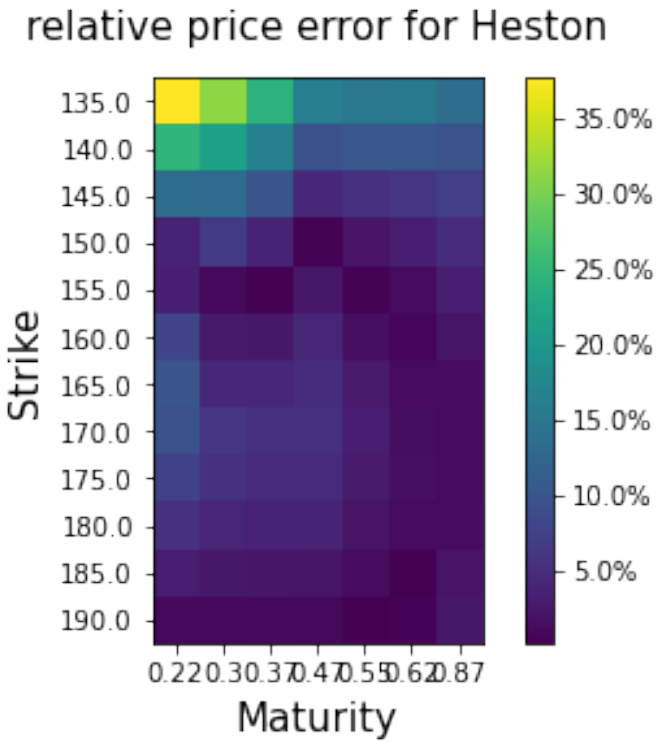
[33]:



[36]:
```
#deep calibration effectiveness
prices=NN_heston(0.04852283606452406,8.56,0.12846271730558342,0.
↪2720433444741522,-0.22)
rela=abs(prices-prices0)/prices0
ax=plt.subplot(1,1,1)
plt.title("deep calibration for Heston",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())
ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(t)
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

[36]: Text(0, 0.5, 'Strike')

[36]:



[0]:
```
from google.colab import drive
drive.mount('/content/drive')
```

2022-XGBoost

September 1, 2022

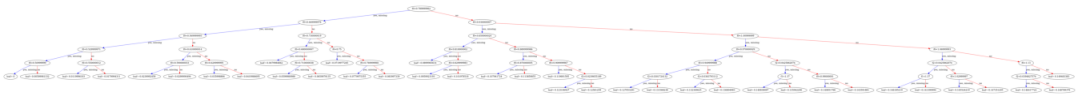
[1]:
```
from numpy import loadtxt
import numpy as np
from xgboost import XGBRegressor
from xgboost import plot_tree
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
validdata=loadtxt('data.txt')
count=0
for i in range(validdata.shape[0]):
  if validdata[i,4]>0.0001:
    count=count+1
validdata1=np.zeros((count,5))
row=0
for k in range(validdata.shape[0]):
  if validdata[k,4]>0.0001:
    validdata1[row,:]=validdata[k,:]
    row=row+1
print(count)

X=validdata1[:,0:4]
Y=validdata1[:,4]
seed=7
test_size=0.1
X_train, X_test, y_train, y_test=train_test_split(X, Y, test_size=test_size,␣
 ↪random_state=seed)
model=XGBRegressor()
model.fit(X_train,y_train)
y_pred=model.predict(X_test)
predictions=[round(value) for value in y_pred]
plot_tree(model)
plt.show()
```

```
/usr/local/lib/python3.8/dist-packages/xgboost/compat.py:93: FutureWarning:
pandas.Int64Index is deprecated and will be removed from pandas in a future
version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
```

```
1372859
```

[1]:



[2]:
```
import numpy as np
def scalingfunction(a,b,c,d,e,f,K,sigma):
    lamb=b/K
    S0=a/lamb
    gamma=f
    alpha=(((lamb**(1-gamma))*sigma)/e)**2
    r=d*alpha
    T=c/alpha
    return np.array((S0,T,r,gamma)),lamb
ans1,lamb1=scalingfunction(1.1,1,1,0.05,0.1,0.9,1,0.1)
ans2,lamb2=scalingfunction(1,1,1,0.05,0.1,0.9,1,0.1)
ans3,lamb3=scalingfunction(0.9,1,0.5,0.02,0.1,0.5,1,0.1)
ans4,lamb4=scalingfunction(2.6,2,1,0.1,0.2,0.8,1,0.1)
print(model.predict(np.array([ans1]))[0],model.predict(np.
 ↪array([ans2]))[0],model.predict(np.array([ans3]))[0],model.predict(np.
 ↪array([ans4]))[0]*lamb4)
```

```
0.0037226975 0.024033546 0.10026258 0.0025751590728759766
```

[3]:
```
import scipy
import scipy.sparse.linalg
import scipy.interpolate
from itertools import product
from math import *
import numpy as np
import scipy.sparse
import matplotlib.pyplot as plt
def bottom_boundary_condition( K, T, S_min, r, t):
    return np.ones(t.shape)*K

def top_boundary_condition( K, T, S_max, r, t):
    return np.maximum(K-S_max,0)

def final_boundary_condition( K, T, S ):
    return np.maximum(K-S,0)

def compute_abc( K, T, sigma, r, S, dt, dS, beta ):
    a = -sigma**2 * S**(2*beta)/(2* dS**2 ) + r*S/(2*dS)
    b = r + sigma**2 * S**(2*beta)/(dS**2)
    c = -sigma**2 * S**(2*beta)/(2* dS**2 ) - r*S/(2*dS)
    return a,b,c
```

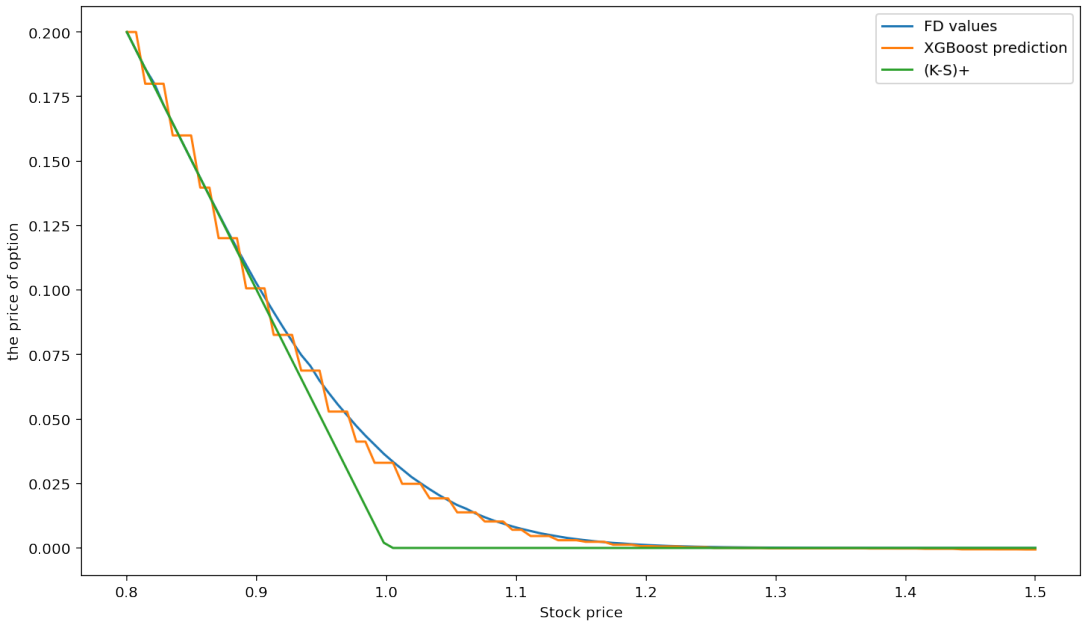
```
def compute_lambda( a,b,c ):
    return scipy.sparse.diags( [a[1:],b,c[:-1]],offsets=[-1,0,1],format='csr')
def compute_W(a,b,c, V0, VM):
    M = len(b)+1
    W = np.zeros(M-1)
    W[0] = a[0]*V0
    W[-1] = c[-1]*VM
    return W
def price_put_crank_nicolson( K, T, r, sigma, N, M, beta,
 ↪typeofoption="American"):
    dt = T/N
    S_min=0
    S_max=2
    dS = (S_max-S_min)/M
    S = np.linspace(S_min,S_max,M+1)
    t = np.linspace(0,T,N+1)
    V = np.zeros((N+1,M+1)) #...
    B=np.zeros(N)
    V[:,-1] = top_boundary_condition(K,T,S_max,r,t)
    V[:,0] = bottom_boundary_condition(K,T,S_max,r,t)
    V[-1,:] = final_boundary_condition(K,T,S)
    a,b,c = compute_abc(K,T,sigma,r,S[1:-1],dt,dS,beta)
    Lambda =compute_lambda(a,b,c) #...
    identity = scipy.sparse.identity(M-1)
    for i in range(N-1,-1,-1):
        Wt = compute_W(a,b,c,V[i,0],V[i,M])
        Wt_plus_dt = compute_W(a,b,c,V[i+1,0],V[i+1,M])
        if typeofoption=='American':
            V[i,1:M] = np.maximum(scipy.sparse.linalg.spsolve(identity+0.
 ↪5*Lambda*dt,(identity-0.5*Lambda*dt).dot(V[i+1,1:M]) -0.5*dt*(Wt_plus_dt +
 ↪Wt)),K-S[1:M])
            sign=(V[i-1,1:M]-(K-S[1:M]))>0
            for j in range(M-1):
                if sign[j]==True:
                    B[i-1]=(j+1)*dS+S_min
                    break
        elif typeofoption=='European':
            V[i,1:M] = scipy.sparse.linalg.spsolve(identity+0.
 ↪5*Lambda*dt,(identity-0.5*Lambda*dt).dot(V[i+1,1:M]) -0.5*dt*(Wt_plus_dt +
 ↪Wt))
        else:
            print('please enter a type from American or European')
    return V, t, S
def findprice(S,T,V):
    col=S*50
    row=(4-T)*(1000/4)
    upcol=ceil(S*50)
```

```
    downcol=floor(S*50)
    uprow=ceil((1000/4)*(4-T))
    downrow=floor((1000/4)*(4-T))
    if (col-int(col))<0.1 and (row-int(row))>0.1:
        f1=scipy.interpolate.interp1d(np.array([downrow,uprow]), np.
 ↪array([V[downrow,int(col)],V[uprow,int(col)]]))
        P=f1(row)
    elif (col-int(col))>0.1 and (row-int(row))<0.1:
        f2=scipy.interpolate.interp1d(np.array([downcol,upcol]), np.
 ↪array([V[int(row),downcol],V[int(row),upcol]]))
        P=f2(col)
    elif (col-int(col))<0.1 and (row-int(row))<0.1:
        P=(V[int(row),int(col)])
    else:
        f3=scipy.interpolate.interp2d(np.array([downrow,uprow]),np.
 ↪array([downcol,upcol]),np.
 ↪array([[V[downrow,downcol],V[downrow,upcol]],[V[uprow,downcol],V[uprow,upcol]]]))
        P=f3(row,col)
    return float(P)
```

[4]:
```
#gamma=1
SS0=np.linspace(0.8,1.5,100)
pred_gamma1=np.zeros(100)
for i in range(100):
    pred_gamma1[i]=model.predict(np.array([[SS0[i],1,0.02,1]]))
V,t,S=price_put_crank_nicolson( 1, 4, 0.01, 0.1, 1000, 100, 1␣
 ↪,typeofoption="American")
FDvalues=np.zeros(100)
for i in range(100):
    FDvalues[i]=findprice(SS0[i],1,V)
plt.plot(SS0,FDvalues,label='FD values')
plt.plot(SS0,pred_gamma1,label='XGBoost prediction')
plt.plot(SS0,np.maximum(1-SS0,0),label='(K-S)+')
plt.xlabel("Stock price")
plt.ylabel("the price of option")
plt.legend('A fan diagram of gamma=0')
plt.legend()
```

[4]: <matplotlib.legend.Legend at 0x7ff2956e44c0>

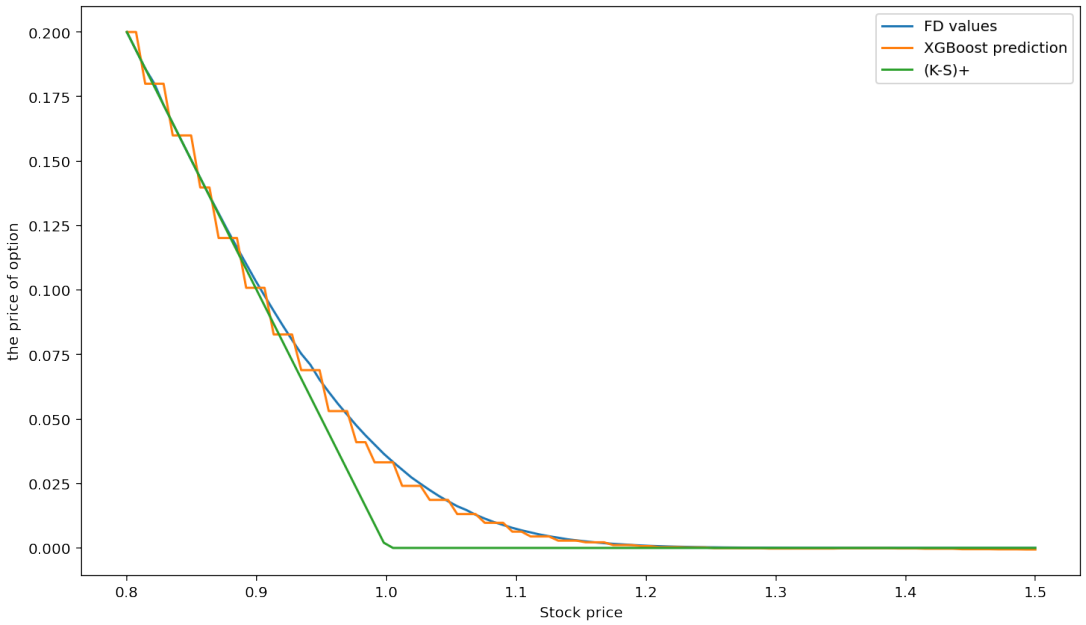
[4]:



[5]:
```
#gamma=0.5
SS0=np.linspace(0.8,1.5,100)
pred_gamma05=np.zeros(100)
for i in range(100):
    pred_gamma05[i]=model.predict(np.array([[SS0[i],1,0.02,0.5]]))
V,t,S=price_put_crank_nicolson( 1, 4, 0.01, 0.1, 1000, 100, 0.5␣
 ↪,typeofoption="American")
FDvalues=np.zeros(100)
for i in range(100):
    FDvalues[i]=findprice(SS0[i],1,V)
plt.plot(SS0,FDvalues,label='FD values')
plt.plot(SS0,pred_gamma05,label='XGBoost prediction')
plt.plot(SS0,np.maximum(1-SS0,0),label='(K-S)+')
plt.xlabel("Stock price")
plt.ylabel("the price of option")
plt.legend('A fan diagram of gamma=0')
plt.legend()
```

[5]: <matplotlib.legend.Legend at 0x7ff2672b9880>

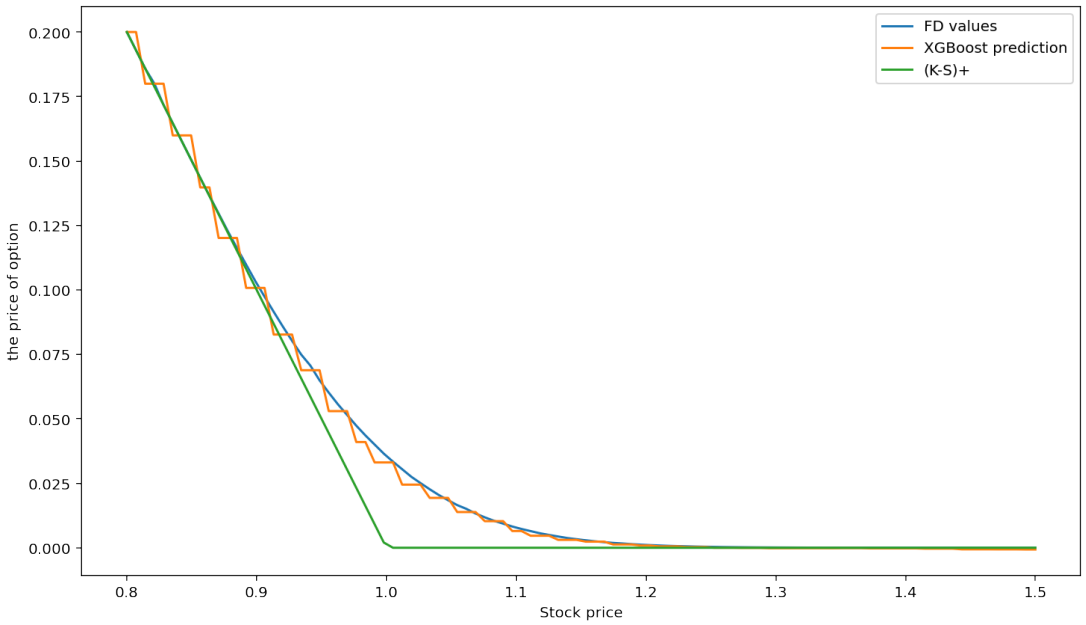
[5]:



[6]:
```
#gamma=0.9
SS0=np.linspace(0.8,1.5,100)
pred_gamma09=np.zeros(100)
for i in range(100):
    pred_gamma09[i]=model.predict(np.array([[SS0[i],1,0.02,0.9]]))
V,t,S=price_put_crank_nicolson( 1, 4, 0.01, 0.1, 1000, 100, 0.9␣
 ↪,typeofoption="American")
FDvalues=np.zeros(100)
for i in range(100):
    FDvalues[i]=findprice(SS0[i],1,V)
plt.plot(SS0,FDvalues,label='FD values')
plt.plot(SS0,pred_gamma09,label='XGBoost prediction')
plt.plot(SS0,np.maximum(1-SS0,0),label='(K-S)+')
plt.xlabel("Stock price")
plt.ylabel("the price of option")
plt.legend('A fan diagram of gamma=0')
plt.legend()
```

[6]: <matplotlib.legend.Legend at 0x7ff2671cfcd0>

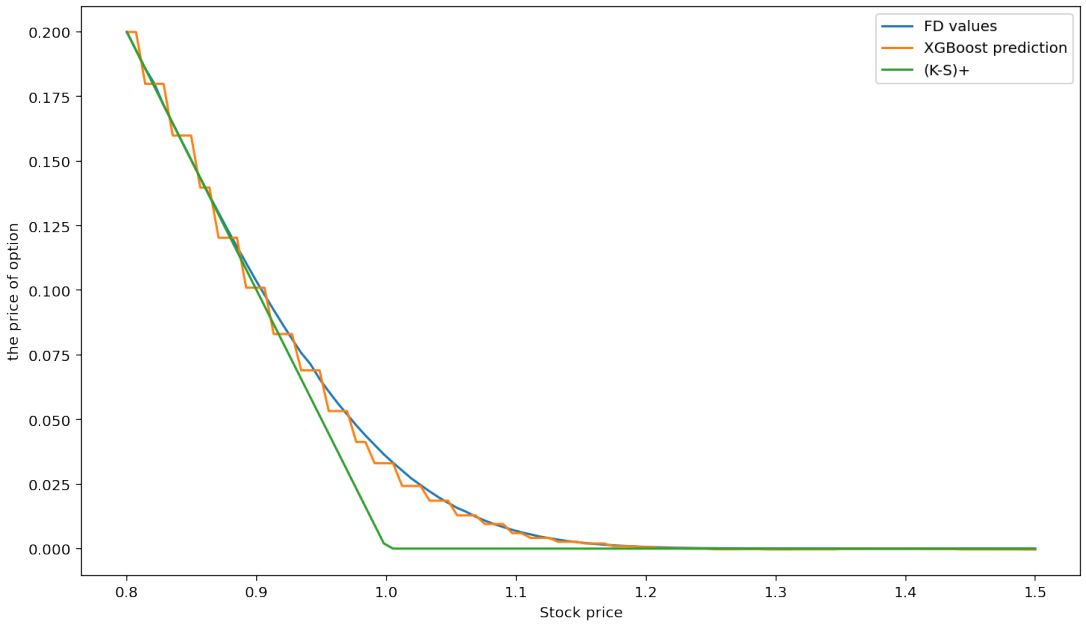
[6]:



[7]:
```
#gamma=0
SS0=np.linspace(0.8,1.5,100)
pred_gamma0=np.zeros(100)
for i in range(100):
    pred_gamma0[i]=model.predict(np.array([[SS0[i],1,0.02,0]]))
V,t,S=price_put_crank_nicolson( 1, 4, 0.01, 0.1, 1000, 100, 0␣
 ↪,typeofoption="American")
FDvalues=np.zeros(100)
for i in range(100):
    FDvalues[i]=findprice(SS0[i],1,V)
plt.plot(SS0,FDvalues,label='FD values')
plt.plot(SS0,pred_gamma0,label='XGBoost prediction')
plt.plot(SS0,np.maximum(1-SS0,0),label='(K-S)+')
plt.xlabel("Stock price")
plt.ylabel("the price of option")
plt.legend('A fan diagram of gamma=0')
plt.legend()
```

[7]: <matplotlib.legend.Legend at 0x7ff2671598b0>

[7]:



[8]:
```
import pandas as pd
K=np.linspace(145,200,12)
t=np.array([57,85,113,148,176,204])
t=t/365
S0=167.53
r=0.02
fixK=1
fixSigma=0.1
prices0 = np.array(pd.read_excel('AmericanAPPL.xlsx', header=0))
prices0=prices0[:,0:6]
```

[9]:
```
def price_XGB(S0,KK,T,r,sigma,gamma):
    para,lamb= scalingfunction(S0,KK,T,r,sigma,gamma,fixK,fixSigma)
    price=model.predict(np.array([para]))[0]*lamb
    return price
def XGB_price_surface(sigma,gamma):
    prices=np.zeros((len(K),len(t)))
    for i in range(len(K)):
        for j in range(len(t)):
            prices[i,j]=price_XGB(S0,K[i],t[j],r,sigma,gamma)
    return prices
def errorXGB(sigma,gamma):
    error=0
    prices=XGB_price_surface(sigma,gamma)
    for i in range(len(K)):
        for j in range(len(t)):
```

```
            error=error+(prices[i,j]-prices0[i,j])**2
    return error
```

[24]:
```
from numpy import unravel_index
vsigma=np.linspace(0,1,50)
vgamma=np.linspace(0,1,50)
errors=np.zeros((50,50))
for i in range(50):
    for j in range(50):
        errors[i,j]=errorXGB(vsigma[i],vgamma[j])
unravel_index(errors.argmin(), errors.shape)
```

```
/tmp/ipykernel_475/4084586274.py:6: RuntimeWarning: divide by zero encountered
in double_scalars
  alpha=(((lamb**(1-gamma))*sigma)/e)**2
```

[24]:
```
array([[1650.2839405 , 1650.2839405 , 1650.2839405 , …, 1651.62579443,
        1651.62579443, 1649.04694588],
       [1650.2839405 , 1650.2839405 , 1650.2839405 , …, 1651.62579443,
        1651.62579443, 1649.04694588],
       [1650.2839405 , 1650.2839405 , 1650.2839405 , …, 1651.62579443,
        1651.62579443, 1649.04694588],
       …,
       [1650.2839405 , 1650.2839405 , 1650.2839405 , …,  620.86868963,
         620.86868963,  619.17516765],
       [1650.2839405 , 1650.2839405 , 1650.2839405 , …,  620.86868963,
         620.86868963,  619.17516765],
       [1650.2839405 , 1650.2839405 , 1650.2839405 , …,  620.86868963,
         620.86868963,  619.17516765]])
```

[25]:
```
print(vsigma[10],vgamma[29])
```

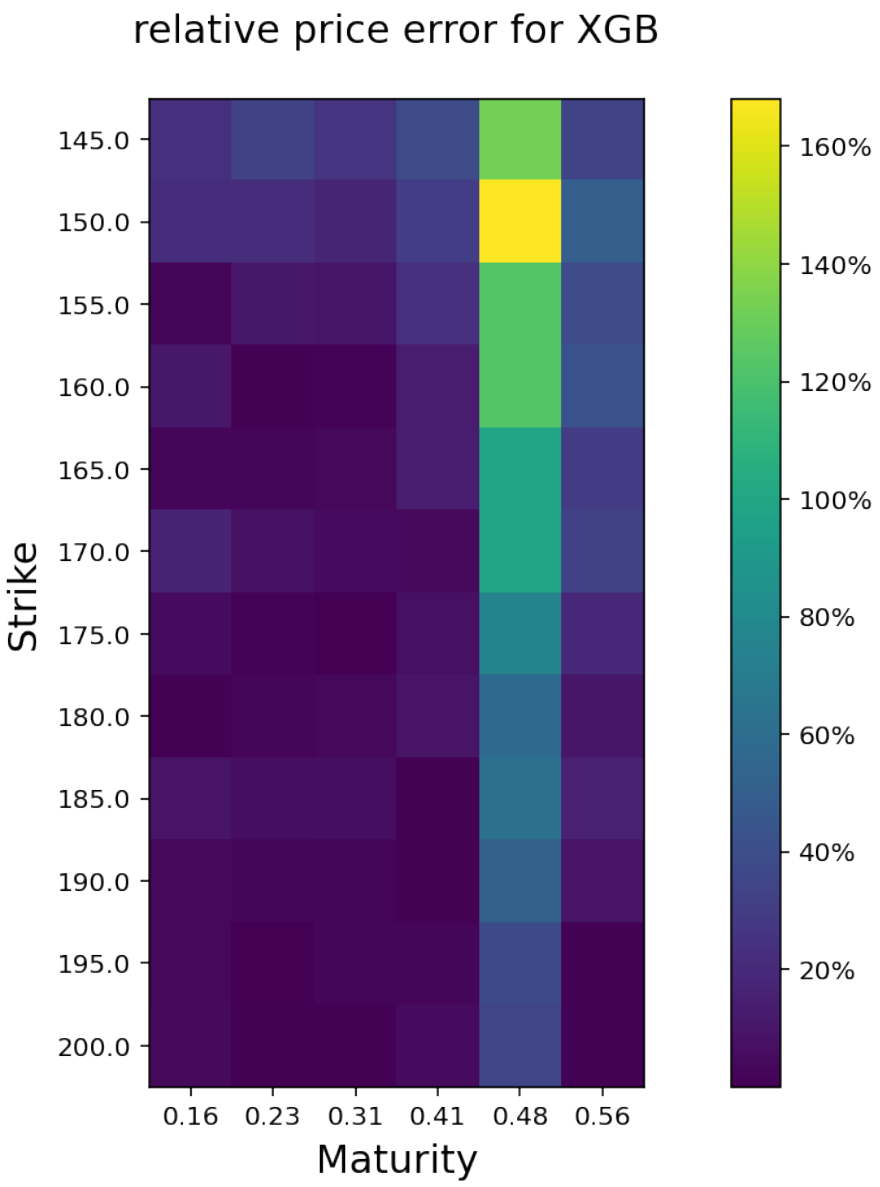
```
0.2040816326530612 0.5918367346938775
```

[21]:
```
import gzip
import pandas as pd
import numpy as np
import matplotlib.ticker as mtick
import csv
prices=XGB_price_surface(vsigma[10],vgamma[29])
rela=abs(prices-prices0)/prices0
ax=plt.subplot(1,1,1)
plt.title("relative price error for XGB",fontsize=15,y=1.04)
plt.imshow(rela*100)
plt.colorbar(format=mtick.PercentFormatter())
ax.set_xticks(np.linspace(0,len(t)-1,len(t)))
ax.set_xticklabels(np.round(t,2))
```

```
ax.set_yticks(np.linspace(0,len(K)-1,len(K)))
ax.set_yticklabels(K)
plt.xlabel("Maturity",fontsize=15,labelpad=5)
plt.ylabel("Strike",fontsize=15,labelpad=5)
```

[21]: Text(0, 0.5, 'Strike')

[21]:



[0]:

[0]:

Bibliography

[1] Mehdi.Tomas Blanka.Horvath, Aitor.Muguruza. *Deep Learning Volatility.* 2019.

[2] Mikko.Pakkanen Blanka.Horvath. *MACHINE LEARNING IN FINANCE.* 2022.

[3] Bowman. *Frank Introduction to Bessel Functions (Dover: New York, 1958).ISBN 0-486-60462-4.* 1958.

[4] Carlos.Guestrin. Chen.Tianqi. *XGBoost: A Scalable Tree Boosting System. http://arxiv.org/abs/1603.02754.* 2016.

[5] Cristin.Buescu. *King's Collega London FM07:lecture 2 https://nms.kcl.ac.uk/cristin.buescu/fm07/slides2.pdf.* 2022.

[6] Myron.Scholes Fischer.Black. *The Pricing of Options and Corporate Liabilities.* The Journal of Political Economy, 81(3), 637–654, 1973.

[7] AKHILESH GANTI. *Implied Volatility Investopedia https://www.investopedia.com/terms/i/iv.asp.* 2022.

[8] Hernandez. *Model calibration with neural networks.* 2017.

[9] J.Armstrong. *Numerical and Computational Methods in Finance https://keats.kcl.ac.uk/course/view.php?id=93365.* 2021.

[10] Jason.Brownlee. *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning.* 2017.

[11] Jeff.Heaton. *Probability Theory.* 2008.

[12] J.Jeffery. *Numerical Analysis and Scientific Computation. Addison Wesley. ISBN 0-201-73499-0.* 2004.

[13] Martin.Forde John.Armstrong. *MSc in Financial Mathematics, FM50/2021–2022,https://keats.kcl.ac.uk/pluginfile.php/8081264/mod_resource/content/3/F*

[14] JohnC.Cox. *The Constant Elasticity of Variance Option Pricing Model.* Journal of portfolio management, 1995.

[15] JohnC.Hull. *OPTIONS FUTURES AND OTHER DERIVATIVES.* Courier Kendallville, 2015.

[16] Markus.Riedle. *Probability Theory.* 2021.

[17] Martin.Forde. *FM02 lecture notes https://keats.kcl.ac.uk/course/view.php?id=93360.* 2021.

[18] optionspricing.suite. *https://play.google.com/store/apps/details?id=com.dipen.pricer.suitehl* 2016.

[19] R.Myiieiii P.Carr, R.Jarrow. *Alternative Cliaiacteiizatioiis of Amei.icari Put Options.* Mathcrriatical Finance 87-106., 1992.

[20] S.Howison P.Wilmott, S.Howson. *The mathematics of financial derivatives: a student introduction.* Cambridge university press, 1995.

[21] RobertC.Merton. The pricing of options and corporate liabilities. 1973.

[22] Steven.Heston. *A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options.* 1993.

[23] Wikipedia. *https://en.wikipedia.org/wiki/Almost_surely.* 2022.

[24] Wikipedia. *https://en.wikipedia.org/wiki/Constant_elasticity_of_variance_model.* 2022.

[25] xgboost.developers. *Introduction to Boosted Trees https://xgboost.readthedocs.io/en/stable/tutorials/model.html.* 2021.

[26] ZHAO.jing. *A Numerical Method for American Option Pricing under CEV Model.* 2007.