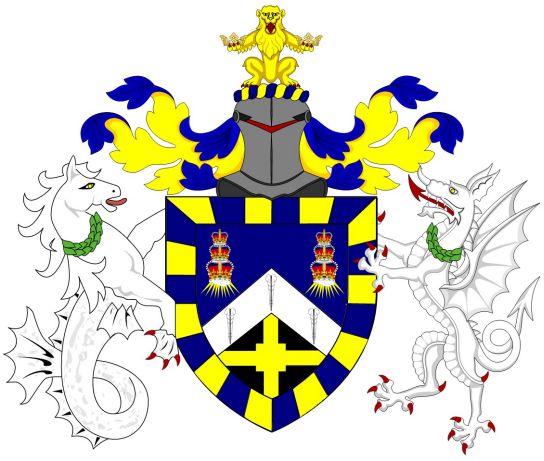
Data Analytics MSc Dissertation MTHM038, 2021

# Higher Order Least-Squares Monte Carlo Methods

A study on the construction of Monte Carlo based models to determine options prices

**Miles Frederick Jenkinson, ID 210405118**

Supervisors: Dr. Kathrin Glau & Dr Linus Wunderlich



A thesis presented for the degree of Master of Science in *MSc Data Analytics*

School of Mathematical Sciences

Queen Mary University of London

# Declaration of original work

This declaration is made on December 29, 2021.

**Student's Declaration:** I Miles Jenkinson hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text, nor has any part been written for me by another person.

Referenced text has been flagged by:

1. Using italic fonts, **and**
2. using quotation marks "...", **and**
3. explicitly mentioning the source in the text.

# Acknowledgements

I would like to give my sincerest thanks to my supervisors, Dr Linus Wunderlich and Dr Kathrin Glau, without whom it would not have been possible to complete this project. Their knowledge and guidance kept me on the right track and ensured that the project was both productive and enjoyable. I would also like to mention my family and partner Phattarasorn for their continued support and encouragement.

# Abstract

This dissertation paper consists of eight main chapters, an introduction to the main topics and a conclusion summary. The main focus being on different Monte Carlo based techniques in determining a fair price for standard and basket call options. Introducing relevant information around the behaviour and pricing of said options. Showing how we can model asset prices using the renowned Black-Scholes Model and Brownian Motion and applying these pricing techniques along side Monte Carlo and Least-Squares to determine a fair price for a call option based on these asset prices. With discussion on how best to calibrate the models and how they react to changes in parameters.

# Contents


**1 Introduction** **6**

1.1 Motivation for this work . . . . . 7

**2 Options** **8**

2.1 Call Option . . . . . 8

2.2 The Basket Case . . . . . 10

2.3 Fair Price . . . . . 10

**3 Black-Scholes Model and Brownian Motion** **12**

3.1 The Black-Scholes pricing formula . . . . . 13

3.2 Brownian Motion . . . . . 13

3.2.1 Geometric Brownian Motion . . . . . 15

3.3 The Stock Price formula . . . . . 16

**4 The Monte Carlo Method** **17**

4.1 Monte Carlo in practice . . . . . 17

4.2 Single Stock case . . . . . 18

**5 Least-Squares Monte Carlo** **22**

5.1 Polynomial Regression problem . . . . . 22

5.2 Least-Squares Monte Carlo in practice . . . . . 24

5.3 Single Stock case . . . . . 24

5.3.1 Approximating Expectations . . . . . 25

5.3.2 Moments for Expectation . . . . . 27

**6 Error Analysis** **32**

6.1 Distribution of Errors . . . . . 35

6.2 Number of Simulations effect on errors . . . . . 38

**7 Calibration of the Model** **41**

7.1 Interest Rates . . . . . 41

7.2 Volatility . . . . . 41

7.2.1 Historical Volatility . . . . . 42

7.2.2 Implied Volatility . . . . . 42

7.3 Correlation and Covariance . . . . . 43

7.3.1 Correlating data with the Cholesky Decomposition . . 44

**8 The Basket Case** **46**

8.1 Monte Carlo Basket Pricing . . . . . 46

8.2 Least-Squares Monte Carlo Basket Pricing . . . . . 48

8.2.1 Monomial Regression Problem . . . . . 48

8.2.2 Monomial Basis . . . . . 48

8.2.3 Monomial Basis Least-Squares Problem . . . . . 51

8.2.4 Approximating Expectations . . . . . 52

8.2.5 Higher Moments for Expectation . . . . . 52

8.2.6 Generating Higher Moments in Practice . . . . . 54

8.2.7 3D Plot of a 2 Dimensional Basket Case . . . . . 57

8.2.8 Least-Squares Monte Carlo Basket Pricing in Practice . 58

8.3 Comparison of Basket Case Models . . . . . 59

**9 Conclusion** **61**

# Chapter 1

# Introduction

In this dissertation we will explore how we can use Monte Carlo based methods in determining the fair price of a call option. Making use of the programming language Python along with packages such as NumPy [1], SciPy [2], Pandas [3], Matplotlib [4], Sympy [5], and Itertools [6] in order to perform our sampling, simulate stock prices, build our models to determine price and plot how they perform. Applying theorems and techniques from the studies of Linear Algebra, Machine Learning, Probability and Statistics.

Also included in this paper are some theories on what options and financial derivatives are and what goes into determining a fair price. Making sure to introduce models and ideas with care to cater to an audience who may be familiar with mathematics but not necessarily the financial modelling side.

In this paper we will use the concept of Brownian motion to simulate stock prices and from this build up a standard Monte Carlo model and a polynomial Least-Squares Monte Carlo model for the pricing of single stock call options and comparing the models with the Black-Scholes approach. After this we will be using similar techniques in order to price basket call options whilst making use of monomials to build our Least-Squares model by applying different theorems and knowledge gained from previous chapters.

The goal of this paper is to develop a deeper understanding on how to put together the building blocks of a pricing model, step by step and a look into how they perform against one another based on the parameters laid out.

## 1.1 Motivation for this work

Throughout finance we can see the use of Least-Squares Monte Carlo methods as an essential tool in both the pricing of derivative securities and risk management

Although we can see famous models such as the Black-Scholes model used in pricing simple call and put options, when we start to look at a basket option with different types of assets, we can see traditional methods fail. This is why we can use Monte Carlo methods to simulate multiple assets and then calculate the fair price explicitly.

Where we can see the standard Monte Carlo method fail is that it requires a large number of simulations in order to gain the desired level of accuracy in the pricing. This can become computationally expensive when attempting to price options for a wide range of different financial derivatives. We can therefore attempt to solve this problem through methods of least-squares and regularised regression, however this does not come without its difficulties. As we deal with large amounts of data and high-dimensions we face issues of a high number of basis functions and bottlenecks.

# Chapter 2

# Options

A *derivative* can be defined as a financial instrument whose value depends on (or derives from) the values of other, more basic underlying assets (for further reading on derivatives see [7]). A stock option is a derivative who's value is dependent on the price of a stock and in general there are two types of option, a *call option* and a *put option*. Within the scope of this paper we will just be focusing on the *call option*.

## 2.1 Call Option

A *call option* gives the holder the right to buy the underlying asset by a certain date for a certain price. The price within the contract is known as the *strike price* $K > 0$ and the date in the contract is known as the *expiration date* or *maturity* $T > 0$. There is also the distinct difference that *American options* can be exercised at any time up until the expiration date whilst *European options* can only be exercised on the expiration date itself. Within this paper we are assuming the style of European Options which means that the owner will exercise this right at maturity only if this results in a positive flux of money.

Consider the case that an investor buys a call option with the strike price of £100 to purchase 1 share of a certain stock. If the current stock price is £96 and the time until maturity $T$ is 1 year on and we have the price of the option contract as £5 then we have an initial investment of £5. If the stock price at maturity is less than the strike price of £100 then the option will expire worthless as it is not wise for the investor to exercise their right as it would be cheaper to purchase at the market price. In this case the investor has lost all of the £5 initial investment. However, if the stock price at maturity is higher than the strike price then the option will be exercised. Suppose, the stock price at maturity is £110, in this case the option has resulted in a positive flux of money. This is because the difference between the stock price £110 and the strike price £100 is a positive £10. Hence the investor is able to buy 1 share at £100 and can sell immediately at £110 for a £10 gain minus the initial investment of $£5$ for a total profit of $£5$. It is also worth noting that an option contract can be for the right to buy any amount of shares agreed and an investor can purchase more than one options contract.

Hence in the case of a call option that results in a negative flux of money i.e the stock price is higher than the strike price at maturity, then the option will not be exercised and the payoff is 0. However in the case that the call option results in a positive flux of money, i.e the stock price is lower than the strike price at maturity, then the option will be exercised and the payoff is the difference between the strike price and the stock price. So in the singular stock case our payoff becomes:

$$\max\{0, S(T) - K\} \tag{2.1}$$

Where $S(T)$ is the stock price at maturity and $K$ is our strike price.

## 2.2 The Basket Case

The *basket call option* is a financial derivative in which the underlying asset is a group or basket of assets or, in our case, stocks. The call option then gives the holder the right but not the obligation to buy this basket at the strike price at maturity.

Indeed this type of *exotic option* has similar characteristics to that of our standard call option. However the strike price is based on the weighted value of the underlying assets. Hence, we can introduce our vector of weights as $\boldsymbol{\omega} = (\omega_1, ..., \omega_d) \in \mathbb{R}^d$. The basket is then defined as the weighted arithmetic average of the $d$ stock prices (See [8] for more information on arithmetic weighting of asset prices). This will then allow us to define our payoff as

$$\max \left\{ 0, \sum_{i=1}^{d} \omega_i S_i(T) - K \right\} \tag{2.2}$$

In the case that the basket is made up of equally weighted stocks we have $\omega_i = \frac{1}{d}\ \forall i \in (1, 2, ..., d)$ a mean of the individual stock payoffs.

In most contributions from the literature on the valuation of such products, the underlying asset prices are assumed to follow lognormal processes. However, the celebrated Black-Scholes formula cannot be easily extended to the basket option case, because the lognormal distribution is not closed under summation. Several approaches have been proposed to solve the problem, one of which are Monte Carlo simulations (Read more on basket options pricing at [8]).

## 2.3 Fair Price

The *Fair Price* of an option is the price at which the buyer and the seller experience zero expected profit. It then makes sense that the fair price will

need to be equal to the expectation of the payoff at maturity. In the real world however there is a difference between the fair price and the *market price* and this can be for a number of reasons. The market price is guided by supply and demand but often people do not act as rational thinking economic or mathematical agents when making investment decisions. This difference in price either goes to the seller in the form of an overpriced contract or to the buyer as an underpriced one.

The key in determining the expectation of the payoff is where there are a range of mathematical models to guide us. However in general the formula for the fair price $\Pi$ of our basket call option is given by:

$$\Pi = e^{-rT} E\left(\max\left\{0, \sum_{i=1}^{d} \omega_i S_i(T) - K\right\}\right) \tag{2.3}$$

Here the fair price is the discounted expected value. We are using $e^{-rT}$ as our discount factor in order to calculate the future payoff under risk-neutral dynamics.

In is debatable as to how, why and whether to incorporate a risk-neutral measure but this is going beyond the scope of this project. However, we can try to motivate some reasoning when considering the no-arbitrage principle that there is no such thing as a 'free lunch'. In other words it is the idea that no one can make any profit without some level of risk.

In order to allow our world to become *risk-neutral* we take the initial prices of assets to be the same as those in the real world, and we adjust the drift rates of the prices equal to $r$. What we are doing is redistributing the real-world probabilities we attach to possible time $T$ stock price outcomes so that the mean of the final stock price distribution yields drift rate $r$ in the risk-neutral world (For more information on risk-neutral dynamics see [9]).

# Chapter 3

# Black-Scholes Model and Brownian Motion

The Black-Scholes or Black-Scholes-Merton model is a concept used throughout fundamental financial theory. The mathematical formula is used to estimate the price for the theoretical value of derivatives and other investment instruments, taking into account the impact of time and other risk factors. As defined in chapter 2, a derivative is a contract that derives its value from the performance of an underlying entity. In our case we can use Black-Scholes to calculate the theoretical price for a call option. Originally developed in 1973, Black-Scholes is still highly regarded as one of the best ways of pricing an options contract.

Under Black-Scholes we assume that all stocks pay no dividends throughout the life of the option and the risk-less interest rate $r$ is constant.

## 3.1 The Black-Scholes pricing formula

The Black-Scholes formula for the price at time 0 of a European call option on a non-dividend paying stock is

$$c = S_0 N(d_1) - K e^{-rT} N(d_2) \tag{3.1}$$

where

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

With $c$ as a our European call price, $S_0$ as the stock price at time 0, $K$ as the strike price of our option, $r$ as the short rate of interest and $\sigma$ as the stock price volatility. Then we have the function $N(x)$ which is the cumulative probability distribution function for a standardised normal distribution. In other words, it is the probability that a variable with a standard normal distribution (mean 0, standard deviation 1) will be less than $x$ (For more on the Black-Scholes pricing formula see [7]).

## 3.2 Brownian Motion

**Definition 3.2.1.** A standard Brownian motion is a random process $\mathbf{W} = W_t : t \in [0, \infty)$ with state space $\mathbb{R}$ that satisfies the following properties:

1. $W_0 = 0$ (with probability 1).
2. $\mathbf{W}$ has stationary increments. That is, for $s, t$ with $s < t$, the distribution of $W_t - W_s$ is the same as the distribution of $W_{t-s}$.
3. $\mathbf{W}$ has independent increments. That is, for $t_1, t_2, ..., t_n \in [0, \infty]$, the random variables $W_{t_1}, W_{t_2} - W_{t_1}, ..., W_{t_n} - W_{t_{n-1}}$ are independent.

4. $W_t$ is normally distributed with mean 0 and variance $t$ for each $t \in (0, \infty)$.

5. With probability 1, $t \to W_t$ is continuous on $[0, \infty)$.

In the multivariate case for an arbitrary $t > 0$, we have that $\mathbf{W}(t) = (W_1(t), ..., W_d(t))$ which is a $d$-variate random vector that is multivariate normally distributed with mean $\mu = 0 \in \mathbb{R}^d$ and a variance equal to $t$.

From a simple piece of python code we can see how Brownian motion can move over time.

```
T,steps,d=5,250,5 #Total time, number of steps and the number
    of variables
time=np.linspace(0,T,steps) #Defining our time steps across
   the total time
t=time[1]-time[0] #Change in time
W=np.random.normal(0,np.sqrt(t),size=(steps-1,d))#brownian
   motion change has a mean of 0 scaled by our change in time
    with n-1 size as brownian motion starts at 0. standard
   deviation of square root t hence variance=t, mean of 0.
W=np.insert(W,0,np.zeros(d),axis=0)
B0=np.zeros((1,d)) #Array of zeros with one row and d columns
B=np.concatenate((B0,np.cumsum(W,axis=0)),axis=0) #Here we
   can see our Brownian motion cumulative sum and contatenate
    across time
```

Then plotting our results we see:

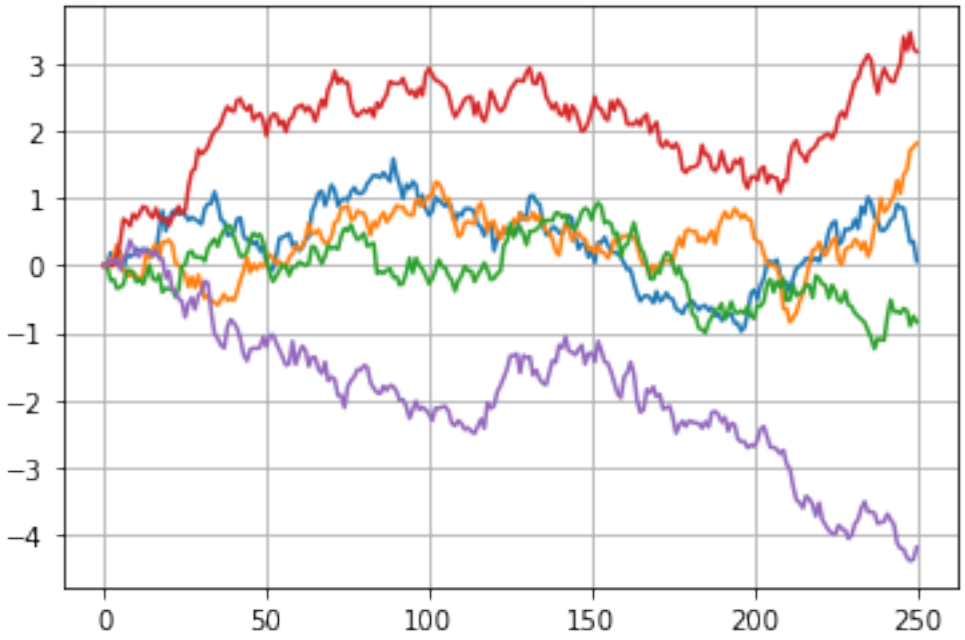



Figure 3.1: Cumulative Sum of Brownian Motion

Here we can see how 5 realisations of Brownian motion movement over the same time period and parameters can vary. Here we can see how the cumulative sum of each dimension moves over the time steps.

### 3.2.1 Geometric Brownian Motion

Within this paper we will be making use of a geometric Brownian motion process to calculate our stock prices.

**Definition 3.2.2.** Suppose that $\mathbf{W} = \{W_t : t \in [0, \infty)\}$ is standard Brownian motion and that $\mu \in \mathbb{R}$ and $\sigma \in (0, \infty)$. Let

$$X_t = \exp\left[\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right], \quad t \in [0, \infty)$$

The stochastic process $\mathbf{X} = \{X_t : t \in [0, \infty)\}$ is geometric Brownian motion with drift parameter $\mu$ and volatility parameter $\sigma$.

It is interesting to note that one of the reasons that geometric Brownian motion is so widely used in financial mathematics is that it is always positive.

## 3.3 The Stock Price formula

We can now make use of the geometric Brownian motion price process assumed by Black-Scholes to help price our call option. Under the risk neutral dynamics we assume a base observation at time 0, with a time now of $t$ on an option that matures at time $T$. We assume that the prices of our underlying stock(s) will grow through time $0 \leq t \leq T$ and have:

$$S_i(t) = S_i(0)e^{\sigma_i W_i(t) + (r - \frac{\sigma_i^2}{2})t} \tag{3.2}$$

For an individual stock price $S_i(t)$ of stock $i$ at time step $t$ for stocks $S_1, ..., S_d$. We require some initial values of $S_1(0), ..., S_d(0)$. In the beginning stages of our problem we introduce $\mathbf{W} \in \mathbb{R}^d$ as independant multivariate standard Brownian motion with mean $\mu = 0$ with variance $t$, $\mathbf{W}$'s exponentiation in the equation produces a GBM in stock price. To begin with we are implying that there are no correlations between stocks, however we know in practice this is not the case.

# Chapter 4

# The Monte Carlo Method

The Monte Carlo Method gets it's name from the famous Monte Carlo Casino based in the Principality of Monaco. They are related in the fact that within a casino there are games that you play with random outcomes that offer some kind of payoff. Monte Carlo methods are a way of using computers to generate a number $M$ of independent simulations of random variables to effectively 'brute force' a way to an expectation by finding the average across all of these simulations.

## 4.1 Monte Carlo in practice

In our case we generate $M$ realisations of our Brownian motion matrix $\hat{\mathbf{W}}_i^m(T)$ for $m = 1, ..., M$ and dimensions $i = 1, ..., d$ in order to generate our stock prices $S_0(T), ..., S_d(T)$ by using the stock price formula (3.2) then for each simulation we will find the payoff using formula (2.3) and then find the average across each simulation. In general we aim to get our optimum

fair price $\hat{\Pi}$ that approximates the true fair price .

$$\hat{\Pi} = e^{-rT}\frac{1}{M}\sum_{m=1}^{M}\max\left\{0, \sum_{i=1}^{d}\omega_i S_i(0)\exp\left\{\sigma_i \hat{W}_i^m(T) + (r - \frac{\sigma_i^2}{2})T\right\} - K\right\} \tag{4.1}$$

Some expectations of this method are due to the Law of Large Numbers, for large $M$ we expect convergence. Whilst the Central Limit Theorem suggests that as $M \to \infty$ our errors will become normally distributed as the errors reduce by a factor of $\frac{1}{\sqrt{M}}$.

## 4.2 Single Stock case

In the single stock case our fair price is calculated the same as (4.1) however now we do not need a weighted arithmetic average. Hence, our formula becomes

$$\hat{\Pi} = e^{-rT}\frac{1}{M}\sum_{m=1}^{M}\max\left\{0, S(0)\exp\left\{\sigma \hat{W}^m(T) + (r - \frac{\sigma^2}{2})T\right\} - K\right\} \tag{4.2}$$

We will do a theoretical option based on the Vanguard SP 500 ETF. One of the most popular funds that aims to track the Standard & Poor's 500 Index, which comprises five hundred large and mid-cap US stocks. These stocks are selected by a committee based on market size, liquidity and industry. We will look at producing a fair price for the call option of a single share of this ETF.(Setting up our example based on data from [10])

| Parameters | Values |
|---|---|
| Daily Treasury Yield Curve Rate rate, $r$ | 0.33 |
| Vanguard S&P 500 ETF current price, $S$ | 438.81 |
| Strike Price, $K$ | 490 |
| Time (years), $T$ | 1 |
| Volatility/Standard Deviation, $\sigma$ | 0.1092 |
| Simulations, $M$ | 1000 |

Table 4.1: Monte Carlo Vanguard SP 500 ETF Parameters

In this brief example we have made some assumptions about the data. For our volatility measure we have decided to use the previous year's standard deviation. However, in practice this is not the best measure to use but we will explore into the different uses of volatility later on in this paper. We assume the interest rate $r$ as the Daily Treasury Yield Curve Rate from the U.S. Department of the Treasury official website [11]. This is a helpful measure to us because it gives a constant rate which is assumed under our model. Whilst we know that interest rates are not constant, the Treasury rate gives us a good estimation to use.

We need to define a function that generates our Brownian motion samples $\mathbf{W}(T) = (W^1(T), ...W^M(T)) \in \mathbb{R}^M$

```
def Monte_Carlo_Sample_Generation(T,M):
    return np.random.normal(0,np.sqrt(T),M)
```

Listing 4.1: Generation of M samples from a uni-variate normal distribution for a mean of 0 and variance T

Then determine our payoffs

```
def get_payoffs(r,S,K,T,sigma,W_T):
    Smat=S*np.exp((sigma*W_T)+(r-0.5*sigma**2)*T) #Formula
   (2.1) being applied to each simulation
```

```
    option_profit=Smat-K #Our options profit that is stock price at maturity minus the strike price
    payoffs=np.maximum(option_profit,0) #Finding the max between our profit and 0
    return payoffs #Returning these payoffs
```

Listing 4.2: Calculating the payoff of a call option for given samples of Brownian motion

From this we will get an array of $M$ stock price simulations at maturity. Our theoretical profit is then our simulation price minus our strike price. The maximum function will return an array with every entry, either the maximum of the corresponding options profit or 0.

```
def Monte_Carlo_Basic(r,T,payoffs):
    fair_price=np.mean(payoffs)*np.exp(-r*T) #Our fair price for the option is the average of the payoffs multiplied by the discount factor
    return fair_price
```

Listing 4.3: Standard Monte Carlo fair pricing process

Then a simple Monte Carlo function takes the mean of all of our payoffs and multiplies it by our discount factor. If we apply these functions to our example above we see the following results:

```
W_T=Monte_Carlo_Sample_Generation(T,M)
payoffs=get_payoffs(r,S,K,T,sigma,W_T)
fair_price=Monte_Carlo_Basic(r,T,payoffs)
```

| Method | $M$ | $\hat{\Pi}$ |
|---|---|---|
| Monte Carlo | 1000 | 84.61 |

Table 4.2: Monte Carlo Fair Price Example caption.

In order to gauge how effective our model is we can compare it to that of the Black-Scholes model.

```
def Black_Scholes_Call(r,S,K,T,sigma):
    d1 = (np.log(S/K)+(r+sigma**2/2)*T)/(sigma*np.sqrt(T))
    d2 = d1 - sigma*np.sqrt(T)
    price=S*norm.cdf(d1, 0, 1) - K*np.exp(-r*T)*norm.cdf(d2,
   0, 1)
    return price #Simple Black Scholes Model that returns the
    price
```

Using the same parameters as before and we see our Black-Scholes price as

| Method | $\hat{\Pi}$ |
|---|---|
| Black-Scholes | 84.89 |

Table 4.3: Black-Scholes Model pricing

# Chapter 5

# Least-Squares Monte Carlo

One way in which we can increase the accuracy without increasing the number of samples generated by Monte Carlo is by introducing a regression problem. We can build a function that best approximates our realised valued for the payoff based on the Monte Carlo sample data and then compute the expectation of this approximation function. Then by multiplying this by our discount factor we will get our fair price as:

$$\hat{\Pi} = e^{-rT}\mathrm{E}[p_{\hat{c}}(\mathbf{W}(T))] \tag{5.1}$$

Where $p$ is our approximation function, $\hat{\mathbf{c}}$ is our vector of optimum weights and our $\mathbf{W}(T)$ are the samples of Brownian motion generated.

## 5.1 Polynomial Regression problem

In polynomial regression the goal is to fit polynomials of degree $d$ to our $\{x_i\}_{i=1}^{M}$ generated sample points. We can achieve this by creating an approximation model of a linear transformation of weights $\mathbf{c} \in \mathbb{R}^{d+1}$ and function

$p_c(x) : \mathbb{R}^d \to \mathbb{R}$ as:

$$p_c(x) := \langle \phi(x), c \rangle = \sum_{j=0}^{d} \phi(x)_j c_j = c_0 + c_1 x + c_2 x^2 + ... + c_d x^d \tag{5.2}$$

as we have $\phi$ that forms a basis for the space of polynomials of degree $d$

$$\phi(x) := (1 \quad x \quad x^2 \ \cdots \ x^d)^T$$

This allows us to create a polynomial basis matrix of the form:

$$\mathbf{V} := \begin{bmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_M)^T \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & & \vdots & \\ 1 & x_M & x_M^2 & \cdots & x_M^d \end{bmatrix} \in \mathbb{R}^{M \times (d+1)} \tag{5.3}$$

With the goal of finding the best $p_c(x)$ that approximates our function $f$ in the least squared sense. Setting up our least-squares minimisation problem

$$\min_{\mathbf{c} \in \mathbb{R}^{d+1}} \left\| \underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & & \vdots & \\ 1 & x_M & x_M^2 & \cdots & x_M^d \end{bmatrix}}_{:=\mathbf{V}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_d \end{bmatrix}}_{:=\mathbf{c}} - \underbrace{\begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_M) \end{bmatrix}}_{:=\mathbf{f}} \right\| \tag{5.4}$$

Where $\mathbf{f} \in \mathbb{R}^M$ are our realised values. Within (5.4) we have made use of the Euclidean Norm.

**Definition 5.1.1** (**Euclidean Norm**)**.** On the $n$-dimensional Euclidean Space $\mathbb{R}^n$, the intuitive notion of the length of the vector $\mathbf{x} = [x_1, x_2, ..., x_n]$ is captured by the formula:

$$||\mathbf{x}|| := \sqrt{\mathbf{x}^T \mathbf{x}} = \sqrt{x_1^2 + ... + x_n^2}$$

We wish to find the optimum vector $\hat{\mathbf{c}}$ that minimises the intuitive distance between our approximation and the realised values of $f(\mathbf{x})$ denoted as $\min_{\mathbf{c}\in\mathbb{R}^{d+1}} ||\mathbf{V}\mathbf{c} - \mathbf{f}||$.

In the case of Ordinary Least Squares Regression there are two methods that can be used to solve such a problem. In a more complicated setting one may choose to opt for *Gradient Descent* however in our case we will be using the solution of *Normal Equation*. Our normal equation can be seen as:

$$\mathbf{V}^T\mathbf{V}\hat{\mathbf{c}} = \mathbf{V}^T\mathbf{f} \tag{5.5}$$

Then by rearranging we can arrive at our least-squares general solution for our optimum $\hat{\mathbf{c}}$:

$$\hat{\mathbf{c}} = (\mathbf{V}^T\mathbf{V})^{-1}\mathbf{V}^T\mathbf{f} \tag{5.6}$$

(For more information on the method of solution by normal equation please read [12]). At this point we have our linear combination $p_{\hat{c}}(x) := \sum_{j=0}^{d} \phi(x)_j \hat{c}_j$ as our approximation of $f$.

## 5.2 Least-Squares Monte Carlo in practice

In practice we can use Least-Squares Monte Carlo to gain a more accurate approximation for our option fair price. In the case of options pricing we will be using $\mathbf{W}(T)$ as our random variable drawn from a Gaussian distribution with our realised values $\mathbf{f}$ as our payoffs as introduced in equation (5.4).

## 5.3 Single Stock case

We will be using the same stock case as in Table 4.1. Using our function Monte_Carlo_Sample_Generation defined in Listing (4.2) to generate our 1000 samples of our $\mathbf{W}(T)$ vector and then the function get_payoffs as seen in

Listing (4.3) to find the associated realised value payoffs for our Brownian Motion

```
W_T=Monte_Carlo_Sample_Generation(T,M) #Our Random Variable
payoffs=get_payoffs(r,S,K,T,sigma,W_T) #Defining our Payoffs
```

Then, to build our polynomial basis matrix $\mathbf{V}$ (5.3) we create a function polynomial_basis that takes our Brownian Motion vector $\mathbf{W}(T)$ and places it into a matrix of the form (5.2) where there are as many rows as there are samples $M$ and as many columns as the intended polynomial degree plus 1 $d+1$

```
def polynomial_basis(inputs, degree=1):
    basis_matrix = np.ones((len(inputs), 1)) #Creating an an
   array of ones with as many rows as samples and one column
    for counter in range(1, degree + 1):
        basis_matrix = np.c_[basis_matrix, np.power(inputs,
   counter)] #Adding columns to  this array with increasing
   powers of the corresponding row input
    return basis_matrix
```

Listing 5.1: Arranging samples into a Polynomial Basis Matrix/Array of a chosen degree

In order to solve for the optimum $\hat{\mathbf{c}}$ that minimises our cost function $\min_{\mathbf{c}\in\mathbb{R}^{d+1}} ||\mathbf{Vc}-\mathbf{f}||$ we must implement our *Normal Equation* standard solution (5.3) by using a SciPy function.

### 5.3.1 Approximating Expectations

In order to compute the expectation of our function $\mathrm{E}[p_{\hat{c}}(\mathbf{W}(T))]$ we can employ the use of two methods. The first being the *Law of the unconscious statistician* (LOTUS)

**Theorem 5.3.1.** *Let $X$ be a continuous random variable with probability density function $f_X(x)$ and $g:\mathbb{R}\to\mathbb{R}$ be a function. If $\int_{\infty}^{\infty}|g(x)|f_X(x)dx <$*

$\infty$, *then*

$$\mathrm{E}[g(X)] = \int_{\infty}^{\infty} g(x) f_X(x) dx \tag{5.7}$$

*For a full proof of this theorem please read at [13]*

As we know that our random variable $\mathbf{W}(T) \sim \mathcal{N}(0,\, T)$ is drawn from a normal distribution it means that we can approximate the expectation of our function by integrating the product of $p_{\hat{c}}(x)$ and the *probability density function* of the normal distribution. For $X \sim \mathcal{N}(\mu, \sigma^2)$ we have our probability distribution as:

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}, \quad \text{for all } x \in \mathbb{R} \tag{5.8}$$

Hence by integrating between the product of the standard normal $\mathcal{N}(0, 1)$ probability density function and our approximating function $p_{\hat{c}}(x)$ between $-\infty$ and $\infty$ our fair price becomes:

$$\hat{\Pi} = \frac{e^{-rT}}{\sigma\sqrt{2\pi}} \int_{\infty}^{\infty} \sum_{j=0}^{d} \phi(x)_j \hat{c}_j \; e^{-\frac{x^2}{2}} dx \tag{5.9}$$

We can implement this into python

```
def Monte_Carlo_Least_Squares_Polynomial_Int(r,T,W_T,P,
    payoffs):
     Matrix=polynomial_basis(W_T,P) #Putting W(T) into our
    basis matrix
     c=sp.linalg.lstsq(Matrix,payoffs)[0] #Finding optimum c
    that minimises our regression problem
     if P==1:
         fair_price=(1/np.sqrt(2*np.pi*T))*integrate.quad(
    lambda x: np.exp((-0.5)*(x/np.sqrt(T))**2)*(c[0]+c[1]*x),-
    np.inf, np.inf)[0]
     elif P==2:
         fair_price=(1/np.sqrt(2*np.pi*T))*integrate.quad(
    lambda x: np.exp((-0.5)*(x/np.sqrt(T))**2)*(c[0]+c[1]*x+c
    [2]*x**2),-np.inf, np.inf)[0] #Computing our Expectation
```

```
    of our function by integrating from -infinity to +infinity
     the pdf of the normal distribution multiplied by our
    polynomial function. We then multiply this by our discount
     factor
8    else:
9        print('Only valid for degrees 1 and 2')#Only able to
    integrate for set degree polynomials up to 2 which is
    limiting
10   return fair_price*np.exp(-r*T), Matrix, c
```

Listing 5.2: Creating an approximate function through Least-Squares regression of our samples and payoffs and then numerically integrating our function to find the expectation

It is understood that this method is computationally expensive because we are having to use SciPy's integration package to integrate between $\infty$ and $-\infty$ and this can lead to numerical rounding errors affecting the accuracy of our model.

Nevertheless by implementing this method with a polynomial degree of 2 we can see the following:

```
price_I, matrix_I, weights_I =
    Monte_Carlo_Least_Squares_Polynomial_Int(r,T,W_T,P,payoffs
    )
```

| Method | $M$ | P | $\hat{\Pi}$ |
|---|---|---|---|
| Least-Squares Monte Carlo | 1000 | 2 | 86.91 |

Table 5.1: Least-Squares Monte Carlo Fair Price Example caption.

### 5.3.2 Moments for Expectation

Due to the linearity of the expectation function we should be able to calculate the expectation of our function $\mathrm{E}[p_{\hat{c}}(x)]$ if we only knew the expectation of $\mathrm{E}[x^k]$. Because expectation is a linear operator it follows certain rules.

**Theorem 5.3.2** (Addition Rule for Expectations)**.** *Let $X$ and $Y$ be two random variables and assume that $X$ and $Y$ are either both discrete or both absolutely continuous. Suppose further that the expectations $\mathrm{E}[X]$ and $\mathrm{E}[Y]$ exists, then $\mathrm{E}[X+Y]$ exists and $\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y]$*

It follows that:

**Corollary 5.3.3** (to Theorem 5.4.4)**.** *Let $X_1, ..., X_n$ be $n$ random variables and let $a_1, ..., a_n$ be $n$ real constants. Suppose that $\mathrm{E}[X_j]$ exists for $j = 1, ..., n$; then*

$$\mathrm{E}\left[\sum_{j=1}^{n} a_j X_j\right] = \sum_{j=1}^{n} a_j \mathrm{E}[X_j]$$

(For proofs and further reading of Theorem 5.3.2 and Corollary 5.3.3 please see [14]).

Our fair price once again becomes:

$$\begin{aligned}\hat{\Pi} &= e^{-rT}\mathrm{E}[p_{\hat{c}}(x)] \\ &= e^{-rT}\mathrm{E}[\sum_{j=0}^{d} \phi(x)_j c_j] = \mathrm{E}[c_0 + c_1 x + c_2 x^2 + \cdots + c_d x^d] \\ &= e^{-rT}(c_0 + c_1\mathrm{E}[x] + c_2\mathrm{E}[x^2] + \cdots + c_d\mathrm{E}[x^d])\end{aligned}$$

In mathematical statistics a moment can be defined as:

**Definition 5.3.4.** The *$k$th moment* of a random variable $X$ is

$$m_k = \mathrm{E}[X^k], \quad k = 1, 2, 3, \cdots$$

as long as the expectation exists (For further reading please see [15]).

In order to calculate the $k$th moment of a normal distribution we can use the following theorem:

**Theorem 5.3.5.** *If $X$ has a normal distribution, the moments exist and are finite for any $k$ whose real part is greater than $-1$. For any non-negative integer $k$ then plain central moments are:*

$$\mathrm{E}[(X-\mu)^k] = \begin{cases} 0 & \text{if } k \text{ is odd,} \\ \sigma^k(k-1)!! & \text{if } k \text{ is even.} \end{cases}$$

*Here $n!!$ denotes the double factorial, that is, the product of all numbers from $n$ to 1 that have the same parity as $n$ (Definition can be found at[16]).*

This theorem 5.3.5 can be implemented into python using the SciPy special package factorial2

```
from scipy.special import factorial2
def generate_cent_moments(std,degree):
   #This std variable is the standard deviation of the random
    variable, in our case W(T)
    E=np.zeros(degree)
    for k in range(degree):
        if (k+1)%2!=0: #Check to see if it is odd or even
            E[k]=0
        else:
            E[k]=(std**(k+1))*factorial2(k)
            #in the case that p is even then we apply our
   formula above
    return E
```

Listing 5.3: Creating an array E that includes all of the moments for our 0 mean normal random variable up to a chosen degree

We can now use this new function to design a Least-Squares Monte Carlo model that implements moments to calculate the fair price.

```
def Monte_Carlo_Least_Squares_Polynomial(r,T,W_T,P,payoffs):
    Matrix=polynomial_basis(W_T,P) #Putting W(T) into our
   basis matrix
    c=sp.linalg.lstsq(Matrix,payoffs)[0] #Finding optimum c
   that minimises our regression problem
```

```
    E=generate_cent_moments(np.sqrt(T),P) #Generate our array
     of moments up to D
    fair_price=(c[0]+np.sum(c[1:]*E))*np.exp(-r*T)
    #Our intercept plus the summation of our weights
   multiplied by out expectations
    return fair_price, Matrix, c
    #Returning the fair price, the polynomial basis matrix
   and the optimum c
```

Listing 5.4: Least-Squares Monte Carlo with Moments

We run the same test again with a polynomial degree of 4 but this time using moments instead of integration to calculate expectations and we observe:

```
price, matrix, weights = Monte_Carlo_Least_Squares_Polynomial
   (r,T,W_T,D,payoffs)
```

| Method | $M$ | P | $\hat{\Pi}$ |
|---|---|---|---|
| Least-Squares Monte Carlo Moments | 1000 | 4 | 86.91 |

Table 5.2: Least-Squares Monte Carlo Moments Fair Price Example

By looking at a plot we can compare our realised value payoffs against our approximate function.

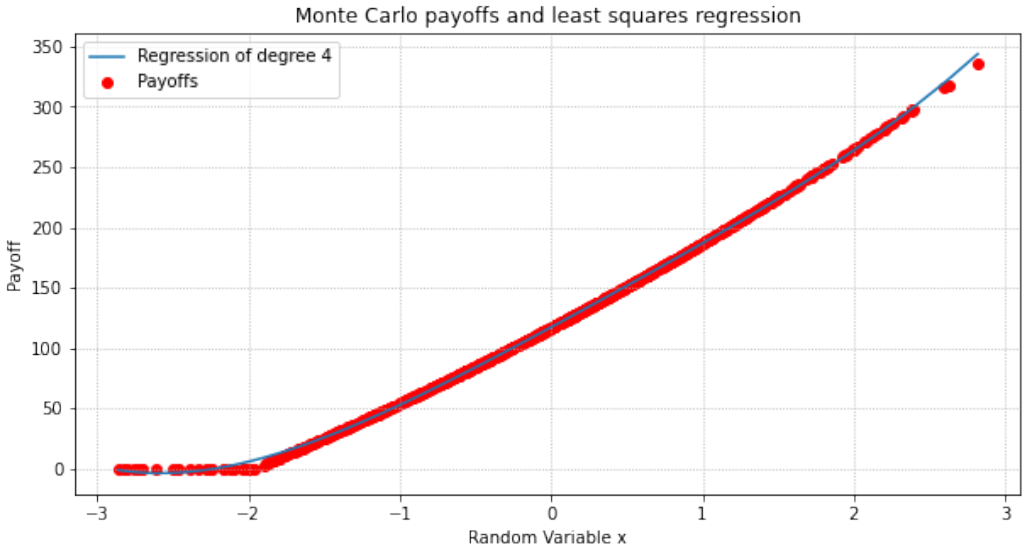



Figure 5.1: Payoffs with a least-squares regression of degree 4

Here we can observe a kink in the observed values as the simulated stock prices $S(T)$ reach the strike price $K$. The regression does its best to fit a curve to this kink.

# Chapter 6

# Error Analysis

Both the Monte Carlo and Least-Square method yield approximate results that incorporate some level of error. This error can usually be reduced by modifying parameters such as the number of simulations $M$ and degree of polynomial $d$, however in modifying these parameters there will be a knock on cost to the model.

In the single stock case we can use our Black-Scholes call option price as our accurate measure to compare our price to which will be denoted as $C_{BS}$. For our approximated fair price for our models we will denote Monte Carlo and Least-Squares Monte Carlo price as $\overline{\Pi}_{MC}$ and $\overline{\Pi}_{LSMC}$ respectively.

We can start to look at how the number of simulations $M$ will effect the accuracy of the models compared to the Black-Scholes price. We are using the same parameters as before but with only a varying number of simulations for each run of the loop.

```
M_range=np.linspace(1000,100000,100).astype('int') #Arranging
    a set of simulations
Monte_Carlo=np.zeros(len(M_range))
Monte_Carlo_Least_Squares=np.zeros(len(M_range))
for i,m in enumerate(M_range):
```

```
5     W_T=Monte_Carlo_Sample_Generation(T,m) #Our Random
    Variable
6     payoffs=get_payoffs(r,S,K,T,sigma,W_T) #Defining our
    Payoffs
7     Monte_Carlo[i]=Monte_Carlo_Basic(r,T,payoffs)  #Adding
    our standard Monte Carlo fair price for every simulation
8     fair_price,_,_=Monte_Carlo_Least_Squares_Polynomial(r,T,
    W_T,P,payoffs)
9     Monte_Carlo_Least_Squares[i]=fair_price #Adding our Least
     Squares Monte Carlo fair price for every simulation
```

Listing 6.1: Generating standard Monte Carlo and Least-Squares Monte Carlo fair prices across a range of simulations

By arranging a set of 100 simulations that range from 1000 to 100,000 we can see how the Least-Squares Monte Carlo method converges at a faster rate to the Black-Scholes price with a tighter variance.

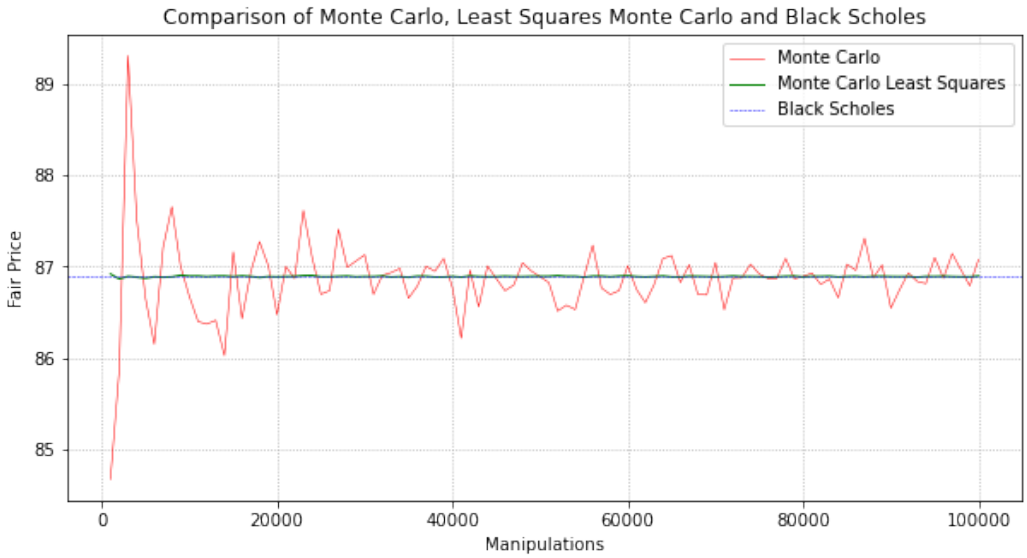



Figure 6.1: Comparison of the fair price models

To take a look at the error we use an absolute measure which is

$$\epsilon_{MC} = |\overline{\Pi}_{MC} - C_{BS}|$$

for the Monte Carlo error, and

$$\epsilon_{LSMC} = |\overline{\Pi}_{LSMC} - C_{BS}|$$

for the Least-Squares Monte Carlo error. Then we can create a plot that looks at these errors for both our models.

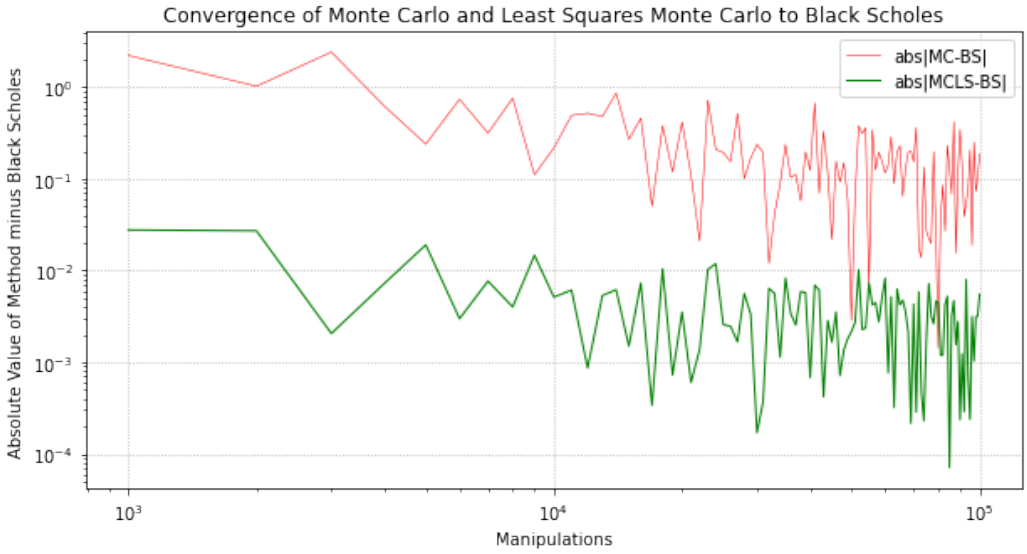



Figure 6.2: Evolving Errors

In this plot we are using a logarithmic scale on both the x and y axis. The reason for this is that we see an exponential decrease in the value of the errors as the number of simulations increase. It is important to note that whilst the errors decrease to a very small value they never reach zero for either method. There seems to be a trade off between the value that is deemed to be an acceptable level of accuracy for the fair price and the amount of computational time and power required for the desired accuracy.

## 6.1 Distribution of Errors

Given a standard Monte Carlo estimator $\overline{\nu}$ as the average of many individual simulations of a random variable $V$, as seen in (4.1).

$$\overline{\nu}_M = \frac{1}{M}\sum_{i=1}^{M}\nu_i$$

We know that for each individual evaluation of the estimator itself, for large M, behaves approximately like a normal variate by the virtue of the *central limit theorem* as the errors are also obtained with the same $M$ random simulations. This implies that the errors are also to be normally distributed (For more information on the central limit theorem see [17]).

Using the same parameter setup as in table 4.1 we can run a test in python to check these claims for a polynomial degree $P$ 4, 10000 simulations $M$ and 1000 runs of our test $n$. By creating a function Error_Generator_MC_MCLS we can generate $n$ amount of errors where each run of the test is for a certain amount $M$ of simulations

```
def Error_Generator_MC_MCLS(r,S,K,T,sigma,P,M,n,MC=True,MCLS=
    False):
     errors=np.zeros(n) #Assign an array of n zeros
     BS=Black_Scholes_Call(r,S,K,T,sigma) #Find the Black-
    Scholes price
     for i in range(n):
          W_T=Monte_Carlo_Sample_Generation(T,M) #For each run
    generate our brownian motion
          payoffs=get_payoffs(r,S,K,T,sigma,W_T) #Find the
    associated payoffs
          if MC is True and MCLS is False: #In the case of
    Monte Carlo test
               errors[i]=Monte_Carlo_Basic(r,T,payoffs)-BS #Add
    the error for this run to the array
          elif MC is False and MCLS is True: #In the case of
```

```
    Least-Squares Monte Carlo test
10              MCLS_Price,_,_=
    Monte_Carlo_Least_Squares_Polynomial(r,T,W_T,P,payoffs)
11              errors[i]=MCLS_Price-BS #Add the error for this
    run to the array
12          elif MC==MCLS:
13              print('One of MC and MCLS needs to be True and
    the other False')
14              break
15      return errors #Return the array of errors
16
17 models=['Monte Carlo','Monte Carlo Least Squares']
18 errors_MC=Error_Generator_MC_MCLS(r,S,K,T,sigma,D,M,n,True,
    False)
19 errors_MCLS=Error_Generator_MC_MCLS(r,S,K,T,sigma,D,M,n,False
    ,True)
```

Listing 6.2: Generating Errors by comparing our model's prices with that of Black-Scholes

We can then plot these errors for both methods in the form of a histogram along with an overlaying of a normal distribution curve.

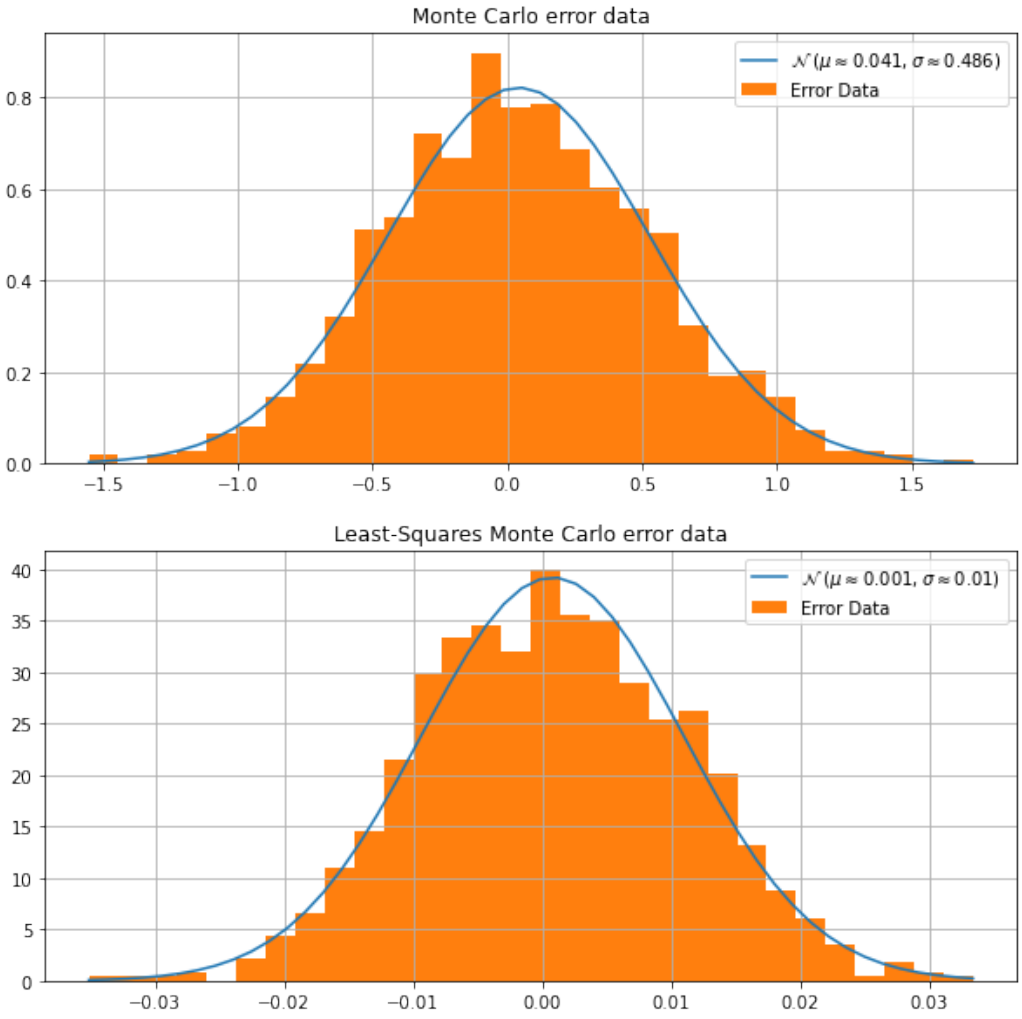



Figure 6.3: Histogram of Errors

The normal distribution curve is fitted to the data by using a domain that ranges from the minimum to maximum error along with the mean and standard deviation of the sample error data.

From figure 6.3 we can see that the histogram shows very little deviation of the sample distribution (in orange) from the theoretical bell curve distribution (blue line). This is the first indication that the data is in fact

normally distributed.

We can once again see this by looking at a QQ plot of the errors in both cases.

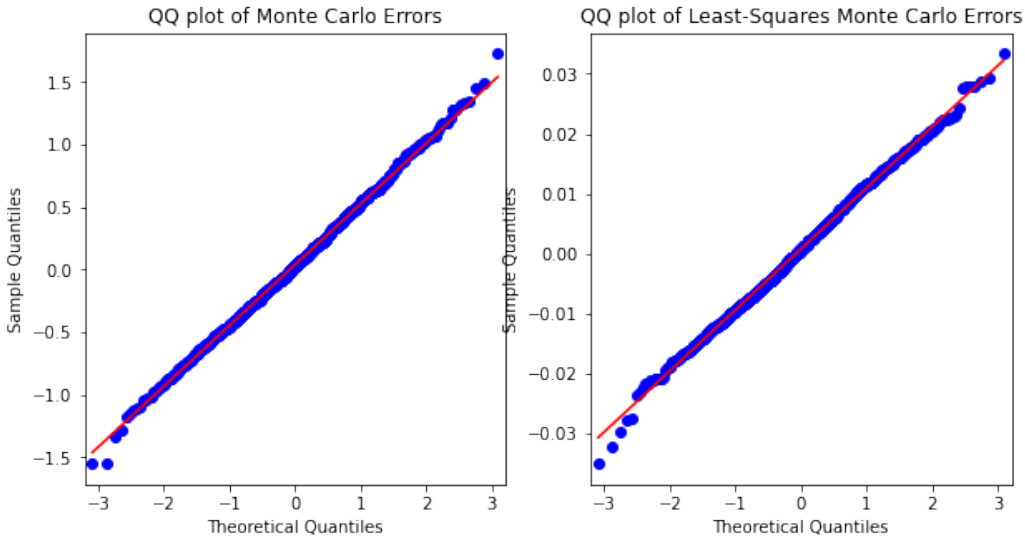



Figure 6.4: QQ Plots of Errors

QQ Plot stands for Quantile vs Quantile Plot, plotting theoretical quantiles against the actual quantiles of our variables.

As the quantiles of our variable seem to be in line with the theoretical normal quantiles it tells us that we have a normal distribution for the Errors of both of our Monte Carlo methods

## 6.2 Number of Simulations effect on errors

As our errors seem to behave as a normal distribution by the central limit theorem we can expect our Monte Carlo method to display $\frac{1}{\sqrt{M}}$ convergence to our Black-Scholes price adjusted by some drift parameter.

To see if our models meet this expectation we can plot the upper critical point of a 95% confidence interval based on the $t$ statistic and see how this changes as the number of simulations increases

```
M_range=np.linspace(10,10000,100).astype('int') #Arranging a set of simulations
Confidence_intervals=[np.zeros((len(M_range),2)),np.zeros((len(M_range),2))] #Creating a list for the confidence intervals
for i,m in enumerate(M_range):
    errors=[]
    errors.append(Error_Generator_MC_MCLS(r,S,K,T,sigma,P,m,n,MC=True,MCLS=False))
    errors.append(Error_Generator_MC_MCLS(r,S,K,T,sigma,P,m,n,MC=False,MCLS=True))
    #Generate the errors for each number of simulations
    means=np.mean(errors,axis=1)
    stds=np.std(errors,axis=1)
    for j in range(2):
        Confidence_intervals[j][i]=means[j]+(t.isf([0.975,0.025],n-1))*(stds[j]) #Add the upper and lower bound critical points
```

Listing 6.3: Confidence Intervals

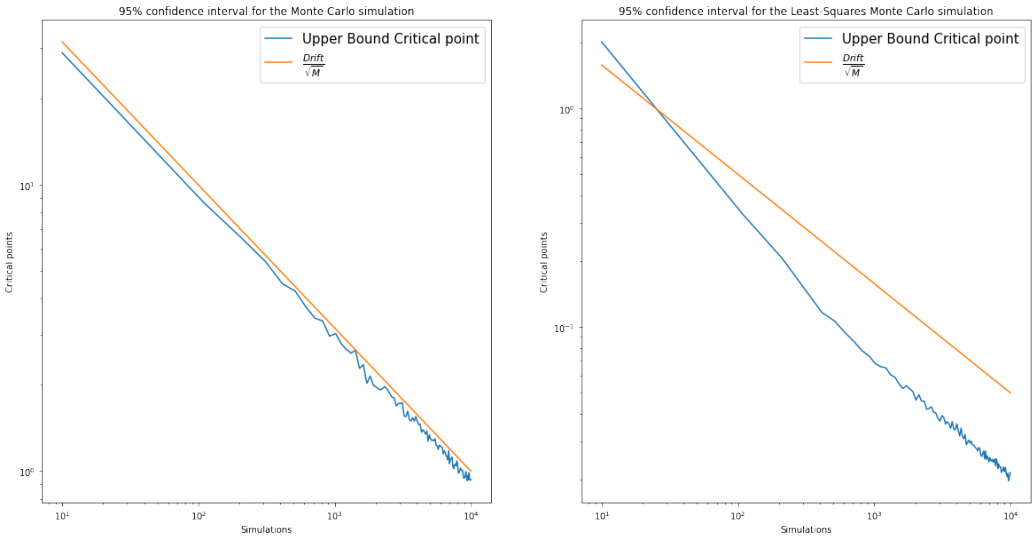



Figure 6.5: Convergence of the upper bound of a 95% confidence interval

We have set both the axis to logarithm scaling. This is because it gives a straight line plot for a plot of $\frac{1}{\sqrt{M}}$.

As we can see in the Monte Carlo plot the Confidence interval clearly follows the path of the $\frac{\text{Drift}}{\sqrt{M}}$ plot and hence we can confirm that the error decays are at the rate we expected. Conversely, for the Least-Squares Monte Carlo plot we can see that the error is decaying at a faster rate however we may expect this rate to slow down at higher numbers of simulations.

# Chapter 7

# Calibration of the Model

## 7.1 Interest Rates

Within our model we incorporate $r$, interest rate as the factor expressing the *risk-free* part of the market, independent of the stock (For more on interest rates see[18]). The risk-free rate is the minimum return that an investor would expect for any investment. They should not accept any additional risk unless the potential rate of return is greater than the risk-free rate. In order to decide on what metric we can use as a proxy for this risk-free rate we must consider the home market of the asset. In the case of Stocks for US companies we consider the US as the home market. Of course in real terms a true perfect risk-free rate does not exist because all investments carry at least some level of risk.

## 7.2 Volatility

The volatility $\sigma$ of the stock can be estimated on the basis of historical data or as the implied volatility.

### 7.2.1 Historical Volatility

The historical volatility is the realised volatility of an asset over a previous time period. We can use historical data to calculate continuously compounded Log-Returns and then take the standard deviation. As standard deviation is the statistical measure of the price change from the mean price change it can act as a good measure for the volatility of the asset.

However, there are drawbacks to using Historical Volatility in that the past performance of an asset is not necessarily a good indicator of future results and fails to take into account shifts as the market goes through different regimes.

### 7.2.2 Implied Volatility

Implied Volatility is the market's forecast of a likely movement in an asset's price. By using some market prices for call options we can inverse the Black-Scholes formula as seen in (3.1) to find a corresponding implied volatility.

The Black-Scholes formula can be written as:

$$c(r, T, K, S_0, \sigma) \tag{7.1}$$

and the function

$$\sigma \to c(r, T, K, S_0, \sigma)$$

is strictly increasing when $r, T, K, S_0$ are fixed. If we wanted to prove this we can compute the derivative with respect to $\sigma$ and show that it is positive. A strictly increasing function can always be inverted so we see the inverse function.

$$c \to \sigma(T, K, r, S_0, c)$$

We know that there is a one-to-one correspondence between the volatility and the option price, so if the Black– Scholes formula accurately reflects market

practice, different option prices should yield the same $\sigma$. However in real markets the resulting $\sigma$'s are not equal, contrary to the Black-Scholes model. For relatively small and large call prices the volatility's are larger than for the intermediate prices. The graph of the volatility against the option price is $U$ shaped and often called the *volatility smile*.

## 7.3 Correlation and Covariance

In section 3.3 we introduced the idea of using independent Brownian Motion to model our stock prices, this implied that stocks $S_1, \cdots, S_d$ are uncorrelated. In real life terms this is often not the case and it can be hard to gauge how different stocks from different industries are linked with one and other.

We introduce the concept of the covariance matrix which is a way of defining the relationship in the entire dimensions between random variables. For perfectly uncorrelated stocks $S_1, ..., S_d$ we can see the covariance matrix $\Sigma$ to be of the form:

$$\Sigma = \text{diag}(\sigma_1^2, \cdots, \sigma_d^2) = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_d^2 \end{pmatrix} \in \mathbb{R}^{d \times d}$$

Such that down the diagonal is the variance of each stocks log returns scaled to the time period. In reality we are likely to see a covariance $\Sigma = \varrho \in \mathbb{R}^{d \times d}$ where:

$$\varrho_{ij} \begin{cases} \sigma_i^2 & \text{if } i = j \\ \mathbb{R} & \text{if } i \neq j \end{cases}$$

In order to correlate our Brownian Motion we can use a Cholesky decomposition.

### 7.3.1 Correlating data with the Cholesky Decomposition

We use the Cholesky decomposition to decompose a positive definite matrix into the the product of a lower triangular matrix and its transpose

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{12} & L_{13} \\ 0 & L_{22} & L_{23} \\ 0 & 0 & L_{33} \end{pmatrix}$$

By using this we can generate correlated random variables by multiplying the lower triangular from decomposing the covariance matrix by standard normals. We can then perform a Cholesky decomposition of our covariance matrix of stocks and apply this to our Brownian Motion.

By importing some modules into Python we can make use of Yahoo finances real world data and can import some stock prices.

```
FANG=['FB','AMZN','NFLX','GOOG'] #Choosing our 4 stocks
Data={}
for stock in FANG:
    Data[stock]=pdr.data.DataReader(stock, data_source='yahoo
    ',start=dt.datetime.today()-relativedelta(years=1)) #
    Importing stock data for our underlyings from the past
    year

dic={}
for item in FANG:
    dic[item+' Returns']=Data[item]['Adj Close'].pct_change()
    #Calculates the returns for each stock

Returns=pd.DataFrame(dic) #Creating the data frame
Returns_Covariance=Returns.cov().to_numpy() #Creating an
    array for the covariance matrix
```

Listing 7.1: Obtaining Covariance data

We can then create another function as a modification of our function from listing (4.2) that incorporates NumPy's inbuilt Cholesky Decomposition thus allowing us to correlate our data:

```
def Correlated_Sample_Generator(cov_matrix,T,d,M):
    return np.linalg.cholesky(cov_matrix)@np.random.normal(0,
   np.sqrt(T),(d,M))
```

Listing 7.2: Multiplying the Cholesky decomposition of a covariance matrix by the random samples to generate correlated data

# Chapter 8

# The Basket Case

As an extension of our model we now look how to expand from the single stock case to the *basket call option* as discussed in section 2.2.

## 8.1 Monte Carlo Basket Pricing

The standard Monte Carlo method for pricing a basket call option works very similarliy to that of the single stock case. First we need to modify our Monte_Carlo_Sample_Generation function to allow for a higher dimension.

```
1 def Monte_Carlo_Sample_Generation_Higher(T,M,d):
2     return np.random.normal(0,np.sqrt(T),(M,d))
```

Listing 8.1: Generation of M samples of d variates from a uni-variate normal distribution for a mean of 0 and variance T

We go back to formula (4.1) to calculate our optimum fair price $\hat{\Pi}$ and modify our function get_payoffs to incorporate higher dimensions i.e more underlying stocks.

```
1 def get_payoffs_basket(r,S,K,T,sigma,W_T,M,d,w):
2     if np.sum(w)==1: #We require the weighted average to add
    up to 1
```

```
3         Smat=np.zeros((d,M)) #Starting our final S as a
    vector of zeros shape d x M
4         for i in range(d):
5                 Smat[i]=w[i]*S[i]*np.exp((sigma[i]*W_T[:,i])
    +(r-0.5*sigma[i]**2)*T)
6                 #For each underlying stock we apply our stock
     price formula
7         S_basket_prices=Smat.sum(axis=0) #Our basket price is
     the sum of the weighted stock prices
8         option_profit=S_basket_prices-K #Our options profit
    will be the stock price at maturity minus the strike price
9         payoffs=np.maximum(option_profit,0) #The option will
    only be exercised if in profit
10        return payoffs
11    else:
12        print('Weighted average must sum to 1') #In the case
    the weighted average does not add to 1 we print an error
    message
```

Listing 8.2: Simulating weighted stock prices from our given Brownian motion and then calculating the payoff for each simulation

This is a simple modification of our previous get_payoffs function as seen in Listing 4.3. We have adapted this for higher dimensions by including the weight vector $\omega$ in the calculation of the stock prices of each underlying stock and then calculating the basket price of each simulation by performing a sum and implementing equation (2.3).

We can run our new model along with our previously used Monte_Carlo_Basic function from Listing 4.4 to gain our fair price.

## 8.2 Least-Squares Monte Carlo Basket Pricing

We can take what we learnt in Chapter 5 and apply this to the basket case of more than one dimension. This time we must setup a new regression problem that allows for inputting $M$ sample points of dimension $d$ to represent the number of underlying assets in our basket. However, following the same structure as before our aim is to build a function that best approximates our realised value payoffs in the least squared sense and then calculate the expectation of the approximation function and multiply it by our discount factor.

### 8.2.1 Monomial Regression Problem

In the updated basket case we must first generate $M$ sample points $\{\mathbf{x}_i\}_{i=1}^{M}$ in which each vector $\mathbf{x}$ is made up $d$ points $\mathbf{x} = (x_1, x_2, ..., x_d)$ i.e dimension $d$. Where $d$ represents the number of underlying assets within the basket option.

### 8.2.2 Monomial Basis

In order to define our monomial basis for degree $P$ we let $\alpha = (\alpha_1, ..., \alpha_d)$ be a multi-index with components $\alpha \in \{0, 1, ..., P\}$ for every $i = 1, ..., d$. Then we can define our monomial functions as the following monomial example:

$$\phi_\alpha := \mathbf{x}^\alpha = x_1^{\alpha_1} * x_2^{\alpha_2} * ...x_d^{\alpha_d} \tag{8.1}$$

Now, for our monomial basis up to degree $P$ we will need to generate our multi-indices that allows for every possible combination of $d$ numbers that sums to $P$ i.e $\sum_{i=1}^{d} \alpha_i = P$. We will then take these sets of combinations and permutate them into all possible unique layouts.

To do this we can build a python function that generates the multi-indices

```
def generate_multindex(degree,dimensions):
    def finding_sums(input_numbers, target, partial=[], partial_sum=0): #This function is used to find all possible combinations of the variable numbers that equal the target
        if partial_sum == target:
            yield partial #If the partial_sum is equal to our target then we have succeeded and we yield the partial
        if partial_sum >= target:
            return #If not we return nothing
        for i, num in enumerate(input_numbers):
            remaining = input_numbers[i:] #Allowing all elements from i onwards to be used in the next recursion
            yield from finding_sums(remaining, target, partial + [num], partial_sum + num) #Using a recursion to go all the way down to all 1's and then back up and changing the partial
    sums=list(finding_sums(np.arange(1,degree+1),degree)) #Turn our sums into a list
    multi_index=np.zeros((len(sums),dimensions)) #We will create an array of zeros with as many columns as we have dimensions
    for i in range(len(sums)):
        if len(sums[i])<=dimensions: #We want all the sums that have elements less than or equal to the dimensions
            for j in range(len(sums[i])):
                multi_index[i,j]=sums[i][j] #Updating our multi_index array
    indexes=multi_index[~np.all(multi_index==0,axis=1)] #Removing all zero rows
    permutated_multindex=np.array([]) #Create an empty array
    for index in indexes:
        for permutation in multiset_permutations(index):
            permutated_multindex=np.append(permutated_multindex,np.array(permutation)) #We use a
```

```
    package from SymPy to find all the possible unique
    permutations of our indexes
21   return permutated_multindex.reshape((-1,dimensions)).
    astype('int') #Output our Multindices in one array
```

Listing 8.3: Generating a permutated multi-index for a certain degree and number of dimensions

This generate_multiindex function first makes use of another function finding_sums that recursively finds all the possible combinations of sums that equal a certain degree. Then we add all the sums that have less or equal elements than our dimensions to a multi-index array. Permutating this multi-index array to find all the possible unique orderings of the array.

By looping through this function we can generate multi-indices for a degree $P$ of 4 and a dimension $d$ of 3 to see

| $\alpha$ | $\phi_\alpha$ |
|---|---|
| (1, 1, 2) | $x_1 * x_2 * x_3^2$ |
| (1, 2, 1) | $x_1 * x_2^2 * x_3$ |
| (2, 1, 1) | $x_1^2 * x_2 * x_3$ |
| (0, 1, 3) | $x_2 * x_3^3$ |
| (0, 3, 1) | $x_2^3 * x_3$ |
| (1, 0, 3) | $x_1 * x_3^3$ |
| (1, 3, 0) | $x_1 * x_2^3$ |
| (3, 0, 1) | $x_1^3 * x_3$ |
| (3, 1, 0) | $x_1^3 * x_2$ |
| (0, 2, 2) | $x_2^2 * x_3^2$ |
| (2, 0, 2) | $x_1^2 * x_3^2$ |
| (2, 2, 0) | $x_1^2 * x_2^2$ |
| (0, 0, 4) | $x_3^4$ |
| (0, 4, 0) | $x_2^4$ |
| (4, 0, 0) | $x_1^4$ |

For each basis function we choose a numration and denote them by $\phi_j$ for $j = 1, ..., b$.

### 8.2.3 Monomial Basis Least-Squares Problem

As seen in section 5.1 we can now build our monomial basis matrix and fit it to our Least-Squares problem:

$$\min_{\mathbf{c}\in\mathbb{R}^{d+1}} \left\| \underbrace{\begin{bmatrix} 1 & \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_b(\mathbf{x}_1) \\ 1 & \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_b(\mathbf{x}_2) \\ \vdots & \vdots & & \vdots & \\ 1 & \phi_1(\mathbf{x}_M) & \phi_2(\mathbf{x}_M) & \cdots & \phi_b(\mathbf{x}_M) \end{bmatrix}}_{:=\mathbf{V}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_b \end{bmatrix}}_{:=\mathbf{c}} - \underbrace{\begin{bmatrix} f(\mathbf{x}_1) \\ f(\mathbf{x}_2) \\ \vdots \\ f(\mathbf{x}_M) \end{bmatrix}}_{:=\mathbf{f}} \right\| \tag{8.2}$$

With the goal of finding the optimum coefficient vector $\hat{c}$ that best approximates our realised values $f(\mathbf{x_i})$ such that it minimises $\min_{\mathbf{c}\in\mathbb{R}^{d+1}} ||\mathbf{Vc}-\mathbf{f}||$. To produce a function $p_{\hat{c}}(\mathbf{x})$ such that

$$f(\mathbf{x}) \approx p_{\hat{c}}(\mathbf{x}) := \hat{c}_0 + \sum_{j=1}^{n} \hat{c}_j \phi_j(\mathbf{x}) \tag{8.3}$$

which is an approximation of $f$ at all points $\mathbf{x} \in \mathbb{R}^d$. In our case we have our realised values of $f$ as our payoffs as seen in 2.2 and we can use our Brownian motion variable to generate $M$ samples of our payoff through our function get_payoffs_basket as seen in Listing 8.2.

Based on our function for creating a polynomial_basis in Listing 5.1 we build a function that creates an array form of our monomial basis matrix $\mathbf{V}$ as seen in (8.2).

```
1 def monomial_basis(inputs, degree=1, dimensions=1):
2     basis_matrix = np.ones((len(inputs), 1)) #Creating a
    basis_matrix with as many rows as inputs and a 1 in the
    first column
3     psi=generate_multindex(1,dimensions) #Generate all the
    possible indexes for degree 1
4     for i in range(2,degree+1):
5         psi=np.vstack((psi,generate_multindex(i,dimensions)))
```

```
    #Vertically stack onto our psi array the multindex for
   all degrees up to our intended degree
6   for index in psi:
7       basis_matrix=np.c_[basis_matrix,np.product(np.power(
   inputs,index),axis=1)] #For each manipulations set of
   brownian motion we apply the multindezs components in the
   form of a monomial
8   return basis_matrix, psi #Finally returning the
   basis_matrix and the important psi
```

Listing 8.4: Arranging samples into a Monomial Basis Matrix/Array of a chosen degree and dimension

We loop across the generate_multiindex function to create $b$ mult-iindices across every degree $1, ..., P$. Then the basis matrix can be formed by column stacking the associated monomial function $\phi_\alpha$ applied to all $M$ samples.

### 8.2.4 Approximating Expectations

As we saw in section 5.3.1, one way to calculate the expectation of our function $p_{\hat{c}}$ is by computing the integral of this function. However, as our input is formed from multivariate Brownian motion in the basket case we could integrate with the function multiplied by the multivariate normal probability distribution.

Another way in which we can calculate our expectation is through the use of Higher Moments.

### 8.2.5 Higher Moments for Expectation

Following on from our section 5.3.2 based on the linearity of the expectation operator we will be able to calculate the expectation of the function $\mathrm{E}[p_{\hat{c}}(\mathbf{x})]$ if we can calculate the expectations of each of our $\phi_j$ for $j = 1, ...b$. Such

that our fair price becomes:

$$\hat{\Pi} = e^{-rT}\mathrm{E}[\hat{c}_0 + p_{\hat{c}}(\mathbf{x})] = e^{-rT}\mathrm{E}[\sum_{j=1}^{b} \phi_j(\mathbf{x})\hat{c}_j] \tag{8.4}$$

As our $\phi$ functions are made up of the associated permutations of the different possible monomials of $\mathbf{x}$ knowing the higher moments becomes very useful.

**Definition 8.2.1.** The $k$th-order moments of $\mathbf{x}$ are given by:

$$= \mathrm{E}\left[\prod_{j=1}^{N} X_{j=1}^{r_j}\right] \quad \text{where} \quad \sum_{j=1}^{N} r_j = k$$

Where $r_1 + r_2 + ... + r_N = k$.

We can also make use of a *Isserlis' Theorem* to better calculate these moments.

**Theorem 8.2.2** (Isserlis' Theorem)**.** *If $(X_1, ..., X_d)$ is a zero-mean multivariate normal random vector then under the even case $d = 2m$*

$$\mathrm{E}[X_1 X_2 \cdots X_{2m}] = \sum\prod \mathrm{E}[X_i X_j] = \sum\prod \mathrm{Cov}(X_i X_j)$$

*and the odd case $d = 2m + 1$*

$$\mathrm{E}[X_1 X_2 \cdots X_{2m+1}] = 0$$

where the notation $\sum\prod$ means summing over all distinct ways of partitioning $(X_1, ..., X_{2m})$ into pairs. That is, the $2m$ variables are partioned into cells containing pairs of the variables. The expected values $\mathrm{E}[X_i X_j]$ are formed for the products of the pairs in each cell and these expected values are multiplied together for all $m$ cells. These products are then summed for all partitions into cells which are distinct, that is, the ordering of the cells

and of the pairs within the cells are not distinguished (For a more detailed breakdown on Isserlis' theorem see [19]).

As we have zero-mean variables we have $\mathrm{Cov}(X_i X_j) = \mathrm{E}[X_i X_j]$ and we can see an example in the 4th-order moment case as:

$$\mathrm{E}[X_1 X_2 X_3 X_4] = \sigma_{12}\sigma_{34} + \sigma_{13}\sigma_{24} + \sigma_{14}\sigma_{23}$$

where $\sigma_{ij}$ is the covariance of $X_i$ and $X_j$.

### 8.2.6 Generating Higher Moments in Practice

In order to find the different pair partitions of $(X_1, ..., X_{2m})$, the even case, we define two functions to help:

```
def pairing(inputs):
    return zip(*([iter(inputs)] * 2)) #By using a zip we can group the data into chronological pairs

def partitioning(input_numbers, n=2):
    set_partitions = set() #Define an empty set
    for permutation in it.permutations(input_numbers): #We use the permutation function from the itertools package to loop over all possible permutations of an iterable sequence
        grouped = pairing(list(permutation)) #We group the permutations into pairs
        sorted_group = tuple(sorted([tuple(sorted(partition)) for partition in grouped])) #Sort the items in the individual pair and then sort all the pairs
        set_partitions.add(sorted_group) #Add to our set, by using a set we get only the unique items
    return set_partitions
```

Listing 8.5: Partitioning all possible X variations inline with Isserlis' theorem

The pairing function zips together the range of a list into chronological pairs. With the partitioning function then finding all possible unique permutations of our range $1, ..., k$ inline with Isserlis' theorem. So that if you want to find the partitions such that we have up to the 6th order as:

| $k$th-order | Partitions |
|---|---|
| 2 | (0,1) |
| 4 | (0, 1), (2, 3)<br>(0, 2), (1, 3)<br>(0, 3), (1, 2) |
| 6 | (0, 1), (2, 3), (4, 5)<br>(0, 1), (2, 4), (3, 5)<br>(0, 1), (2, 5), (3, 4)<br>(0, 2), (1, 3), (4, 5)<br>(0, 2), (1, 4), (3, 5)<br>(0, 2), (1, 5), (3, 4)<br>(0, 3), (1, 2), (4, 5)<br>(0, 3), (1, 4), (2, 5)<br>(0, 3), (1, 5), (2, 4)<br>(0, 4), (1, 2), (3, 5)<br>(0, 4), (1, 3), (2, 5)<br>(0, 4), (1, 5), (2, 3)<br>(0, 5), (1, 2), (3, 4)<br>(0, 5), (1, 3), (2, 4)<br>(0, 5), (1, 4), (2, 3) |

Where the number of terms within the partitions in the even case $d = 2m$ is $(2m-1)!!$. Putting this together we can generate our higher moments using two functions below:

```
def ordering_of_Xs(alpha):
    ordering=np.array([]) #Create an empty array
    for i,alph in enumerate(alpha):
```

```
4           ordering=np.append(ordering,i*np.ones(alph)) #Add to
    the array a sequence of indexes for however many is stated
     in the multiindex intended to be used as the slice for
    the covariance matrix
5       return ordering.astype('int')
6
7   def generate_higher_moments(covariance_matrix,psi):
8       E=np.zeros(len(psi)) #We will have the same number of
    moments as we have number of psi functions
9       for i,alpha in enumerate(psi): #Loop across all psi
    functions
10          k=sum(alpha) #The sum of the multiindex is the order
    of the moment
11          if k%2!=0: #Check to see if it is odd
12              E[i]=0 #If the order is odd then the moment is 0
13          if k==2: #If the order is two we can do a simple
    calculation
14              ordering=ordering_of_Xs(alpha) #Find the correct
    ordering of alpha
15              E[i]=covariance_matrix[ordering[0],ordering[1]] #
    The second order moment is just the covariance of the two
    variables
16          if k%2==0 and k!=2: #If the order of the moment is
    even and not equal to two
17              partitions=partitioning(range(k)) #We find all
    the partition pairs
18              ordering=ordering_of_Xs(alpha) #Find the ordering
     of the Xs
19              sum_run=0 #Initiate a sum at 0
20              for partition in partitions:
21                  prod_run=1 #Initiate a product at 1
22                  for pair in partition:
23                      prod_run*=covariance_matrix[ordering[pair
    [0]],ordering[pair[1]]] #We multiply all pairs covariances
     together
24                  sum_run+=prod_run #We sum for every
    covariance product for every partition
```

```
25                E[i]=sum_run #The moment becomes this sum
26        return E
```

Listing 8.6: Generating Higher Moments for our chosen basis functions based on a covariance matrix of the data

Our ordering_of_Xs function will take the multindex and create a new array where each element represents one of the random variables $X_i$ for all $i$. The generate_higher_moments creates an array of expectations with the same amount of elements as the number of basis functions $b$. As the generate_higher_moments loops through the basis functions if the order $k$ is odd then the corresponding moment will be 0. If the moment is even then the function will implement Isserlis' theorem so that the corresponding expectation of that basis function is formed by multiplying together and then summing the corresponding covariances from each partition of that order $k$ of said basis function.

### 8.2.7 3D Plot of a 2 Dimensional Basket Case

We setup the parameters such that:

| Parameters | Values |
|---|---|
| Interest rate, $r$ | 0.1 |
| Initial Stock Prices, $S$ | [50,60] |
| Strike Price, $K$ | 60 |
| Time (years), $T$ | 1 |
| Volatility $\sigma$ | [0.15,0.3] |
| Simulations, $M$ | 1000 |
| Weights, $\omega$ | [0.5,0.5] |
| Monomial Degree, $P$ | 5 |

and look at a 3D plot of our function $p_{\hat{c}}$ in comparison to our realised value $\mathbf{f}$ payoffs

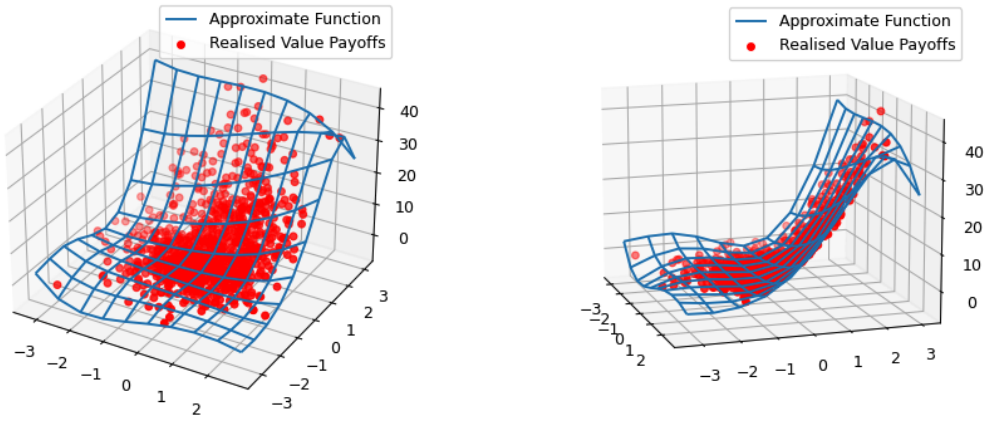



Figure 8.1: Approximation function of degree 5 in comparison to realised value payoffs

As we saw in the one dimensional case we observe a king in the observed values as our basket price reaches the strike price $K$. The model does a good job at fitting the function to our realised values but could be improved through a variation in monomial basis/degree and an increased number of samples.

### 8.2.8 Least-Squares Monte Carlo Basket Pricing in Practice

Putting together all of the functions we have made together we can build up our function in Listing 5.4 to calculate the fair price of a basket call option of $d$ underlying assets up to a chosen degree $P$.

```
1 def Monte_Carlo_Least_Squares_Monomial_Higher_Moments(r,T,W_T
    ,P,payoffs,d):
2     X, psi=monomial_basis(W_T,P,d) #Putting the brownian
    motion into our basis matrix and also outputting the psi
    functions
3     c=sp.linalg.lstsq(X,payoffs)[0] #Finding optimum c that
    minimises our regression problem
```

```
4     E=generate_higher_moments(np.cov(W_T.T),psi) #Generate
    our array of higher moments for every psi function
5     fair_price=(c[0]+np.sum(c[1:]*E))*np.exp(-r*T) #Our
    intercept plus the summation of our weights multiplied by
    our expectations
6     return fair_price #Returning our expectation multplied by
     our discount factor which is our fair price
```

Listing 8.7: Calculating the fair price of a basket option using Least-Squares Monte Carlo and higher moments

The function first generates our basis matrix and basis functions and solves for the optimum $\hat{\mathbf{c}}$. Then by generating the higher moments it can calculate the fair price by implementing our equation (8.4).

## 8.3 Comparison of Basket Case Models

If we setup a 4 dimensional case as such that:

| Parameters | Values for Stocks A,B,C,D |
|---|---|
| Interest rate, $r$ | 0.1 |
| Initial Stock Prices, $S$ | [50,60,70,80] |
| Strike Price, $K$ | 65 |
| Time (years), $T$ | 1 |
| Volatility $\sigma$ | [0.15,0.3,0.6,0.2] |
| Weights, $\omega$ | [0.4,0.25,0.25,0.1] |
| Monomial Degree, $P$ | 7 |

Using this data we can then generate fair prices for both our models equally spaced number of simulations ranging from 1000 to 10000 with

```
1 M_range=np.linspace(1000,100000,100).astype('int')#Arranging
    a set of simulations
2 Monte_Carlo=np.zeros(len(M_range))
3 Monte_Carlo_Least_Squares=np.zeros(len(M_range))
4 for i,m in enumerate(M_range):
```

```
5     W_T=Monte_Carlo_Sample_Generation_Higher(T,m,d) #Our
    Random Variable
6     payoffs=get_payoffs_basket(r,S,K,T,sigma,W_T,m,d,w) #
    Defining our Payoffs
7     Monte_Carlo[i]=Monte_Carlo_Basic(r,T,payoffs) #Adding our
     standard Monte Carlo fair price for every simulation
8     Monte_Carlo_Least_Squares[i]=
    Monte_Carlo_Least_Squares_Monomial_Higher_Moments(r,T,W_T,
    P,payoffs,d) #Adding our Least Squares Monte Carlo fair
    price for every simulation
```

Listing 8.8: Generating fair prices for our models across a range of simulations

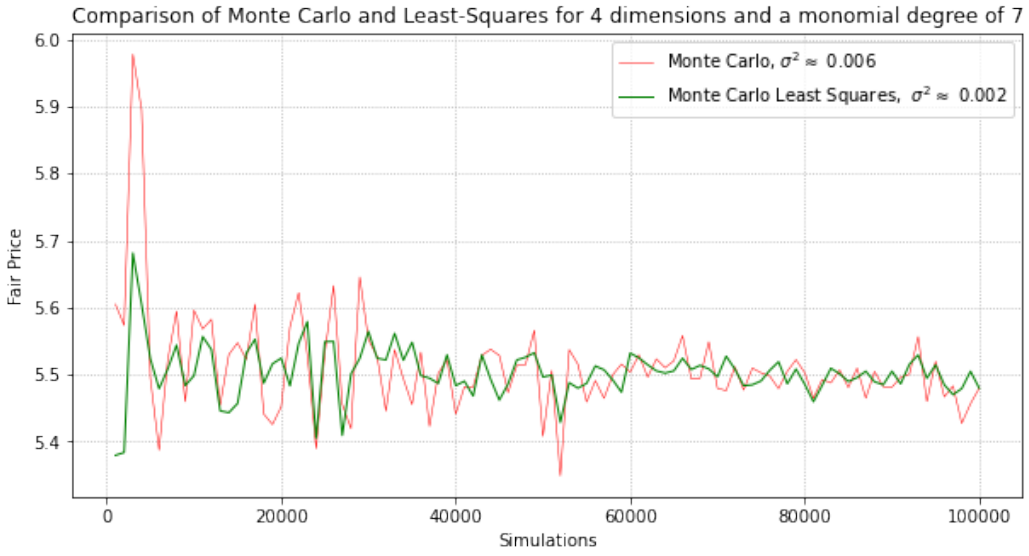



Figure 8.2: Payoffs with a least-squares regression of degree 4

By plotting this data we can see across this simulation domain that the Least-Squares Monte Carlo has a tighter variance of $\sigma^2 = 0.002$ compared to standard Monte Carlo at 0.006.

# Chapter 9

# Conclusion

In this paper we have gone through the fundamentals of using Monte Carlo and Least-Squares methods to price standard and basket call options. By importing the concept of least-squares into our traditional Monte Carlo method we can see how we can gain a greater level of accuracy under the same number of simulations. This is done whilst exploring how to setup of these methods and a comparison of these methods across different examples.

We can now see how in the case of basket options, in which it can be difficult to find a fair price we can use these methods to 'brute force' an answer to the price for a derivative. Of course these models can be calibrated in order to get better results but by outlining the fundamentals of each method and building upon each chapter we find certain areas of the models that can be explored further.

Both Monte Carlo and Least-Squares Monte Carlo have weaknesses. Our standard Monte Carlo method can become computationally expensive because it requires a large number of samples to reach the desired level of accuracy. The Least-Squares Monte Carlo faces issues of bottle necks when working with a high level of degree and dimensions due to the ever increasing number of basis functions required and the matrix manipulations involved.

Some suggestions for future work would be to look at different ways to define our basis functions and how choosing only a subset of all possible monomial functions will effect the overall accuracy. In a real life setting there is clearly a trade off needed between time, power and accuracy and so I think it would be interesting to explore where we can tweak these methods and parameters for the desired balance.

# Bibliography


[1] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature. 2020 Sep;585(7825):357-62. Available from: https://doi.org/10.1038/s41586-020-2649-2.

[2] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods. 2020;17:261-72.

[3] McKinney W, et al. Data structures for statistical computing in python. In: Proceedings of the 9th Python in Science Conference. vol. 445. Austin, TX; 2010. p. 51-6.

[4] Hunter JD. Matplotlib: A 2D graphics environment. Computing in Science & Engineering. 2007;9(3):90-5.

[5] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, et al. SymPy: symbolic computing in Python. PeerJ Computer Science. 2017 Jan;3:e103. Available from: https://doi.org/10.7717/peerj-cs.103.

[6] Van Rossum G. The Python Library Reference, release 3.8.2. Python Software Foundation; 2020.

[7] Hull JC. Options, futures and other derivatives. Pearson; 2019.

[8] Caldana R, Fusai G, Gnoatto A, Grasselli M. General closed-form basket option pricing bounds. Quantitative Finance. 2016;16(4):535-54.

[9] Crack TF. Basic Black-Scholes: Option Pricing and Trading. TF Crack; 2009.

[10] Vanguard SP 500 ETF;. Available from: https://markets.ft.com/data/etfs/tearsheet/risk?s=VOO:PCQ:USD.

[11] U.S. Department of the Treasury; 2021. Available from: https://www.treasury.gov/resource-center/data-chart-center/interest-rates/pages/textview.aspx?data=yield.

[12] Stoer J, Bulirsch R. Introduction to numerical analysis. vol. 12. Springer Science & Business Media; 2013.

[13] Suhov Y, Kelbert M. Probability and statistics by example: volume 2, Markov chains: a primer in random processes and their applications. vol. 2. Cambridge University Press; 2008.

[14] Larson HJ. Introduction to probability theory and statistical inference; 1974.

[15] Ramachandran KM, Tsokos CP. Mathematical Statistics with Applications. Elsevier Science; 2009.

[16] Papoulis A, Pillai SU. Probability, random variables, and stochastic processes. Tata McGraw-Hill Education; 2002.

[17] Jäckel P. Monte Carlo methods in finance. vol. 71. J. Wiley; 2002.

[18] Capiński M, Kopp E. The Black–Scholes Model. Cambridge University Press; 2012.

[19] Michalowicz J, Nichols J, Bucholtz F, Olson C. A general Isserlis theorem for mixed-Gaussian random variables. Statistics & probability letters. 2011;81(8):1233-40.