Computational Methods in Finance

Elisha Gretton

MAS462 Semester 1

1 Obtaining and cleaning financial data

S&P 500 is a summary fund that tracks the current top 500 publicly-traded US companies. It captures the performance of the US stock market due to the diversity of companies, and overall has a good growth rate over time. In the last year, S&P 500 has grown by 29.19% [1].

Using *Yahoo! Finance*, we can upload five year's worth of historical data of S&P 500 and investigate the data. The programming language used is Python.

Taking the first two entries of the S&P 500 data, the historical data has the following format,

Date	Open	High	Low	Close	Adj Close**	Volume
Nov 01, 2021	4,610.62	4,620.34	4,595.06	4,613.67	4,613.67	2,924,000,000
Oct 29, 2021	4,572.87	4,608.08	4,567.59	4,605.38	4,605.38	3,632,260,000

where

- **Date:** day of trade,
- **Open:** starting price of the day,
- **High:** most expensive price,
- **Low:** cheapest price,
- **Close:** ending price of the day,
- **Adj Close**:** close price adjusted for dividends and splits,
- **Volume:** number of shares traded.

By using the package `yfinance`, the last five years of this data can be downloaded using the following code,

```
import yfinance as yf
dataset = yf.download('^GSPC', start='2016-11-16', end='2021-11-16').
```

`^GSPC` is the an abbreviation used in the UK stock market for the S&P 500. This is known as a ticker [2] and can be changed depending on the chosen dataset. You can also specify the start and end date of the chosen data by altering the code above.

Now this data has been uploaded, it is important to check for missing values to avoid misrepresentation in the data. To do this, *interpolation* is used. This numerical technique works by looking for trends in previous or following data and estimates the missing value from these points [4]. Some examples include linear, polynomial and padding interpolation.

1. **Linear Interpolation**
   This works by looking at the previous values in the data and looking for a linear relationship. After this, the missing value is estimated from this relationship [5]. This can be implemented in Python using the following command `dataset.interpolate()` .

2. **Polynomial Interpolation**
   Polynomial interpolation starts by finding a low-degree polynomial relationship between previous data and then estimates the missing value from this relationship. A polynomial of degree 2 can be implemented in Python using the command,
   `dataset.interpolation(method="polynomial", order=2)`.

3. **Interpolation through padding**
   Interpolation through padding means to fill in missing values with the value of its previous datapoint. A limit must be specified, which means how many missing values to fill. A padding of limit 2 can be implemented in Python using the command,
   `dataset.interpolate(method='pad', limit=2)`.

In the examples above, we have been looking at forward interpolation. This is when you look at previous values to estimate data that follows. You can also interpolate backwards by looking at following data to estimate previous data.

Next, we will begin to look at Date and Adjusted Prices. Adjusted Prices are important as they take adjustments into account when calculating the stock's value such as dividends and splits [6]. Dividends is an amount of money paid regularly to shareholders from profits, and splits are when the number of shares are increased by issuing more shares to current shareholders [10]. These are important to factor into the Adjusted Price as they both affect the true value of the asset. As a result, this portrays a more accurate reflection of the stock price. The following command creates a new dataset using the Adjusted Price column, `new_dataset = dataset[['Adj Close']].copy()`. We will then convert this dataset into an numpy array using the command
`new_dataset.to_numpy()`, to use in Part 5.

2 Binomial tree method for derivative pricing

Binomial trees can be used to calculate the price of derivatives. The concept of payoff is important for binomial trees. Two important facts are:

- European options give the owner the right to buy/sell its underlying asset at a certain price on a certain date. Let X be the strike price, with expiration at time $t = T$, and S_T the spot price of the underlying asset at the expiration of the option. The payoff for a European call option is $\max\{X - S_T, 0\}$ and for a European put option is $\max\{S_T - X, 0\}$.
- American options give the owner the right to buy/sell its underlying asset before expiration.

When forming a binomial tree, the underlying asset price evolves in an upward, u, or downward motion, d. This is why we use a tree-like structure to visualise the upward and downward evolution of price. It allows us to easily understand how each node evolves by following each tree branch, which will result in a combination of u and d. Working backwards, the price of the derivative can be calculated by using the concept of risk neutral valuation. Risk-neutral valuation assumes that assets grow and can be discounted at a risk-free rate. Using this theory, we can assume the price of a derivative is calculated as the expected present value of its payoff. In mathematical form, this definition translates to the equation,

$$E = qS_u + (1 - q)S_d = e^{rt}S, \tag{1}$$

where S_u is the price of the upward direction, S_d is the price of the downward direction, q and $1 - q$ are the corresponding probabilities of each price, r is the annual interest rate, t is the time to expiration, and S is the current price of the asset.

2.1 Implementation of the American Put Option

The American put option can be implemented in Python using the concepts of risk neutral valuation and payoff. The difference with an European put option is that it can be exercised at any time. Therefore, it is important to consider immediate exercise at each node and factor this into the binomial tree.

Input: S_0 (initial price of asset), X (strike price), T (time to expiration), r (annual interest rate), u (upwards movement), d (downwards movement), n (number of time steps).

Output: P (vector containing price of derivative at each node)

Algorithm 1: Pseudocode for an American Put Option

Input: S_0, X, T, r, u, d, n
$dt \leftarrow T/n$;
$q \leftarrow \frac{e^{rt}-d}{u-d}$;
for i *in* S **do**
  $S[i] = S_0 u^i j^{n-i}$ Calculate price of nodes
end
for i *in* $n+1$ **do**
  $C[i] = max(0, X - S[i])$ Calculate payoff of last column;
  Add $C[i]$ to P.
end
for *column from last* **do**
  for *row in column* **do**
    Calculate price S;
    Calculate price of derivative using equation (1);
    $C[row] = max(C[row], X - S)$ Calculate payoff of American put;
    Add $C[row]$ to P.
  end
end
Output: P

In algorithm (1), the pseudocode runs through the basis of the code. When calculating the payoff for an American put, it is calculated as
`C[row]=max(C[row],X-S).`
This is taking into account immediate exercise, which would be $X - S$. See the attached Python file to see this implemented.

3 Monte Carlo simulation for derivative pricing

3.1 Brownian Motion

Brownian motion is a random process that can be used in modelling stock prices. It is important to introduce randomness in financial models as naturally, randomness exists, but also as financial models fluctuate a lot. We should imitate natural fluctuations in data by introducing randomness with mathematical techniques, such as Brownian motion. There are four important properties to Brownian motion:

1. $B_0 = 0$,
2. for $0 \leq s < t$ the increment $B_t - B_s$ is normally distributed with mean 0 and variance $t - s$,
3. for any $0 \leq t_1 < t_2 < ... < t_n$ the increments $B_{t_1} - B_0, B_{t_2} - B_{t_1}, ..., B_{t_n} - B_{t_{n-1}}$ are independent random variables,
4. for any $\omega \in \Omega$ the function $t \mapsto B_t(\omega)$ is continuous.

In order to implement Brownian motion in Python, these properties are used.

3.2 Implementation of Brownian Motion in Python

A function called `standard_brownian_motion` is created with input `n_steps`. The input `n_steps` represents the number of time steps for the motion. Below is the implementation of this process with comments to explain the code.

```
# Import relevant modules
import numpy as np
import matplotlib.pyplot as plt

def standard_brownian_motion(n_steps):
    # Set overall time period
    T = 1.

    # Create equally-spaced points over time period, T
    times = np.linspace(0,T,n_steps)

    # By property (2), variance = t-s.
    dt = times[1] - times[0]

    # By property (2), calculate increments
    dB=np.sqrt(dt)*np.random.normal(size=(n_steps-1,))

    # By property (1), initial value B0=0
    B0=np.zeros(shape=(1,))

    # Add initial value to cumulative sum of increments
    B=np.concatenate((B0,np.cumsum(dB)))

    # Plot time vs B for Brownian motion
    plt.plot(times,B)
```

Below (Figure 1) are four cases of this standard Brownian motion being implemented.

3.3 Geometric Brownian Motion

Geometric Brownian motion (GBM) is another stochastic process that is based off standard Brownian motion but includes a drift term, μ, and a volatility term (measure of spread), σ. GBM is a popular technique as it is known to imitate financial data well. For example, the paths given by GBM are said to show the same 'roughness' compared to real stock prices, only positive values are considered, and the independence between expected returns and stock prices exists [8]. All of these properties are realistic for financial data. The downside to GBM is that volatility is assumed constant. In reality, volatility may change

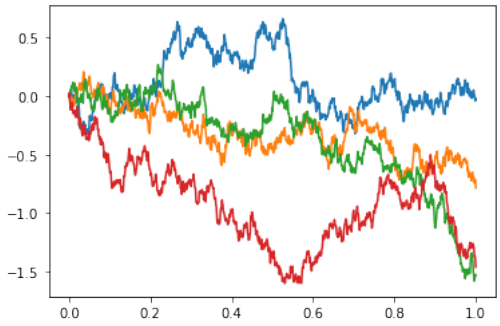

Figure 1: Four instances of Brownian motion with $n\_steps = 1000$.

due to unpredictable events. Despite this, GBM is still a good method to price options and can be given in the Euler discretization of the equation:

$$S(t + \Delta t) = S(t) + \mu S(t)\Delta t + \sigma S(t)\Delta B(t), \tag{2}$$

where S is the asset price, $B(t)$ is the standard brownian motion, ΔB is the change of brownian motion over a time period Δt. Below is the implementation of geometric brownian motion with comments to explain the code.

```
def geometric_brownian_motion(mu,sigma,S0,n_steps):
  # Create time steps, dt
  dt = 1.0 / n_steps

  # Create array of all times
  t = np.arange(dt, 1 + dt, dt)

  # Using the same concepts of Brownian motion, calculate
  dB and B.
  dB = np.sqrt(dt) * np.random.randn(n_steps)
  B  = np.cumsum(dB)

  # Using the Euler's discretization, calculate S at each
  time step
  S_simulation, S=[],S0
  for j in range(n_steps):
    S+= mu*S*dt+sigma*S*dB[j] # Euler discretization
    S_simulation.append(S) # add S to an array

  # Plot time vs. all S data to generate GBM
  label1=``mu="+str(mu),``sigma="+str(sigma)
  plt.plot(t,S_simulation,label=label1)
  plt.legend()
```

In Figure 2, there are four instances of geometric brownian motion being implemented.

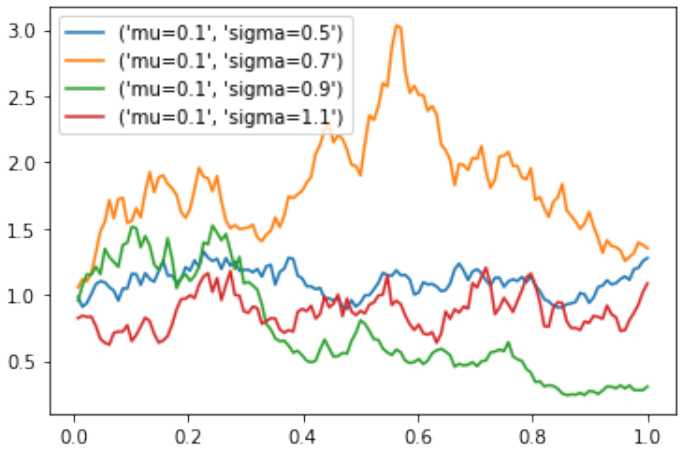

Figure 2: Four instances of Geometric Brownian motion with $n\_steps = 2^7$.

3.4 Monte Carlo estimation of European option prices

Monte Carlo is another method that is used to price derivatives. This method is useful if there are multiple sources of uncertainty. Using regular approaches to price derivatives such as Black-Scholes equation would be difficult, therefore the Monte Carlo method takes advantage of random sampling to beat uncertainty. In more detail, it works by randomly sampling from a set of possible outcomes and then taking a fraction of the random draws to estimate the value of a derivative. Below is the pseudocode outlining the Monte Carlo method for pricing an option.

Algorithm 2: Pseudocode for Monte Carlo Estimation of European Option

Input: $S_0, K, T, r, \sigma, n\_steps$
Create empty list C to store price of asset at each time step;
Set $C\_total = 0$ to store $C_1 + C_2 + ... + C_N$ later on;
for i *in* n_steps **do**
  Generate random variable with normal distribution Z_i;
  Calculate asset price by $S_i(T) = S(0)exp([r - \frac{1}{2}s^2]T + \sigma\sqrt{T}Z_i)$;
  Calculate payoff depending on option type;
  Discount payoff to the present;
  Add discounted payoff to C;
  Add C to C_total so $C\_total = C_1 + C_2 + ... + C_N$;
end
Calculated estimated price using;
$C\_hat = C\_total/n\_steps = C_1 + C_2 + ... + C_N/n\_steps$,
Output: C, C_hat

The implementation of this pseudocode can be seen in the Python file.

The theory behind this method is built off the Law of Large Numbers and Central Limit Theorem. Assume that the random numbers $C_1, C_2, ..., C_N$ are generated from distribution Z. The Monte Carlo estimation of the option price will be

$$\hat{C} = \frac{1}{N}\sum_{i=1}^{N} f(C_i).$$

By the law of large numbers, the approximation $\hat{C}$ converges to the true value as N increases. By the central limit theorem, we can estimate $\sigma^2_{N-1} = Var[f(Z)]$ and also create 95% confidence intervals for $\hat{C}$. The formula for the 95% confidence interval of $\hat{C}$ is

$$(\hat{C_N} - 1.96\frac{Var[f(Z)]}{\sqrt{N}}, \hat{C_N} - 1.96\frac{Var[f(Z)]}{\sqrt{N}}).$$

To apply this theory, an example is used. By using parameters $S_0 = 100, K = 80, T = 1, r = 0.05, \sigma = 0.3, n\_steps = 1000$, the estimated price for an European call option is 26.62 with a 95% confidence interval $(26.25, 26.99)$. The estimated price for a European put option is 2.57 with a 95% confidence interval $(2.48, 2.65)$.

Next we will see how changing the parameters affects the call and put option values. Below is a table with the same parameters $S_0 = 100, K = 80, T = 1, r = 0.05, n\_steps = 1000$ but varying σ.

Volatility	Call	Put
0	23.902	0
0.1	48.923	23.565
0.2	57.23	48.891
0.3	67.441	64.978
0.4	88.266	72.213

It looks like as volatility increases, the value of the call and put option increases. The reason for this is due to the concept of downside and upside risk. When there is downside risk, the strike price X of the option is lower than the spot price S_T, therefore the call option is not exercised, but the put option is. Upside risk is the opposite so when the strike price is higher than the spot price. The call option is exercised and the put option isn't. Higher volatility means higher downside or upside risk, which means that call and put options become more valuable as volatility increases [11]. Moreover, increasing interest rate is also tested. As interest rates increases, the value of the call option increases and the value of the put option decreases. The reason for the call option increases is the payoff increases. The payoff of a call option is $\max\{S_T - X, 0\}$ If interest rates are larger, S_T is larger, therefore the payoff for a call option increases. The payoff for a put option is $\max\{X - S_T, 0\}$. If interest rates are larger, S_T is larger, therefore $X - S_T$ is smaller, so the payoff function decreases. This decreases the value of the put function.

The question that rises from using Monte Carlo estimation is its data-independence when pricing options. When simulations are ran repeatedly, Monte Carlo can generate many possibilities using the probability distribution. This would lead us to think the estimation is data-independent. However, the technique is generating results given no knowledge of the market e.g. inflation rates. It is important to take into account these other factors that aren't factored into mathematical distributions.

4 Finite Differences for derivative pricing

4.1 Introduction to Black-Scholes and Comparing Methods

The Black-Scholes equation is another way of pricing options. It is an equation that takes in consideration strike price, spot price, time to expiration, the risk-free rate of return, and volatility of an asset. It allows investors to protect all option risk through buying and selling options over time [12]. A downside to Black-Scholes is that there are a lot of assumptions that decrease the flexibility of this method. One example is volatility being constant, when volatility may change over time [9].

Finite difference methods, such as the Explicit Euler, Implicit Euler, and Explicit Crank-Nicolson method, can be used to solve the Black-Scholes equation. To compare each of the methods, the computational efficiency is first

evaluated. The lack of matrix systems makes the explicit Euler method more computationally efficient, therefore it takes less time for a solution to be computed. The other two methods use matrix systems, therefore they have a much longer computational time. Despite this, the Explicit Euler scheme requires Δt and Δx for stability, whereas the implicit Euler and Crank-Nicolson method do not. This is a disadvantage as decreasing Δx can make the Explicit Euler method more unstable, whereas it has no effect on the stability of the other two methods. The other two methods are therefore more stable than explicit Euler. Additionally, the Crank-Nicolson method has the smallest truncation error so it is more accurate. To decide which method is best, it is important to decide between computational time, stability and truncation error. If computational time isn't an issue, then choose Crank-Nicolson as it has the lowest error, and is stable. If error isn't an issue, then choose explicit Euler as it computes solutions a lot faster.

If we want to compare Black-Scholes to binomial trees, the binomial tree method is more flexible as each step can be changed if there are differences caused by irregular option features, such as option type. There is less flexibility in the standard Black-Scholes method. Binomial trees are difficult to construct and can be rather large, decreasing its space efficiency. Due to the size of the tree, it can also increase the computational time, making this process less efficient. The Black-Scholes method can be calculated using less computational time and space, as the method requires less memory compared to trees, and is a much simplier process to implement [13].

Comparing Black-Scholes to the Monte Carlo method, the Monte Carlo method is a lot more flexible as you can add in other inputs such as transaction costs and dividends. On the other hand, the Monte Carlo method is computationally more expensive as there can be thousands of calculations depending on the amount of uncertainties and probability distribution.

4.2 Transforming Black-Scholes to the heat equation

Next, we will discuss the transformation of Black-Scholes to the heat equation. The Black-Scholes equation is as follows:

Proof.

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{3}$$

Start with the substitution $S = e^y$ and $t = T - \tau$. By the chain rule, we obtain

$$\frac{\partial V}{\partial S} = \frac{\partial V}{\partial y}\frac{\partial y}{\partial S} = \frac{\partial V}{\partial y}\frac{1}{S}, \tag{4}$$

as $S = e^y$ means $y = ln(S)$ and $\frac{\partial y}{\partial S} = \frac{1}{S}$. This also gives

$$\frac{\partial^2 V}{\partial S^2} = \frac{\partial V}{\partial y}\frac{-1}{S^2} + \frac{1}{S^2}\frac{\partial^2 V}{\partial x^2}. \tag{5}$$

We can also use the chain rule to calculate $\frac{\partial V}{\partial t}$.

$$\frac{\partial V}{\partial t} = \frac{\partial V}{\partial \tau}\frac{\partial \tau}{\partial t} = \frac{\partial V}{\partial \tau}(-1), \tag{6}$$

as $t = T - \tau$ so $\frac{\partial t}{\partial t} = -\frac{\partial \tau}{\partial t}$, which gives $\frac{\partial \tau}{\partial t} = -1$. By substituting equation (4),(5) and (6) into equation (3), we obtain the equation:

$$-\frac{\partial V}{\partial \tau} + \frac{1}{2}\sigma^2 S^2(\frac{\partial V}{\partial y}\frac{-1}{S^2} + \frac{1}{S^2}\frac{\partial^2 V}{\partial x^2}) + r\frac{\partial V}{\partial y} - rV = 0.$$

By rearranging, this gives

$$\frac{\partial V}{\partial \tau} - \frac{1}{2}\sigma^2\frac{\partial^2 V}{\partial x^2} - (r - \frac{1}{2}\sigma^2)\frac{\partial V}{\partial y} + rV = 0. \tag{7}$$

Now let $u = e^{r\tau}V$. This gives $V = ue^{-r\tau}$. By the chain rule,

$$\frac{\partial V}{\partial \tau} = -rue^{-r\tau} + \frac{\partial u}{\partial t}e^{-r\tau},$$

$$\frac{\partial V}{\partial y} = e^{-r\tau}\frac{\partial u}{\partial y},$$

$$\frac{\partial^2 V}{\partial y^2} = e^{-r\tau}\frac{\partial^2 u}{\partial y^2}.$$

Substituting the equations above into equation (7) gives

$$-rue^{-r\tau} + \frac{\partial u}{\partial \tau}e^{-r\tau} - \frac{1}{2}\sigma^2[e^{-r\tau}\frac{\partial^2 u}{\partial y}] - (r - \frac{1}{2}\sigma^2)e^{-r\tau}\frac{\partial u}{\partial y} + rue^{-r\tau} = 0$$

$$= e^{-r\tau}[-ru + \frac{\partial u}{\partial \tau} - \frac{1}{2}\sigma^2\frac{\partial^2 u}{\partial y^2} - (r - \frac{1}{2}\sigma^2)\frac{\partial u}{\partial y} + ru].$$

By rearranging, this gives the final equation

$$\frac{\partial u}{\partial \tau} - \frac{1}{2}\sigma^2\frac{\partial^2 u}{\partial y^2} - (r - \frac{1}{2}\sigma^2)\frac{\partial u}{\partial y}. \tag{8}$$

Finally, let $x = y + (r - \frac{1}{2}\sigma^2)\tau$ such that $\frac{\partial x}{\partial \tau} = r - \frac{\sigma^2}{2}$. By the chain rule,

$$\frac{\partial u}{\partial \tau} = \frac{\partial x}{\partial \tau}\frac{\partial u}{\partial x} + \frac{\partial \tau}{\partial \tau}\frac{\partial u}{\partial \tau} = (r - \frac{\sigma^2}{2})\frac{\partial u}{\partial x} + \frac{\partial u}{\partial \tau} = (r - \frac{\sigma^2}{2})\frac{\partial u}{\partial y} + \frac{\partial u}{\partial \tau}.$$

We can also calculate $\frac{\partial^2 u}{\partial y^2}$ using the chain rule.

$$\frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial x^2}(\frac{\partial u}{\partial x})^2 + (\frac{\partial u}{\partial x})\frac{\partial^2 x}{\partial y^2} = \frac{\partial^2 u}{\partial x^2}(1)^2 + (1)(0) = \frac{\partial^2 u}{\partial x^2}.$$

Substituting the above into equation (8) gives

$$(r - \frac{\sigma^2}{2})\frac{\partial u}{\partial y} + \frac{\partial u}{\partial \tau} - \frac{\sigma^2}{2}\frac{\partial^2 u}{\partial x^2} - (r - \frac{\sigma^2}{2})\frac{\partial u}{\partial y} = 0.$$

The $(r - \frac{\sigma^2}{2})\frac{\partial u}{\partial y}$ term cancels, which leaves us with the equation:

$$\frac{\partial u}{\partial \tau} - \frac{\sigma^2}{2}\frac{\partial^2 u}{\partial x^2} = 0.$$

Rearranging gives the heat equation,

$$\frac{\partial u}{\partial \tau} = \frac{1}{2}\sigma^2\frac{\partial^2 u}{\partial y^2}, \tag{9}$$

as required. □

4.3 Using the Explicit Euler Method to solve the Heat equation

Consider the solution of the heat equation

$$u_t = \frac{1}{4}y_{xx}, \qquad 0 < x < 2, t > 0,$$

for the function $u(x,t)$ with initial condition and boundary conditions given by

$$u(x,0) = 4xe^x(2-x), \qquad 0 < x < 2$$

$$u(0,t) = u(2,t) = 0, \qquad t \leq 0.$$

Using the explicit Euler method, this problem will be solved using five nodes $\{x_0, x_1, x_2, x_3, x_4\} = \{0, 0.5, 1, 1.5, 2\}$ and step-length in time $\Delta t = 0.1$. We approximate

$$u_t(x_j, t_l) = \frac{u(x_j, t_{l+1}) - u(x_j, t_l)}{\Delta t},$$

$$u_{xx}(x_j, t_l) = \frac{u(x_{j+1}, t_l) - 2u(x_j, t_l) + u(x_{j-1}, t_l)}{(\Delta x)^2}.$$

By substituting these approximations into the heat equation, gives the equation:

$$\frac{w_j^{l+1} - w_j^l}{\Delta t} = \alpha^2\frac{w_{j+1}^l - 2w_j^l + w_{j-1}^l}{(\Delta x)^2},$$

where $w_j^0 = u_0(x_j)$, and $w_0^l = w_n^l = 0$. This means that w_j^l is the numerical approximation of $u(x_j, t_l)$. Below is the working out for the solutions $w_1^1, w_2^1, w_3^1, w_1^2, w_2^2, w_3^2$:

$$w_0^0 = 0$$

$$w_1^0 = 4(1)e^1(2-1) = 4e$$

$$w_2^0 = 4(2)e^2(2-2) = 0$$
$$w_3^0 = 4(3)e^3(2-3) = -12e^3$$
$$w_4^0 = 4(4)e^4(2-4) = -32e^4.$$

Let $\mu = \frac{\alpha^2 \Delta t}{(\Delta x)^2}$,where $\alpha^2 = \frac{1}{4}, \Delta t = 0.1$ and $\Delta x = 0.5$. This gives $\mu = 0.1$. Using this to calculate w_1^2, w_2^2, w_3^2 gives,

$$w_1^1 = \mu w_0^0 + (1-2\mu)w_1^0 + \mu w_2^0 = 0.1 * 0 + (1 - 2 * 0.1)4e + 0.1 * 0 = \frac{16e}{5}.$$

$$w_2^1 = \mu w_1^0 + (1-2\mu)w_2^0 + \mu w_3^0 = 0.1 * 4e + (1-2*0.1)0 + 0.1 * -12e^3 = 0.1(4e - 12e^3)$$

$$w_3^1 = \mu w_2^0 + (1-2\mu)w_3^0 + \mu w_4^0 = 0.1 * 0 + (1 - 2 * 0.1) - 12e^3 + 0.1 * 0 = \frac{-48e^3}{5}.$$

Therefore the set of solutions is,

$$w_0^0 = 0,$$
$$w_1^0 = 4e,$$
$$w_2^0 = 0,$$
$$w_3^0 = -12e^3,$$
$$w_4^0 = -32e^4,$$
$$w_1^1 = \frac{16e}{5},$$
$$w_2^1 = 0.1(4e - 12e^3),$$
$$w_3^1 = \frac{-48e^3}{5}.$$

Now solving with a different time and space step and number of nodes, take $\Delta t = 0.25, n = 3, \Delta x = 1$. This gives $\mu = \frac{0.25*0.25}{1^2} = \frac{1}{16}$. Below is the set of solutions

$$w_1^0 = 4e,$$
$$w_2^0 = 0,$$
$$w_3^0 = -12e^3,$$
$$w_1^1 = \frac{7}{8}4e + \frac{1}{16}0 = 9.5139864,$$
$$w_2^1 = \frac{1}{16}4e + \frac{7}{8}0 + \frac{1}{16}(-12e^3) = -14.3846,$$
$$w_3^1 = \frac{1}{16}0 + \frac{7}{8}(-12e^3) = -210.898,$$
$$w_1^2 = 7.4257,$$
$$w_2^2 = -25.173,$$
$$w_3^2 = -185.4349.$$

4.4 Using the Crank-Nicolson method

The explicit Euler method can be calculated using matrices. This also applies to the Crank-Nicolson method. Using the formula for the Crank-Nicolson method,

$$\frac{w_j^{l+1}}{\Delta t}-\frac{1}{2}\alpha^2\left[\frac{w_{j+1}^{l+1}-2w_j^{l+1}+w_{j-1}^{l+1}}{(\Delta x)^2}\right]=\frac{w_j^{l}}{\Delta t}-\frac{1}{2}\alpha^2\left[\frac{w_{j+1}^{l}-2w_j^{l}+w_{j-1}^{l}}{(\Delta x)^2}\right]. \quad (10)$$

By rearranging,

$$w_j^{l+1}-\frac{\alpha^2\Delta t}{2(\Delta x)^2}\left[w_{j+1}^{l+1}-2w_j^{l+1}+w_{j-1}^{l+1}\right]=w_j^l+\frac{\alpha^2\Delta t}{2(\Delta x)^2}\left[w_{j+1}^{l}-2w_j^{l}+w_{j-1}^{l}\right],$$

$$-\frac{\alpha^2\Delta t}{2(\Delta x)^2}\left[w_{j+1}^{l+1}+w_{j-1}^{l+1}\right]+\left[\frac{\alpha^2\Delta t}{(\Delta x)^2}+1\right]w_j^{l+1}=\frac{\alpha^2\Delta t}{2(\Delta x)^2}[w_{j+1}^{l}+w_{j-1}^{l}]+\left[\frac{-\alpha^2\Delta t}{(\Delta x)^2}+1\right]w_j^{l+1}.$$

Let $\mu=\frac{\alpha^2\Delta t}{2(\Delta x)^2}$, this gives,

$$-\mu w_{j+1}^{l+1}+\left[\frac{\mu}{2}+1\right]w_j^{l+1}-\mu w_{j-1}^{l+1}=\mu w_{j+1}^{l}+\left[1-\frac{\mu}{2}\right]w_j^{l}+\mu w_{j-1}^{l}. \quad (11)$$

By putting equation (11) in matrix form, we obtain the matrices

$$\begin{pmatrix} 1+\frac{\mu}{2} & -\mu & & 0 \\ -\mu & 1+\frac{\mu}{2} & -\mu & \\ & \ddots & \ddots & \ddots \\ & -\mu & 1+\frac{\mu}{2} & -\mu \\ 0 & & -\mu & 1+\frac{\mu}{2} \end{pmatrix}\begin{pmatrix} w_1^{l+1} \\ w_2^{l+2} \\ \vdots \\ w_{n-1}^{l+1} \end{pmatrix}=\begin{pmatrix} 1-\frac{\mu}{2} & \mu & & 0 \\ \mu & 1-\frac{\mu}{2} & \mu & \\ & \ddots & \ddots & \ddots \\ & \mu & 1-\frac{\mu}{2} & \mu \\ 0 & & \mu & 1-\frac{\mu}{2} \end{pmatrix}\begin{pmatrix} w_1^{l} \\ w_2^{l} \\ \vdots \\ w_{n-1}^{l} \end{pmatrix}.$$

5 Practical Risk Measurement in Modern Portfolio Theory

Given a portfolio of assets, each asset has a weight representing the budget for an asset. These weights are optimised to fix return and minimise risk to get a set of optimal portfolios.

In this part of the project, we will be looking at S&P500 data again. This time we will get the Adjusted Close data for each asset in S&P500. This is the portfolio of assets and the aim is to optimize it. The outline to optimise the portfolio follows the method:

1. Download data
2. Calculate returns of data
3. Interpolate data if some missing values
4. Calculate correlation, $\hat{P}$, of data
5. Calculate covariance, $\hat{C}$, of data
6. Calculate the optimal allocation vector, $q*$
7. Plot the optimal allocation vector, also known as the efficient frontier.

5.1 Download data

To download the last five years of each asset in the S&P500 data, follow this code:

```
# Load all the tickers from S&P500 data
tickers = pd.read_html('https://en.wikipedia.org/wiki/List
_of_S%26P_500_companies')[0]

# Download data from Yahoo!Finance of all the tickers.
data = yf.download(tickers.Symbol.to_list(),start='2016-11-16',
                   end='2021-11-16')['Adj Close']
```

5.2 Calculating Returns and Data Preparation

The formula to calculate returns is

$$ln\frac{p(n)}{p(n-1)} = ln[p(n)] - ln[p(n-1)].$$

This can be written in code using the following command,

```
returns= np.log(data).diff()
```

This works by taking the natural log of all the data, and then calculating the difference between rows so $p(n)$ and $p(n-1)$.

Due to the differences method, there are missing values at least in the first two rows. Backwards interpolation is used to fix this and reduce the number of missing values (see Part 1). There are still 2522 missing values, therefore we must locate where these values occur in the data. All of the missing values occur in the columns of **BF.B** and **BRK.B** as no financial data was found for these stocks on Yahoo! Finance. The two columns are deleted from the dataset.

5.3 Calculating correlation and covariance

Now the data has been prepared, correlation and covariance can be estimated. The correlation can be estimated using the formula:

$$\hat{\boldsymbol{P}} = \frac{1}{M}\boldsymbol{R}^{\boldsymbol{T}}\boldsymbol{R}, \tag{12}$$

where M is the available returns per assets and $\boldsymbol{R}$ is the normalised returns matrix. From this, the covariance can be estimated by

$$\hat{\boldsymbol{C}} = \boldsymbol{\Sigma}^{\boldsymbol{T}}\hat{\boldsymbol{P}}\boldsymbol{\Sigma}, \tag{13}$$

where $\boldsymbol{\Sigma}$ is the diagonal matrix of volatility and $\hat{\boldsymbol{P}}$ is the estimated correlation. Starting with correlation,

```
# Calculate R by storing the returns_interpolated dataframe
in a numpy array
R = returns_interpolated.to_numpy()

# Calculate M by calculating how many rows there are in
the dataframe
M = len(returns_interpolated.index)

# Use the above to calculate correlation, P_hat by equation (12)
P_hat = (1/M)*np.matmul(R.transpose(), R)
```

Covariance can then be estimated using equation (13).

```
# Calculate volatility matrix
vol= np.diag(returns_interpolated.std().to_numpy())

# Estimate covariance, C_hat, using vol and P_hat
C_hat = np.matmul(vol.transpose(),np.matmul(P_hat,vol))
```

5.4 Solving for the optimal allocation vector

The optimal allocation vector solves for the optimal weights of the portfolio given a target return. The vector can be plotted for varying values of target return, forming the Markowitz bullet. The portfolios on the upper-half of the Markowitz bullet is known as the efficient frontier as they offer the highest expected return for a defined level of risk.

First of all, we will start with solving for the optimal allocation vector. The formula to calculate the optimal allocation vector is as follows:

$$\boldsymbol{q}^* = \frac{\begin{vmatrix} \mu & \mathbf{1}^T\boldsymbol{C}^{-1}\boldsymbol{\mu} \\ 1 & \mathbf{1}^T\boldsymbol{C}^{-1}\mathbf{1} \end{vmatrix} \boldsymbol{C}^{-1}\boldsymbol{\mu} + \begin{vmatrix} \boldsymbol{\mu}^T\boldsymbol{C}^{-1}\boldsymbol{\mu} & \mu \\ \boldsymbol{\mu}^T\boldsymbol{C}^{-1}\mathbf{1} & 1 \end{vmatrix} \boldsymbol{C}^{-1}\mathbf{1}}{\begin{vmatrix} \boldsymbol{\mu}^T\boldsymbol{C}^{-1}\boldsymbol{\mu} & \mathbf{1}^T\boldsymbol{C}^{-1}\boldsymbol{\mu} \\ \boldsymbol{\mu}^T\boldsymbol{C}^{-1}\mathbf{1} & \mathbf{1}^T\boldsymbol{C}^{-1}\mathbf{1} \end{vmatrix}} \tag{14}$$

To simplify equation (14), let $\boldsymbol{q}^*$ be expressed in the form

$$\boldsymbol{q}^* = \frac{\begin{vmatrix} A & B \\ C & D \end{vmatrix} E + \begin{vmatrix} F & G \\ H & I \end{vmatrix} J}{\begin{vmatrix} K & L \\ M & N \end{vmatrix}} \tag{15}$$

where equation (15) equals equation (16). Using Python, solve equation (15).

```
# Define determinant function
def det(a,b,c,d):
  return np.dot(a,d)-np.dot(b,c)
```

```
# Define q* function
def optimal_allocation_vector(target_return):
  ones=np.ones(503)

  # Solve for each part of the equation
  A = target_return
  B = np.dot(ones.transpose(),np.dot(np.linalg.inv(Chat),mean_hat))
  C = ones
  D = np.dot(ones.transpose(),np.dot(np.linalg.inv(Chat),ones))

  E = np.dot(np.linalg.inv(Chat),mean_hat)

  F = np.dot(mean_hat.transpose(),np.dot(np.linalg.inv(Chat),mean_hat))
  G = target_return
  H = np.dot(mean_hat.transpose(),np.dot(np.linalg.inv(Chat),ones))
  I = 1

  J = np.dot(np.linalg.inv(Chat),ones)

  K = np.dot(mean_hat.transpose(),np.dot(np.linalg.inv(Chat),mean_hat))
  L= np.dot(ones.transpose(),np.dot(np.linalg.inv(Chat),mean_hat))
  M= np.dot(mean_hat.transpose(),np.dot(np.linalg.inv(Chat),ones))
  N = np.dot(ones.transpose(),np.dot(np.linalg.inv(Chat),ones))

  # Calculate each of the determinants in equation (15)
  first = det(A,B,C,D)
  second = det(F,G,H,I)
  third = det(K,L,M,N)

  # Calculate final result, q* and return result
  total = (np.dot(first,E)+np.dot(second,J))/third
  return total
```

Now that the optimal allocation vector can be calculated, the next step is to plot the efficient frontier. To do this, the following steps are followed:

1. For multiple target returns, calculate the optimal allocation vector.
2. Calculate the risk of each portfolio that is calculated by the optimal allocation vector.
3. Plot Risk vs. Returns to get the efficient frontier.

```
# Define risk calculation function
def find_risk(q):
  # Find variance of portfolio
  portfolio_variance = np.dot(q.transpose(),np.dot(Chat, q))
```

```
    # Find risk of portfolio
    portfolio_risk = np.sqrt(portfolio_variance)
    return portfolio_risk

# Create empty arrays to store in returns and risks.
returns = []
volatility = []

# For every target return 0-10
for i in np.linspace(0,10,100):
    # Store to returns
    returns.append(i)

    # Find q* for the target return
    q = optimal_allocation_vector(i)

    # Find the corresponding risk of the portfolio
    v = find_risk(q)

    # Store in volatility vector
    volatility.append(v)

# Plot volatility vs. log(returns) to get the efficient frontier.
plt.plot(volatility,np.log(returns),'o',markersize=1)
plt.xlabel('volatility')
plt.ylabel('log(returns)')
```

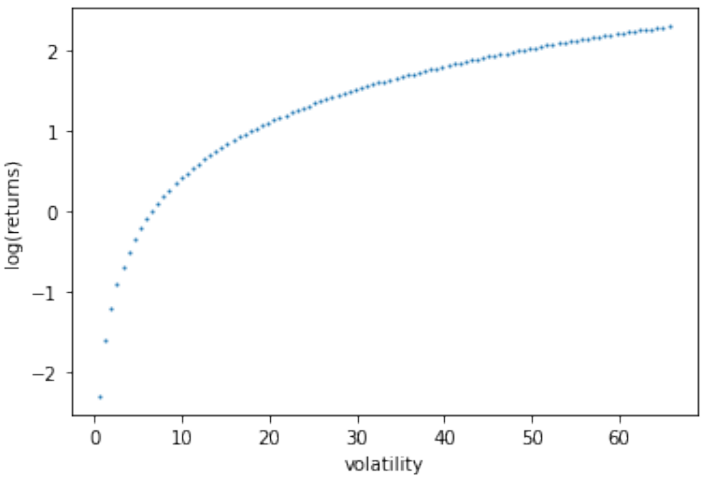

Figure 3: Efficient frontier of S&P 500 data from 16/11/16 to 16/11/21.

From Figure 3, the graph shows a relationship between increasing volatility resulting in higher returns. The issue with the graph in Figure 3 is the size of volatility. When returns increase, volatility increases drastically, which seems unrealistic. This could be due to an issue in the scaling in the data, or perhaps some issues during interpolation and preparing the data set. I also tried preparing the dataset by getting rid of the first two rows with NaN values, and then any other columns with NaN values, but this only resulted in higher volatilities. This shows that the error may lie at this point in the method. Despite this, the efficient frontier is still a good diagram to decide at which point a portfolio could be marked too risky, and which portfolios are too low risk.

6 Conclusion

In this assignment, the techniques involved in pricing options and portfolio optimisation have been discussed. The examples that have been investigated to price options are: binomial trees, Brownian motion, Monte Carlo and finite differences methods. For portfolio optimisation, the estimation of correlation and covariance have been used to find the optimal allocation vector, and then to find the efficient frontier.

By comparing all of the methods (see Section 4.1), it is important to consider the flexibility of the method to add in other factors such as option type and dividends, as well as the error and computational time. Black-Scholes is very accurate, however lacks the flexibility of other methods such as binomial trees, finite difference methods and Monte Carlo. Some methods, such as binomial trees, are easy to understand and can provide useful visualisations for investors, which can also be another positive. However, factoring in time and space complexity is just as important as usually the methods with the most flexibility, visualisations and complex calculations, can take up more time and space.

References

[1] S&P 500 Price, Real-time Quote News - Google Finance.

[2] Ticker symbol.

[3] Financial Data from Yahoo Finance with Python.

[4] Interpolation – Power of Interpolation in Python to fill Missing Values.

[5] Using Interpolation To Fill Missing Entries in Python.

[6] Adjusted Closing Price vs. Closing Price

[7] Monte Carlo and Binomial Simulations for European Option Pricing

[8] Geometric Brownian Motion

[9] Option Pricing Theory

[10] Understanding Stock Splits.

[11] Why does volatility impact call and put options in the same way?

[12] Black-Scholes Model

[13] Option Pricing: Black-Scholes v Binomial v Monte Carlo Simulation