QF205 – (G2)
Computing Technology for Finance
Project Title: Python Programming and its Applications in Option Pricing

AY 2023-24 – Term 2

Prepared for: Professor Zhao Yibao

Date of Submission:

Done by: Group 8

Name	Student ID
Alden Ng	01399819
Goh Isaac	01455235
Jonathan Hsienzheng I'anson Holton	01424959
Premjit Singh Aujala	01414283
Yash Chellani	01410412

Table of Contents

1. Introduction

Traditionally defined, a stock option is a financial instrument that gives its holder the right, but not the obligation to buy or sell a stock. To put it simpler, we illustrate with an example. Say person A wants to buy 1 share of Apple company, thinking that Apple share price will increase in the future, however, for some reason (e.g. not enough capital, not willing to take risk), they do not want to buy an actual share of Apple. Instead they buy an option on the share. The option allows them to buy the share of Apple at $170, but there is no compulsion to do so, i.e. they can choose to not act on the option, and not buy the share at $170. The option acts as a “reservation” on the share. However, as the share is “reserved” and no one else can buy it, this “reservation” will come at a cost – known as a premium. Finding out just how much one should pay for this premium is the scope of our report.

There are various types of “reservations” available. Some allow the holder to exercise them anytime (American option) or exercise only at a fixed date (European option). Some “reservations” allow the holder to buy the underlying share (call option) or sell the underlying (put option).

This report dives into the essence of option pricing, unravelling the complexities that underscore this financial phenomenon. Through the lens of Python programming, we explore three cornerstone methodologies - the Binomial Tree Method, the Monte Carlo Simulation Method, and the Crank-Nicolson Finite Difference Method. Each of these methodologies provides a unique approach to valuing options, reflecting the multifaceted nature of financial markets and the underlying assets. By leveraging Python's computational prowess, this report aims to offer a comprehensive examination of these methods, elucidating their theoretical foundations and practical applications in option pricing.

2. Application

The practical applications of Python programming in financial modelling, particularly in option pricing, extend far beyond academic theory, permeating the decision-making processes of financial institutions and individual investors alike. The methodologies explored in this report are instrumental in tackling the inherent uncertainties of financial markets, enabling practitioners to:

1. **Binomial Tree Method:** This method excels in scenarios where the flexibility to exercise options at discrete time intervals is crucial, such as with American options. It simulates possible future movements of an asset's price over discrete time steps, providing a lattice-based framework that is particularly suited for options that allow early exercise.
2. **Monte Carlo Simulation Method:** Ideal for valuing complex derivatives and options with path-dependent features, this method employs randomness to simulate a vast array of potential future asset price paths. Its strength lies in its versatility and the capacity to model the probabilistic nature of markets, making it invaluable for pricing European options, Asian options, and other exotic derivatives.
3. **Black-Scholes (exact) Method:** The Black-Scholes model is a fundamental framework in financial mathematics for pricing European-style options. Developed by

Fischer Black, Myron Scholes, and Robert Merton in the early 1970s, this model revolutionized the field of finance by providing a closed-form solution for the price of European call options on a stock that does not pay dividends.

4. **Forward-Time Central-Space Explicit Method**: A numerical solution used primarily for solving partial differential equations (PDEs), particularly in the field of financial mathematics for option pricing. This method is part of the finite difference methods used to approximate solutions to PDEs governing options like the Black-Scholes equation for pricing options.
5. **Crank-Nicolson Finite Difference Method:** Another numerical solution to the Black-Scholes equation, this method offers a balanced approach to accuracy and computational efficiency. It is adept at pricing European options where analytic solutions are challenging to obtain, providing a stable and accurate method for solving partial differential equations that describe option pricing models.

3. Algorithm 1: Binomial Tree Method

Full Code

```python
import math
from option_types import Option

def binomial_tree(K: int, T: int, S0: int, r: float, N: int, u: float, d: float, opttype) -> float | str:

    dt = T / N
    q = (math.exp(r * dt) - d) / (u - d)
    disc = math.exp(-r * dt)

    # Initialise stock prices at maturity
    S = [0] * (N + 1)
    for j in range(0, N + 1):
        S[j] = S0 * u ** j * d ** (N - j)
    C = [0] * (N + 1)

    # Calculate option payoff
    for j in range(0, N + 1):
        if opttype == Option.PUT:
            C[j] = max(0, K - S[j])
        elif opttype == Option.CALL:
            C[j] = max(0, S[j] - K)
        else:
            return 'Invalid Option Type'

    # Backward traversal through the tree
    for i in range(N - 1, -1, -1):
        for j in range(0, i + 1):
            S_current = S0 * u ** j * d ** (i - j)
            C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])
            if opttype == Option.PUT:
                C[j] = max(C[j], K - S_current)
            elif opttype == Option.CALL:
                C[j] = max(C[j], S_current - K)
            else:
                return 'Invalid Option Type'

    return C[0]
```

3.1 Import Statements

```
import math
from option_types import Option
```

Purpose: These import statements bring in external modules and custom classes necessary for performing mathematical calculations and defining the option types.
Explanation:
import math: This module provides various mathematical functions like exponentials, logarithms, trigonometric functions, etc., which are essential for calculations involved in finance and option pricing.
from option_types import Option: This imports the Option class or enumeration from the option_types module, which defines the types of options (CALL or PUT) used in the binomial option pricing model.

Let's delve into the programming constructs used in the Option enumeration. We'll examine each part of the code and explain how it contributes to defining the different types of options (CALL and PUT) using the Enum class from the enum module.

```
from enum import Enum

class Option(Enum):
    CALL = 'C'
    PUT = 'P'
```

3.1.1. Import Statement

Purpose: This import statement brings in the Enum class from the enum module, which is necessary for defining enumerations in Python.
3.1.2. Enum Definition: Option
Purpose: This class defines an enumeration called Option, which represents the types of options (CALL and PUT).
Explanation:
Enum: This is the base class for creating enumerations in Python.
Option: This is the name of the enumeration.
CALL and PUT: These are the two members of the enumeration, representing the types of options.
CALL is assigned the value 'C', indicating it represents a call option.
PUT is assigned the value 'P', indicating it represents a put option.

3.2 Function Definition: binomial_tree

```
def binomial_tree(K: int, T: int, S0: int, r: float, N: int, u: float, d: float, opttype) -> float | str:
```

Purpose: This function calculates the price of an option using the binomial tree method.
Explanation:

- The function takes several *parameters* representing the characteristics of the option and the underlying asset, such as strike price (K), time to expiration (T), current price of the underlying asset (S0), risk-free interest rate (r), number of time steps (N), upward and downward movement factors (u and d), and option type (opttype).
- The return type annotation -> float | str indicates that the function returns either a floating-point number (option price) or a string (error message).

3.3 Initialization

```
dt = T / N
q = (math.exp(r * dt) - d) / (u - d)
disc = math.exp(-r * dt)
```

Purpose: These lines initialise *variables* used in the binomial option pricing model.
Explanation:

- *dt*: Represents the time interval between each step in the binomial tree, calculated by dividing the total time to expiration (T) by the number of time steps (N).
- *q*: Represents the probability of an upward movement in the binomial tree, calculated using the risk-neutral probability formula.
- *disc*: Represents the discount factor, calculated using the risk-free interest rate and time interval.

3.4 Creating Data Structures

```
S = [0] * (N + 1)
for j in range(0, N + 1):
    S[j] = S0 * u ** j * d ** (N - j)
C = [0] * (N + 1)
```

Purpose: These lines create *lists* to store asset prices and option prices at each node in the binomial tree. *Lists* can be thought of as numerous items (i.e. data) grouped together.
Explanation:

- *S*: Represents a *list* to store the asset prices at each node in the binomial tree.
- *C*: Represents a *list* to store the option prices at each node in the binomial tree.
- This initialization step sets up the framework for dynamic programming, as it precomputes and stores values that will be used in subsequent calculations, avoiding redundant computations.

3.5 Calculate Option Payoff

```
for j in range(0, N + 1):
    if opttype == Option.PUT:
        C[j] = max(0, K - S[j])
    elif opttype == Option.CALL:
        C[j] = max(0, S[j] - K)
    else:
        return 'Invalid Option Type'
```

Purpose: This *loop* calculates the option payoff at maturity for each node in the binomial tree based on the option type. A *loop* simply repeats the code below it multiple times.
Explanation:

- The loop iterates through each node in the binomial tree (represented by j).
- Depending on the option type (opttype), it calculates the option payoff at maturity using the appropriate payoff function for PUT or CALL options.
- If the option type is neither PUT nor CALL, it returns an error message indicating an invalid option type.

3.6 Backward Traversal

```
# Backward traversal through the tree
for i in range(N - 1, -1, -1):
    for j in range(0, i + 1):
        S_current = S0 * u ** j * d ** (i - j)
        C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])
        if opttype == Option.PUT:
            C[j] = max(C[j], K - S_current)
        elif opttype == Option.CALL:
            C[j] = max(C[j], S_current - K)
        else:
            return 'Invalid Option Type'
```

Purpose: This part implements the backward traversal through the binomial tree to calculate the option prices at earlier time steps.
Explanation:

- The outer *loop* iterates backwards through the time steps of the binomial tree, starting from the second-to-last step down to the initial step (i).
    - The inner *loop* iterates through each node at the current time step (j).
        - It calculates the current asset price (S_current) at each node.
        - By traversing the tree backwards and reusing previously calculated values stored in the C array, the code avoids recalculating the same values multiple times, thereby improving efficiency.
        - It calculates the option price at the current node using the risk-neutral pricing formula and updates the option price list (C).
        - Depending on the option type, it checks and updates the option price based on the option payoff condition.

3.7 Return Result

```
return C[0]
```

Purpose: This line *returns* the option's price at time zero (initial step).
Explanation: It returns the first element of the option price list (C), which represents the price of the option at the initial step of the binomial tree.

Overall, this code utilises dynamic programming techniques in the context of the binomial option pricing model. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems and solving each subproblem only once, storing the solutions to avoid redundant calculations.

4. Algorithm 2: Monte Carlo Simulation

The code is designed to simulate the possible future prices of a stock and then use those simulations to estimate the value of what's called a European call option. This type of financial option gives the holder the right, but not the obligation, to buy a stock at a certain price (the strike price) on a specific date in the future (the time to maturity).

4.1 Data Structures

The code uses a few key data structures from Python's NumPy library:

- *Arrays* (numpy.ndarray): These are like super-charged lists. The code uses arrays for the following:
  - paths: Stores all the different simulated stock price paths. It has dimensions of (number of simulations, number of steps + 1).
  - rand: Stores random numbers that drive the stock price fluctuations.
  - payoffs: Stores the calculated value of the option at the end of each simulated path.

4.2 Functions

The code is broken down into two helpful functions:

4.2.1 generate_stock_paths()

```
def generate_stock_paths(S0, mu, sigma, T, dt, num_simulations):
    """
    Generates multiple stock price paths using Geometric Brownian
Motion.

    Args:
        S0 (float): Initial stock price.
        mu (float): Expected drift (annualized return).
        sigma (float): Volatility (annualized standard deviation).
        T (float): Time to maturity of the option (in years).
        dt (float): Time increment for simulation.
        num_simulations (int): Number of simulated paths.

    Returns:
        numpy.ndarray: Array of simulated stock price paths.
    """

    num_steps = int(T / dt)
    paths = np.zeros((num_simulations, num_steps + 1))
    paths[:, 0] = S0

    rand = np.random.standard_normal((num_simulations, num_steps))
```

```
    for t in range(1, num_steps + 1):
        paths[:, t] = paths[:, t - 1] * np.exp((mu - 0.5 * sigma ** 2) *
dt + sigma * sqrt(dt) * rand[:, t - 1])

    return paths
```

This function simulates multiple possible price movements for a stock over a specific time period. It uses a mathematical model called Geometric Brownian Motion (GBM), which captures how stock prices can fluctuate randomly but with a tendency to grow over time.

Purpose: Simulates how a stock's price might change over time, taking into account things like average return, volatility, and randomness.

- **Number of Steps:**
  - `num_steps = int(T / dt)` calculates the number of steps needed for the simulation. Here, `T` is the total time and `dt` is the time between each step (like time slices). The smaller `dt`, the more precise the simulation, but it also takes longer to compute. Imagine a movie - more frames (smaller time slices) create a smoother picture.
- **Initializing Stock Prices:**
  - `paths = np.zeros((num_simulations, num_steps + 1))` creates a NumPy array named `paths`. This array will store the simulated stock prices for all simulations and all time steps.
  - `paths[:, 0] = S0` sets the initial stock price (`S0`) for each simulation at the first time step ([:, 0] selects all rows and the 0th column).
- **Random Price Changes:**
  - `rand = np.random.standard_normal((num_simulations, num_steps))` generates random numbers using the standard normal distribution (a bell-shaped curve centered around 0). These numbers represent the unexpected ups and downs in the stock price.
- **Simulating Each Step:**

```
for t in range(1, num_steps + 1):
    paths[:, t] = paths[:, t - 1] * np.exp((mu - 0.5 * sigma ** 2) * dt
+ sigma * sqrt(dt) * rand[:, t - 1])
```

  - This loop iterates through each time step (`t`) and updates the stock price for each simulation (`paths[:, t]`). Here's what the magic happens:
  - `paths[:, t - 1]`: This refers to the stock price at the previous time step for each simulation.
  - `np.exp(...)`: This calculates the exponential term based on several factors:
  - `mu`: This is the expected average return of the stock per time step. If `mu` is positive, the stock price tends to go up on average over time.
  - `sigma`: This is the volatility, which represents the amount of unexpected fluctuation in the stock price. Higher volatility means more ups and downs.

- `dt`: This is the time step size. As we discussed earlier, a smaller `dt` leads to a more precise but computationally expensive simulation.
- `rand[:, t - 1]`: This injects the random factor from the previously generated random numbers, influencing how much the stock price goes up or down in this time step.
- `sigma * sqrt(dt)`: This scales the random numbers based on the volatility (`sigma`) and the time step (`dt`). A larger volatility or smaller time step will lead to a bigger impact of the random factors.

❖ **Example:**

Imagine a stock priced at $100 (`S0`) with a maturity of 1 year (`T`). We run 100 simulations (`num_simulations`) with 12 monthly time steps (`dt = T / num_steps = 1 / 12`). This means `paths` will be a 100 (simulations) x 13 (time steps) array. At each time step, the price in each simulation will be adjusted based on the average return (`mu`), volatility (`sigma`), and random factors (`rand`).

Key Point

- The `generate_stock_paths` function doesn't tell us if the stock price will go up or down for sure. It creates multiple possible scenarios (paths) that account for both the expected rise in price and the random fluctuations.

4.2.2 monte_carlo_option_price()

```
def monte_carlo_option_price(S0, K, T, r, sigma, num_simulations,
opt_type):
    """
    Prices a European call option using Monte Carlo simulation.

    Args:
        S0 (float): Initial stock price.
        K (float): Strike price of the option.
        T (float): Time to maturity of the option (in years).
        r (float): Risk-free interest rate (annualized).
        sigma (float): Volatility (annualized standard deviation).
        num_simulations (int): Number of simulated paths.

    Returns:
        float: Estimated price of the call option.
    """

    dt = T / num_simulations  # Smaller time steps for better accuracy
    paths = generate_stock_paths(S0, r, sigma, T, dt, num_simulations)

    # Payoffs at maturity
```

```
    if opt_type == Option.CALL:
        payoffs = np.maximum(paths[:, -1] - K, 0)

    elif opt_type == Option.PUT:
        payoffs = np.maximum(K - paths[:, -1], 0)

    # Discounting back to present value and averaging
    option_price = exp(-r * T) * payoffs.mean()

    return option_price
```

This function takes several inputs (stock price, strike price, etc.) and, with the help of the stock price simulations, estimates the price of a European call option.

Purpose: Figures out the estimated price of a European call option.

- **Smaller Time Steps:**
  - `dt = T / num_simulations` calculates the time step size for the simulation. Remember, a smaller time step generally leads to a more accurate option price estimation.
- **Generating Stock Paths:**
  - `paths = generate_stock_paths(S0, r, sigma, T, dt, num_simulations)` calls the function we explained earlier. It generates multiple potential trajectories that the stock price could follow over time. Notice that we use the risk-free interest rate (`r`) as the expected return here. This is because we are valuing the option from a risk-neutral perspective.
- **Final Stock Prices:**
  - `payoffs = np.maximum(paths[:, -1] - K, 0)` finds the payoff of the option at the end of the simulation. Here's how:
    - `paths[:, -1]`: This selects the final (last time step) simulated stock prices from each of the scenarios stored in the `paths` array.
    - `-K`: Subtracts the strike price (`K`) from the final stock prices. This represents the potential profit if the option is exercised.
    - `np.maximum(..., 0)`: This ensures the payoff isn't negative because a call option holder can simply choose not to exercise the option if it is unprofitable, leading to a payoff of 0.
- **Discounting to Present Value:**
  - `option_price = exp(-r * T) * payoffs.mean()`:
    - `payoffs.mean()`: Calculates the average payoff across all the simulated stock paths.
    - `exp(-r * T)`: Applies a discount factor to bring the future payoffs to their equivalent present-day value. The risk-free interest rate (`r`) is used here as the discounting rate.

❖ **Example:**

Let's use our previous example where we ran 100 simulations of possible stock prices over 1 year:

1. The function calculates the value of exercising a call option at the end of each simulation. If the final stock price (`paths[:, -1]`) is higher than the strike price (`K`), the call option has value; otherwise, it's worthless.
2. It takes the average of these payoffs across all the simulations.
3. Finally, this average is adjusted back to its present value using the risk-free interest rate.

Key Point

1. The `monte_carlo_option_price` function doesn't give you a single answer for the option's price. Due to the randomness involved in simulations, the estimated price might change slightly each time you run it. The more simulations (`num_simulations`) you perform, the more likely you are to converge on a reliable estimate.

4.3 Analysis

Let's analyse how the example code works in conjunction with the functions we've explained.

```
# Example usage
S0 = 100  # Initial stock price
K = 100   # Strike price
T = 1     # Time to maturity (years)
r = 0.06  # Risk-free interest rate
sigma = 0.2  # Volatility
num_simulations = 100  # Increase for better accuracy
opt_type = None

call_price = monte_carlo_option_price(S0, K, T, r, sigma,
num_simulations, opt_type)
print("Estimated price of the European call option:", call_price)
```

1. `S0 = 100 # Initial stock price` Sets the initial price of the underlying stock to $100.
2. `K = 100 # Strike price` Sets the strike price of the call option to $100. This means the option holder has the right to buy the stock at $100 on the maturity date.
3. `T = 1 # Time to maturity (years)` Sets the time to maturity of the option to 1 year. The option can only be exercised at the end of this period.
4. `r = 0.06 # Risk-free interest rate` Sets the risk-free interest rate to 6% per year. This rate is used to discount the future value of the option's payoff back to the present day.
5. `sigma = 0.2 # Volatility` Sets the volatility of the stock to 20%. A higher volatility implies more significant and unexpected swings in the stock's price.
6. `num_simulations = 100 # Increase for better accuracy` Sets the

number of Monte Carlo simulations to 100. Running more simulations generally improves the accuracy of the option price estimate, but it also takes more time to compute.

7. `call_price = monte_carlo_option_price(S0, K, T, r, sigma, num_simulations, opt_type` This line is the core of the calculation. Let's break down what happens inside the function call:
    - `generate_stock_paths(...)` is executed, producing 100 hypothetical paths the stock price might follow over the next year. These paths take into account expected return, volatility, and randomness.
    - For each of the simulated paths, a final payoff is calculated. The payoff is only positive if the stock ends up higher than the strike price of $100.
    - The average of the final payoffs across all simulations is calculated and adjusted to its present value using the risk-free interest rate.
    - This calculated average price is assigned to the call_price variable.
8. `print("Estimated price of the European call option:", call_price` This line neatly prints the result to the screen. The value that will be printed is the estimated price of the European call option, taking into account the initial stock price, strike price, time to maturity, risk-free interest rate, and volatility.

5. Algorithm 3: Black Scholes and 2 numerical solutions: FTCS & Crank-Nicolson

5.1 Import statements

```
1  import numpy as np
2  from scipy.linalg import solve
3  from scipy.stats import norm
```

The *scipy* library is a fundamental toolset for scientific computing in Python, offering modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, and more.

scipy.linalg : used for linear algebra functions
scipy.stats : is the module in SciPy that includes functions and classes for probability distributions

Required variables

```
6   # Initial setup
7   S0 = 20          # current price
8   K = 10           # exercise/strike price
9   T = 0.25         # expiry time
10  r = 0.1          # no-risk interest rate
11  sigma = 0.4      # volatility of underlying asset
12
13  # preset
14  N = 2000         # number of time steps
15  M = 200          # number of space grids
16
17  # ensure S0 is well within the range
18  S_max = max(4 * K, S0 * 2)
```

Number of time-steps (N) and number of space grids(M) are preset within the module (shown later) as they are a measure of how much accuracy the user wants at the cost of computational resources. N = 2000 & M = 200 is within standards for achieving balance between accuracy and computational demands.

The choice of S_max being set to the maximum of 4 * K or S0 * 2 ensures that the range of stock prices considered is sufficient to cover meaningful movements in the stock price while keeping the computation manageable.

Black Scholes Function

```
69  def calculate_exact_black_scholes(S0, K, T, r, sigma):
70
71      d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
72
73      d2 = d1 - sigma * np.sqrt(T)
74
75      C = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
76
77      return max(C, 0)
```

Calculating d1 and d2

- *np.log*:
    - This calculates the natural logarithm of the current stock price divided by the strike price.

- *Arithmetic operations*:
    - +, -, *, and / are standard arithmetic operators for addition, subtraction, multiplication, and division, respectively.

- *np.sqrt*:
    - Calculates the square root, another function from numpy. It's used here to adjust the time factor (T) under the volatility term.

These two variables, d1 and d2, are intermediate calculations required to find the probabilities that the option will finish in the money (profitable) at expiration.

Calculating the Call Option Price, C

```
75      C = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
```

- *norm.cdf*:
    - This is the cumulative distribution function for a standard normal distribution from the scipy.stats module. It calculates the probability that a normally distributed random variable is less than or equal to d1 or d2. These probabilities are essential for determining the expected benefit of owning the stock or the strike price at expiration.

- *np.exp*:
    - Calculates the exponential of the given expression, which in this context is used to discount the strike price back to its present value using the risk-free rate.

Ensuring Non-negative Output

```
77          return max(C, 0)
```

max():

This function returns the maximum of the two values. Since an option's price cannot be negative (you wouldn't pay to hold an option that will be worthless), this ensures the function always returns a non-negative value.

FTCS

```
22  def calculate_ftcs(S0, K, T, r, sigma):
23      Click to collapse the range.
24      # potentially preset
25      N = 2000        # number of time steps
26      M = 200         # number of space grids
27
28      # ensure S0 is well within the range
29      S_max = max(4 * K, S0 * 2)
30
31      dt = T / N
32      s = np.linspace(0, S_max, M+1)  # stock price range
33      C = np.clip(s - K, 0, None)      # initial condition
34      index = np.arange(1, M)
35
36      for n in range(N):
37          C[1:-1] = (0.5 * (sigma**2 * index**2 * dt - r * index * dt) * C[:-2] +
38                     (1 - sigma**2 * index**2 * dt - r * dt) * C[1:-1] +
39                     0.5 * (sigma**2 * index**2 * dt + r * index * dt) * C[2:])
40
41      s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
42      return C[s_idx]  # Return the option price at S0
```

N,M and S_max are defined here as mentioned earlier.

Creating the Grids:

```
31      dt = T / N
32      s = np.linspace(0, S_max, M+1)  # stock price range
```

- dt:
  - This calculates the size of each time step by dividing the total time by the number of steps.
- *np.linspace*:
  - Generates an array of values from 0 to S_max evenly spaced over M+1 points. This represents possible stock prices at each point in space.

Setting Initial Array Conditions:

```
33      C = np.clip(s - K, 0, None)      # initial condition
```

- *np.clip*:
    - Limits the values in an array. Is used here to ensure that option values cannot go below zero, implementing the payoff condition at maturity for a European call option, which is the maximum of (stock price - strike price) or 0.

Updating the Option Values:

```
34      index = np.arange(1, M)
35
36      for n in range(N):
37          C[1:-1] = (0.5 * (sigma**2 * index**2 * dt - r * index * dt) * C[:-2] +
38                     (1 - sigma**2 * index**2 * dt - r * dt) * C[1:-1] +
39                     0.5 * (sigma**2 * index**2 * dt + r * index * dt) * C[2:])
40
41      s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
42      return C[s_idx]  # Return the option price at S0
```

- for-loop:
    - Iterates over each time step, updating the option values.The equation inside the loop calculates new values for the option at each point using the values from the previous time step. The terms involve calculations based on the volatility (sigma), the risk-free rate (r), and the discretization steps.

Finding the Option Value at Current Stock Price:

```
41      s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
42      return C[s_idx]  # Return the option price at S0
```

- *np.searchsorted*:
    - Finds the appropriate index in the array s where the current stock price S0 fits. This is used to locate the grid point closest to the current stock price.
- return *C[s_idx]*:
    - Checks Array C at *index* [s_idx] and returns the option price corresponding to the current stock price from the array of calculated option values.

Crank-Nicolson

```
44  def calculate_crank_nicolson(S0, K, T, r, sigma):
45
46      N = 2000
47      M = 200
48      S_max = max(4 * K, S0 * 2)
49
50      dt = T / N
51      s = np.linspace(0, S_max, M+1)
52      C = np.clip(s - K, 0, None)
53      index = np.arange(1, M)
54
55      alpha = dt / 4 * (r * index - sigma**2 * index**2)
56      beta = dt / 2 * (r + sigma**2 * index**2)
57      gamma = -dt / 4 * (r * index + sigma**2 * index**2)
58
59      A = np.diag(1 + beta) + np.diag(gamma[:-1], k=1) + np.diag(alpha[1:], k=-1)
60
61      for t in range(N):
62          b = np.dot(np.diag(1 - beta) + np.diag(-gamma[:-1], k=1) + np.diag(-alpha[1:], k=-1), C[1:-1])
63          b[-1] += -2 * gamma[-1] * (S_max - K)
64          C[1:-1] = solve(A, b)
65
66      s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
67      return C[s_idx]  # Return the option price at S0
```

*lines 46 - 53, 6-67 are identical to FTCS

Coefficients for the Finite Difference Scheme

```
55      alpha = dt / 4 * (r * index - sigma**2 * index**2)
56      beta = dt / 2 * (r + sigma**2 * index**2)
57      gamma = -dt / 4 * (r * index + sigma**2 * index**2)
```

- *The Greeks* ~ alpha, beta, gamma:
    - These coefficients are used in the finite difference approximation, affecting how each point's value is influenced by its neighbours in the grid.

Constructing the Tridiagonal Matrix A

```
59      A = np.diag(1 + beta) + np.diag(gamma[:-1], k=1) + np.diag(alpha[1:], k=-1)
```

- *np.diag*:
    - create diagonal matrices for the central, upper, and lower diagonals. The matrix A is crucial for the Crank-Nicolson scheme, representing the linear system that must be solved at each time step.

Time-stepping Loop

```
61      for t in range(N):
62          b = np.dot(np.diag(1 - beta) + np.diag(-gamma[:-1], k=1) + np.diag(-alpha[1:], k=-1), C[1:-1])
63          b[-1] += -2 * gamma[-1] * (S_max - K)
64          C[1:-1] = solve(A, b)
```

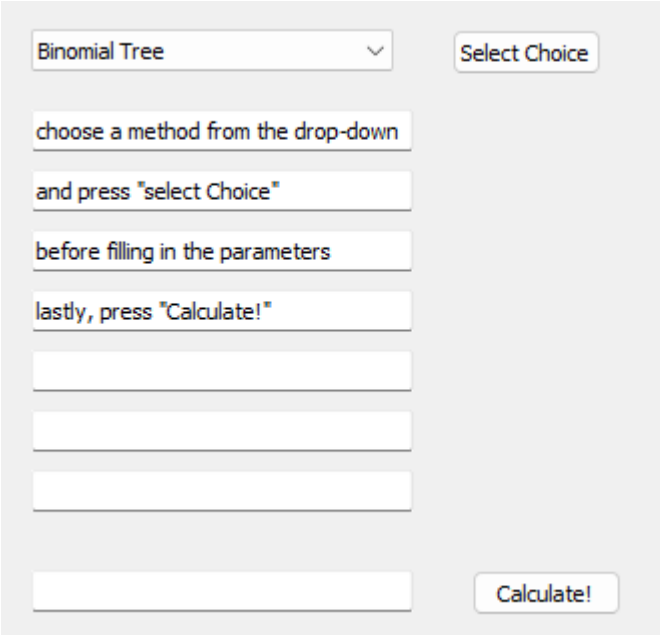
- np.dot:
    - Computes the product of two arrays (matrix multiplication), used here to apply the finite difference operator to the option values.

- solve:
    - A function from scipy.linalg that solves the matrix equation Ax = b. This step is key to the Crank-Nicolson method, solving for new option values at each time step.

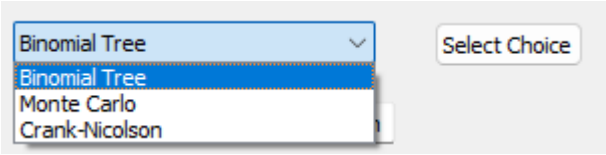
6. User Interface (UI)

User Interface walkthrough

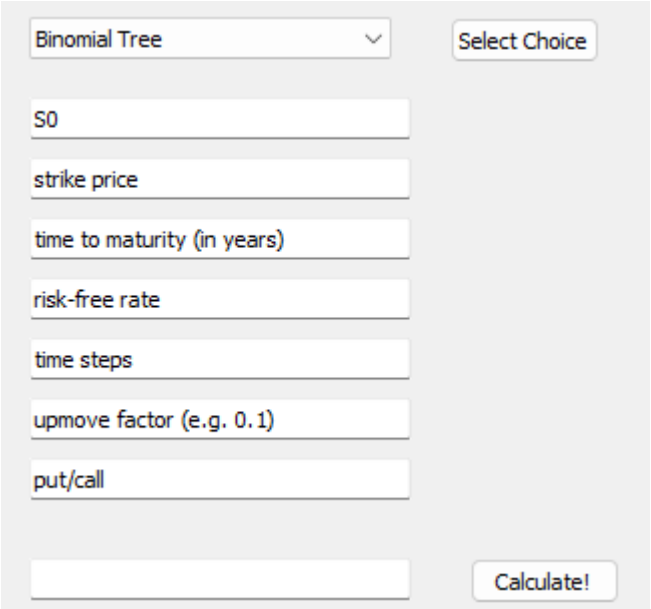
When running the code, the user interface appears as follows to the user:



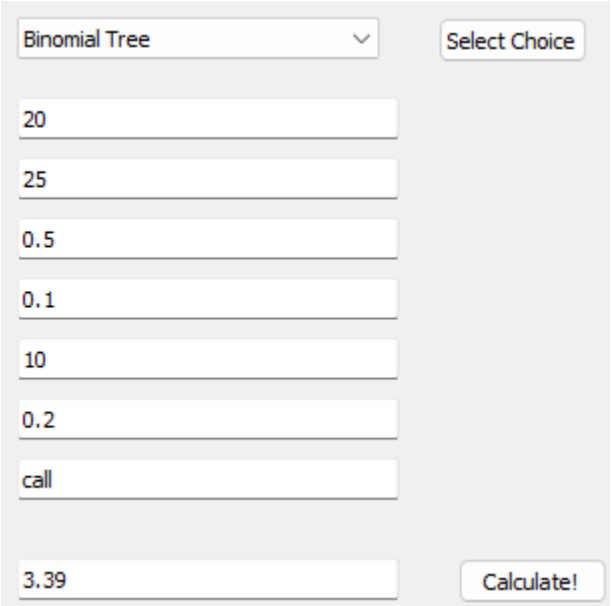
At the top is a dropdown box that allows the user to select their preferred method of calculating the option price, and below it are instructions to the user. For example, we select “Binomial Tree” from the list and click “Select Choice” as follows:



The UI switches to show the relevant inputs necessary for the Binomial Tree option pricing method.



Typing in some example values and pressing “Calculate!”, we get the option price:



6. UI

We first import the necessary *libraries* – these can be thought of as extensions to the Python programming language.

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QApplication
3  from PyQt5 import uic
4  import logging
5
6  from option_types import Option
7  from binomial_tree_options_pricing import binomial_tree
8  from monte_carlo_options_pricing import monte_carlo_option_price
9  from blackScholes import calculate_ftcs,calculate_crank_nicolson,calculate_exact_black_scholes
```

The first chunk of imports are for displaying the UI, whilst the next chunk is for option pricing the option pricing methods.

The last two lines help to load a predesigned template for the UI into the program.

```
14  # Setup logging
15  logger = logging.getLogger('OptionPricingApp')
16  logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(name)s - %(levelname)s - %(message)s")
```

- Logging
    - Setup custom logging for easy code tracing and debugging. format="%(asctime)s - %(name)s - %(levelname)s - %(message)s") tells Python to log “info” level of severity or higherhow we would like to receive the information, with time, followed by the logger name (OptionPricingApp), the information level name (info), and the error message that might appear.

```
19  class Main(QMainWindow, Ui_MainWindow):
20      def __init__(self):
21          super().__init__()
22          self.setupUi(self)
23
24          self.choose.clicked.connect(self.choose_method)
25
26          self.calculate.clicked.connect(self.calculate_out)
```

- class *Main*:
    - defines a class named Main. In Python, a class is a blueprint for creating objects (a particular data structure) that consists of data and functionality together.

- QMainWindow, Ui_MainWindow:
    - These are classes from which Main inherits. QMainWindow is a type of window that provides a framework for building the application's main window. Ui_MainWindow is a class generated by PyQt's uic tool from a .ui file designed in Qt Designer, containing the layout and widgets of the GUI

- def *__init__*(self):
    - This method is a special method in Python known as the *constructor*. It is automatically invoked when a new object of the class is created. Its primary role is to initialize the object's state or data.

- *super*().__init__():
    - This line calls the constructor of the *parent class* (QMainWindow and Ui_MainWindow). It's necessary to ensure that the initialization associated with those parent classes is correctly performed.

- self.setupUi(self):

- This method is part of the Ui_MainWindow class. It sets up the user interface from the pre-designed .ui file. When called, it configures all the widgets (buttons, labels, text fields, etc.) as designed in the Qt Designer.

```
28      def choose_method(self):
29          method = self.choice.currentText()
30          self.S0.setText("S0")
31          self.rf.setText("risk-free rate")
32          self.strike.setText("strike price")
33          self.time.setText("time to maturity (in years)")
34          self.opt_type.setText("put/call")
35
36          if method == "Binomial Tree":
37              self.wildcard1.setText("time steps")
38              self.wildcard2.setText("upmove factor (e.g. 0.1)")
```

- def choose_method(*self*):
  - defines a method named choose_method within the class. The self parameter refers to the *instance* of the class, allowing the method to access attributes (variables) and other methods of the class.

- self.*choice*:
  - Retrieves user 'choice' imput.

- .*currentText*():
  - This method retrieves the text of the *currently selected item* in the dropdown menu, which indicates the option pricing model chosen by the user

- *setText*:
  - This method sets the text displayed in various text input fields on the GUI. These fields allow users to input essential parameters such as the current stock price (S0), risk-free rate (rf), strike price (strike), time to maturity (time)

- *Conditional Statements* (if, elif, else):
  - These statements control the flow of the program based on the condition being true. They are used to direct applicationI to the specific logic of the selected option pricing method.

- *'wildcards'*:
  - additional configurable fields (wildcard1, wildcard2) that are used for method-specific parameters. For example, "time steps" and "upmove factor" for the Binomial Tree, or "sigma" (volatility) and the number of simulations for Monte Carlo. Used in conjunction with hide.
- hide:

- This method is called to hide the wildcard2 field when it is not needed, as in the case of Crank-Nicolson, FTCS, and Black-Scholes, where only one additional parameter (sigma) is necessary.

```
51      def calculate_out(self):
52          try:
53              S0 = float(self.S0.text())
54              rf = float(self.rf.text())
55              strike = float(self.strike.text())
56              time = float(self.time.text())
57              method = self.choice.currentText()
58
59              if self.opt_type.text().upper() == "CALL":
60                  opt_type = Option.CALL
61              elif self.opt_type.text().upper() == "PUT":
62                  opt_type = Option.PUT
63
64              # Define a dictionary to act as a switch-case
65              method_function = {
66                  "Binomial Tree": lambda: binomial_tree(
67                      K=strike, T=time, S0=S0, r=rf,
68                      N=int(self.wildcard1.text()),
69                      u=1 + float(self.wildcard2.text()),
70                      d=1 / (1 + float(self.wildcard2.text())),
71                      opttype=opt_type
72                  ),
```

- Method Definition:
  - calculate_out is a method in a class that presumably inherits from a PyQt5 class. It's triggered by an event in the GUI (like clicking a 'Calculate' button).

- Exception Handling:
  - The try...except block is used to handle exceptions—errors that occur during the execution of the code within the try block. If an error occurs, the code within the except block runs, logging the error. This prevents the application from crashing and provides debugging information.

- Data Retrieval and Conversion:
  - Values are retrieved from GUI elements (like text boxes and a dropdown menu) and converted into appropriate data types (floats for numerical inputs and strings for text). This conversion is crucial as mathematical operations on these values are performed later.

- Conditional Statements(using *enums*):
  - The code checks the text of an option type selector in the GUI and assigns an enum value based on the selection. Enums are a way to organize code

involving fixed sets of constants (like 'CALL' and 'PUT' options), improving readability and maintainability.

- *Dictionary as a Switch-Case*:
    - This dictionary acts like a switch-case statement found in other programming languages. Each key-value pair corresponds to a method name and a lambda function poised to execute the respective pricing model's function.

- *Lambda Functions*:
    - These are small anonymous functions defined with the keyword lambda. They are used here to defer the execution of the pricing functions until needed, which is when method_function[method]() is called.

- Displaying Results(*round*):
    - After computing the result, it is rounded and converted to a string, then displayed in the GUI. This ensures that the user sees a neatly formatted result.

```
90              # Execute the appropriate function based on the user's choice
91              result = method_function[method]()
92              logger.info('result: ' + str(result))
93              self.output.setText(str(round(result, 4)))
```

- Logging:
    - logs the result.

Error Handling

```
95          except Exception as e:
96              logger.error(f"Error calculating option price: {e}")
```

- Handles error via logging to terminal

7. Conclusion

In conclusion, the Python programming language has many uses. In this report, we use it in the context of financial option pricing as well as building a UI that allows for easy use of the code as a calculator.

8. Appendix

Github link:
https://github.com/yashchellani/qf205

combined.py

```
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic
import logging

from option_types import Option
from binomial_tree_options_pricing import binomial_tree
from monte_carlo_options_pricing import monte_carlo_option_price
from blackScholes import
calculate_ftcs,calculate_crank_nicolson,calculate_exact_black_scholes

qtCreatorFile = "design.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

# Setup logging
logger = logging.getLogger('OptionPricingApp')
logging.basicConfig(level=logging.INFO, format="%(asctime)s - %(name)s -
%(levelname)s - %(message)s")

class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        self.choose.clicked.connect(self.choose_method)

        self.calculate.clicked.connect(self.calculate_out)

    def choose_method(self):
        method = self.choice.currentText()
        self.S0.setText("S0")
        self.rf.setText("risk-free rate")
        self.strike.setText("strike price")
        self.time.setText("time to maturity (in years)")
        self.opt_type.setText("put/call")

        if method == "Binomial Tree":
            self.wildcard1.setText("time steps")
            self.wildcard2.setText("upmove factor (e.g. 0.1)")
```

```
        elif method == "Monte Carlo":
            self.wildcard1.setText("sigma aka volatility")
            self.wildcard2.setText("no. of simulations")

        else:  # Applies to Crank-Nicolson, FTCS, and Black-Scholes
            self.wildcard1.setText("sigma aka volatility")
            self.wildcard2.hide()  # Hide second wildcard if not needed


    def calculate_out(self):
        try:
            S0 = float(self.S0.text())
            rf = float(self.rf.text())
            strike = float(self.strike.text())
            time = float(self.time.text())
            method = self.choice.currentText()

            if self.opt_type.text().upper() == "CALL":
                opt_type = Option.CALL
            elif self.opt_type.text().upper() == "PUT":
                opt_type = Option.PUT

            # Define a dictionary to act as a switch-case
            method_function = {
                "Binomial Tree": lambda: binomial_tree(
                    K=strike, T=time, S0=S0, r=rf,
                    N=int(self.wildcard1.text()),
                    u=1 + float(self.wildcard2.text()),
                    d=1 / (1 + float(self.wildcard2.text())),
                    opttype=opt_type
                ),
                "Monte Carlo": lambda: monte_carlo_option_price(
                    S0=S0, K=strike, T=time, r=rf,
                    sigma=float(self.wildcard1.text()),
                    num_simulations=int(self.wildcard2.text()),
                    opttype=opt_type
                ),
                "Crank-Nicolson": lambda: calculate_crank_nicolson(
                    S0, strike, time, rf, float(self.wildcard1.text())
                ),
                "FTCS": lambda: calculate_ftcs(
                    S0, strike, time, rf, float(self.wildcard1.text())
```

```
                ),
                "Black-Scholes": lambda: calculate_exact_black_scholes(
                    S0, strike, time, rf, float(self.wildcard1.text())
                )
            }

            # Execute the appropriate function based on the user's choice
            result = method_function[method]()
            logger.info('result: ' + str(result))
            self.output.setText(str(round(result, 4)))

        except Exception as e:
            logger.error(f"Error calculating option price: {e}")


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main = Main()
    main.show()
    sys.exit(app.exec_())
```

binomial_tree_options_pricing.py

```
import math
from option_types import Option

def binomial_tree(K: int, T: int, S0: int, r: float, N: int, u: float, d:
float, opttype) -> float | str:

    """

    Args:
        S0 (float): Initial stock price.
        K (float): Strike price of the option.
        T (float): Time to maturity of the option (in years).
        r (float): Risk-free interest rate (annualized).
        N (int): time steps
        u (float): up-factor
        d = 1/u (float): ensure recombining tree

    Returns:
        float: Estimated price of the call option.
```

```
    """

    dt = T / N
    q = (math.exp(r * dt) - d) / (u - d)
    disc = math.exp(-r * dt)

    # Initialise stock prices at maturity
    S = [0] * (N + 1)
    for j in range(0, N + 1):
        S[j] = S0 * u ** j * d ** (N - j)

    # Calculate option payoff
    C = [0] * (N + 1)
    for j in range(0, N + 1):
        if opttype == Option.PUT:
            C[j] = max(0, K - S[j])
        elif opttype == Option.CALL:
            C[j] = max(0, S[j] - K)
        else:
            return 'Invalid Option Type'

    # Backward traversal through the tree
    for i in range(N - 1, -1, -1):
        for j in range(0, i + 1):
            S_current = S0 * u ** j * d ** (i - j)
            C[j] = disc * (q * C[j + 1] + (1 - q) * C[j])
            if opttype == Option.PUT:
                C[j] = max(C[j], K - S_current)
            elif opttype == Option.CALL:
                C[j] = max(C[j], S_current - K)
            else:
                return 'Invalid Option Type'

    return C[0]
```

blackScholes.py

```
import numpy as np
from scipy.linalg import solve
from scipy.stats import norm
```

```
# Initial setup
S0 = 20         # current price
K = 10          # exercise price
T = 0.25        # expiry time
r = 0.1         # no-risk interest rate
sigma = 0.4     # volatility of underlying asset

# potentially preset
N = 2000        # number of time steps
M = 200         # number of space grids

# ensure S0 is well within the range
S_max = max(4 * K, S0 * 2)



def calculate_ftcs(S0, K, T, r, sigma):

    # potentially preset
    N = 2000        # number of time steps
    M = 200         # number of space grids

    # ensure S0 is well within the range
    S_max = max(4 * K, S0 * 2)

    dt = T / N
    s = np.linspace(0, S_max, M+1)  # stock price range
    C = np.clip(s - K, 0, None)      # initial condition
    index = np.arange(1, M)

    for n in range(N):
        C[1:-1] = (0.5 * (sigma**2 * index**2 * dt - r * index * dt) * C[:-2]
+
                   (1 - sigma**2 * index**2 * dt - r * dt) * C[1:-1] +
                   0.5 * (sigma**2 * index**2 * dt + r * index * dt) * C[2:])

    s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
    return C[s_idx]  # Return the option price at S0

def calculate_crank_nicolson(S0, K, T, r, sigma):

    N = 2000
```

```
    M = 200
    S_max = max(4 * K, S0 * 2)

    dt = T / N
    s = np.linspace(0, S_max, M+1)
    C = np.clip(s - K, 0, None)
    index = np.arange(1, M)

    alpha = dt / 4 * (r * index - sigma**2 * index**2)
    beta = dt / 2 * (r + sigma**2 * index**2)
    gamma = -dt / 4 * (r * index + sigma**2 * index**2)

    A = np.diag(1 + beta) + np.diag(gamma[:-1], k=1) + np.diag(alpha[1:],
k=-1)

    for t in range(N):
        b = np.dot(np.diag(1 - beta) + np.diag(-gamma[:-1], k=1) +
np.diag(-alpha[1:], k=-1), C[1:-1])
        b[-1] += -2 * gamma[-1] * (S_max - K)
        C[1:-1] = solve(A, b)

    s_idx = np.searchsorted(s, S0)  # Find the index closest to S0
    return C[s_idx]  # Return the option price at S0

def calculate_exact_black_scholes(S0, K, T, r, sigma):
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    C = S0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    return max(C, 0)

# Display results
if __name__ == '__main__':
    print("FTCS Option Price at S0:", calculate_ftcs(S0, K, T, r, sigma))
    print("Crank-Nicolson Option Price at S0:", calculate_crank_nicolson(S0,
K, T, r, sigma))
    print("Exact Black-Scholes Option Price at S0:",
calculate_exact_black_scholes(S0, K, T, r, sigma))
```

crank-nicolson.py

```
import numpy

from scipy.linalg import solve
```

```
from scipy.stats import norm

T = 0.25       # expiry time
r = 0.1        # no-risk interest rate
sigma = 0.4    # volatility of underlying asset
E = 10.        # exercise price
S_max = 4*E    # upper bound of price of the stock (4*E)

def FTCS(C, N, M, dt, r, sigma):
    """using forward-time central-space scheme to solve the Black-Scholes
equation for the call option price

    Arguments:
        C:       array of the price of call option
        N:       total number of time steps
        M:       total number of spatials grids
        dt:      time step
        r:       no-risk interest rate
        sigma:   volatility of the stock

    Returns:
        C:       array of the price of call option
    """
    index = numpy.arange(1,M)
    for n in range(N):

        C[1:-1] = 0.5 * (sigma**2 * index**2 * dt - r*index*dt) * C[0:-2] \
             +       (1 - sigma**2* index**2 *dt - r*dt) * C[1:-1]   \
             + 0.5 * (sigma**2 * index**2 * dt + r*index*dt) * C[2:]
    return C


N = 2000       # number of time steps
M = 200        # number of space grids
dt = T/N       # time step
s = numpy.linspace(0, S_max, M+1)   # spatial grid (stock's price)

# initial condition & boundary condition
C = s - E
C = numpy.clip(C, 0, S_max-E)

C_exp = FTCS(C, N, M, dt, r, sigma)
```

```
print ('the price of the call option should be around {}, \
if the current price of stock is 20 dollar.'.format(C_exp[int(M/2)]))


def LHS_matrix(M, alpha, beta, gamma):
    """generate and return the LHS coefficient matrix A.

    Arguments:
        M:       total number of spatials grids
        alpha:   array of coefficients on lower diagnoal
        beta:    array of coefficients on diagnoal
        gamma:   array of coefficients on upper diagnoal

    Returns:
        A:       LHS coefficient matrix
    """
    # diagonal
    d = numpy.diag(1+beta)
    # upper diagonal
    ud = numpy.diag(gamma[:-1], 1)
    # lower diagonal
    ld = numpy.diag(alpha[1:], -1)

    A = d +ud +ld
    return A

def RHS(C, alpha, beta, gamma, S_max, E):
    """generate and return the RHS vector b.

    Arguments:
        C:       array of the price of call option at previous time step
        alpha:   array of coefficients on lower diagnoal
        beta:    array of coefficients on diagnoal
        gamma:   array of coefficients on upper diagnoal
        S_max:   upper bound of stock price
        E:       exercise price

    Returns:
        b:       RHS vector
    """
    # diagonal of A_star
    d = numpy.diag(1-beta)
```

```
    # upper diagonal of A_star
    ud = numpy.diag(-gamma[:-1], 1)
    # lower diagonal of A_star
    ld = numpy.diag(-alpha[1:], -1)

    A_star = d + ud + ld
    b = numpy.dot(A_star,C[1:-1])
    # add BC for the right bound (the last element)
    b[-1] += -2*gamma[-1] * (S_max-E)

    return b

def CrankNicolson(C, A, N, alpha, beta, gamma, S_max, E):
    """using Crank-Nicolson scheme to solve the Black-Scholes equation for the
call option price.

    Arguments:
        C:       array of the price of call option
        A:       LHS coefficient matrix
        N:       total number of time steps
        alpha:   array of coefficients on lower diagnoal
        beta:    array of coefficients on diagnoal
        gamma:   array of coefficients on upper diagnoal
        S_max:   upper bound of stock price
        E:       exercise price

    Returns:
        C:       array of the price of call option
    """
    for t in range(N):
        b = RHS(C, alpha, beta, gamma, S_max, E)
        # use numpy.linalg.solve
        C[1:-1] = solve(A,b)
    return C

N = 200        # number of time steps
dt = T/N       # time step

# initial condition & boundary condition
C = s - E
C = numpy.clip(C, 0, S_max-E)
```

```
# calculating the coefficient arrays
index = numpy.arange(1,M)

alpha = dt/4 * (r*index - sigma**2*index**2)
beta = dt/2 * (r + sigma**2*index**2)
gamma = -dt/4 * (r*index + sigma**2*index**2)

A = LHS_matrix(M, alpha, beta, gamma)
C_imp = CrankNicolson(C, A, N, alpha, beta, gamma, S_max, E)
print ('the price of the call option should be around {}, \
if the price of stock is 20 dollar.'.format(C_imp[int(M/2)]))



C_exact = numpy.zeros(M+1)

d1 = (numpy.log1p(s/E) + (r+0.5*sigma**2)*T) / (sigma * numpy.sqrt(T))
d2 = d1 - (sigma * numpy.sqrt(T))
C_exact = s * norm.cdf(d1) - E*numpy.exp(-r*T) * norm.cdf(d2)
C_exact = numpy.clip(C_exact, 0, numpy.inf)

# FTCS:           C(S=20, t=0) = 10.2470001259
# Crank Nicolson: C(S=20, t=0) = 10.2469995107
# Exact Solution: C(S=20, t=0) = 10.2469009391
```

monte_carlo_options_pricing.py

```
import numpy as np
from math import exp, sqrt
from option_types import Option

# Example usage
S0 = 100  # Initial stock price
K = 100   # Strike price
T = 1     # Time to maturity (years)
r = 0.06  # Risk-free interest rate
sigma = 0.2  # Volatility
num_simulations = 100  # Increase for better accuracy

def generate_stock_paths(S0, mu, sigma, T, dt, num_simulations):
    """
    Generates multiple stock price paths using Geometric Brownian Motion.

    Args:
```

```
        S0 (float): Initial stock price.
        mu (float): Expected drift (annualized return).
        sigma (float): Volatility (annualized standard deviation).
        T (float): Time to maturity of the option (in years).
        dt (float): Time increment for simulation.
        num_simulations (int): Number of simulated paths.

    Returns:
        numpy.ndarray: Array of simulated stock price paths.
    """

    num_steps = int(T / dt)
    paths = np.zeros((num_simulations, num_steps + 1))
    paths[:, 0] = S0

    rand = np.random.standard_normal((num_simulations, num_steps))
    for t in range(1, num_steps + 1):
        paths[:, t] = paths[:, t - 1] * np.exp((mu - 0.5 * sigma ** 2) * dt +
sigma * sqrt(dt) * rand[:, t - 1])

    return paths

def monte_carlo_option_price(S0, K, T, r, sigma, num_simulations, opttype):
    """
    Prices a European call option using Monte Carlo simulation.

    Args:
        S0 (float): Initial stock price.
        K (float): Strike price of the option.
        T (float): Time to maturity of the option (in years).
        r (float): Risk-free interest rate (annualized).
        sigma (float): Volatility (annualized standard deviation).
        num_simulations (int): Number of simulated paths.

    Returns:
        float: Estimated price of the call option.
    """

    dt = T / num_simulations  # Smaller time steps for better accuracy
    paths = generate_stock_paths(S0, r, sigma, T, dt, num_simulations)

    # Payoffs at maturity
```

```
    if opttype == Option.CALL:
        payoffs = np.maximum(paths[:, -1] - K, 0)

    elif opttype == Option.PUT:
        payoffs = np.maximum(K - paths[:, -1], 0)

    # Discounting back to present value and averaging
    option_price = exp(-r * T) * payoffs.mean()

    return option_price

if __name__ == '__main__':
    opttype = Option.CALL
    call_price = monte_carlo_option_price(S0, K, T, r, sigma, num_simulations,
opttype)
    print("Estimated price of the European call option:", call_price)
```

option_types.py

```
from enum import Enum

class Option(Enum):
    CALL = 'C'
    PUT = 'P'
```