**MSCF Financial Computing I**

**Mini 1, 2022**

**Homework 2**

***Due At 11:59 pm Sunday, Sept. 11, 2021***

***You will lose 10 points per hour after that time***

1. **European Options (20 points)**

In this part of the homework, you will continue developing code for pricing “plain vanilla” European Call and European Put options. By “plain vanilla” we mean that the underlying asset pays no dividend, that the volatility of the underlying asset’s price movements is constant throughout time, that the payoff of the call/put is made at expiration time T and cannot be collected at any prior time, that the risk-free interest rate is constant, and the asset price evolution follows a geometric Brownian motion.

I have provided my **OptionPrices2.py** solutions file from Homework 1 as a starting point. If you prefer, you may start with your own **OptionPrices.py** solution file and rename it as **OptionPrices2.py**.

1. As a method of the **EuropeanCallOption** class, implement a Black-Scholes-Merton European Call pricing function, **BSMPrice()**, using the closed-form solution shown on Week 2 Part 3 Lecture slide 5.

*Notice* the two **import** statements at the top of the **OptionPrices2.py** file: we have imported the **log()** (natural log), **exp()** (*ex*), and **sqrt()** (square root) functions from module **scipy**, and the **norm** (Normal CDF) function from **scipy.stats**, so that you can call these functions without using a module name prefix.

Uncomment the code for part 1.e that tests the **BSMPrice()** function. Save and test. What do you observe?

1. The option price will decrease as volatility decreases, and increase as volatility increases. Add tests in your program to show the price of the European Call on the $50 stock with $50 strike, for volatilities of .05, .10, .20, .40, .80, and 1.60, and 3.20. Use the **binomialPrice**() function with 1000 steps rather than the **BSMPrice()** function. Save and test. Is the price linear or non-linear with increasing volatility? Try plotting the results using **matplotlib** (you don’t need to turn in your plot).
2. Implement a **EuropeanPutOption** class, which is essentially the same as the **EuropeanCallOption** class but with a different payoff function: you make money if the stock price goes *below* the strike, whereas with a call you make money if the stock price goes *above* the strike:

Put payoff = **Max{K - ST, 0}**

You can define the **EuropeanPutOption** class and member function(s) in **OptionPrices.py**, below or above the **EuropeanCallOption** class code. In the testing code, add the same series of tests for the **EuropeanPutOption** that we performed on the **EuropeanCallOption**. Save and test.

1. For the **EuropeanPutOption** class, implement a **BSMPrice()** function that computes the option price according to the Black-Scholes-Merton formula for a European Put. (You will need to look up the formula.)

Confirm that the **BSMPrice** is very close to the **binomialPrice** (with 1000 time steps) for the put options you have defined in this part of the homework.

1. ERI, the Economic Research Institute, provides a convenient online BSM calculator, with the same units that we have used in our European option classes.

**https://www.erieri.com/blackscholes**

Confirm that your binomial (with large N) and BSM prices match (within a penny) the

prices produced by this calculator. (Make a comment about this in your source code.)

1. **American Options (30 points)**

In this part of the homework, you will develop code for pricing “plain vanilla” American Call and American Put options. By “plain vanilla” we mean that the underlying asset pays no dividend, that the volatility of the underlying asset’s price movements are constant throughout time, and so forth.

Unlike a European Call or Put, an American Call or Put can be exercised ***at any time step prior to and including expiration time.*** Because of this, there are no closed-form formulas for the prices of American Options. The Binomial Tree model will still work, at the dual costs of more time and less precision.

In this part of the homework, you will add code to your existing **OptionPrices2.py** code file.

1. The expiration time value of an American Call or Put—the expiration time payoff—is the same as the expiration time value of a European Call or Put, respectively. But since an American Call or Put *can be exercised at any time step*, the backward induction formula at the interior nodes of the Binomial Tree is different. For the European Call (ECi,j) we had:

For the American Call (ACi,j) we have:

That is, if St – K is greater than the value of continuing to hold the option, we will exercise early at time t. The formula for the American Put is similarly changed.

Copy and modify your **EuropeanCall** and **EuropeanPut** class code to create **AmericanCall** and **AmericanPut** classes.

Copy the testing code that you wrote for the European Call and European Put, *excluding* the code for testing the BSM formulas, and modify the code to test the American Call and American Put prices. Save and test. What do you observe?

1. Find an online American option pricing tool, and compare results.
2. **Binomial Tree: Inheritance and Vectorization (35 points)**
3. You will have noticed that your binomial tree implementations of the **EuropeanCallOption**, **EuropeanPutOption**, **AmericanCallOption**, and **AmericanPutOption** classes in Parts 1 and 2 were almost completely identical *except* for the final time payoff computation and the backward induction computation. This duplication of code has both conceptual and maintainability problems: if you discover a bug in one, you need to make identical changes in all four (which has a high probability of introducing more bugs!); if you need to add a feature, you need to add the same feature in four places; and so forth.

These options are all examples of what are called “plain vanilla options” with simple contract characteristics. In a new code file named **VanillaOptions.py**, define a class named **PlainVanillaOption** that contains all of the code common to the four option classes from Parts 1 and 2, then use *inheritance* to define **EuropeanCallOption** to be “a kind of” **PlainVanillaOption** with specific payoff and backward induction methods that can be passed down to the base class **binomialPrice** method. Write test code for your **EuropeanCallOption** pricing similar to that from Parts 1 and 2. Save and test.

Then do likewise to specify **EuropeanPutOption**, **AmericanCallOption**, and **AmericanPutOption** as “kinds of” **PlainVanillaOption**. Write test code, save and test.

1. NumPy’s *vectorized* functions on **ndarray**s are vastly more efficient than loops on **list** items. Copy your **VanillaOptions.py** file to **VanillaOptions\_vectorized.py**. Replace the **list**-based **binomialPrice** code with **ndarray**-based code using vectorized functions.

***Hints:*** We created our **list**-of-**list**s-based binomial tree to reflect how binomial trees of stock prices forward in time and option values backward in time are usually displayed in book and classroom diagrams: with boxes containing both the stock price and the option price. Since items in an **ndarray** must all have the same data type and **ndarray** is defined to be as efficient as possible with built-in data types like **int32** and **float64**, you will get better performance if you use two *separate* **ndarray** objects of **float64** values for the stock prices and for the option values.

Also, using a **list**-of-**list**s enabled us to build a triangular binomial tree, with 1 node at the top and N+1 nodes at the bottom, where N is the number of time steps in the tree. A 2-dimensional **ndarray** must be rectangular, not triangular. But this is okay: just use the lower triangular part of the square **ndarray** object to store your stock prices or option values, and ignore the items above the diagonal.

Save and test your NumPy-based code.

1. The **time** module has a **time()** function that returns the current time in seconds since the *epoch*, the point where time starts. (Often the epoch is the UNIX standard: January 1, 1970, 00:00:00 UTC, but it might be something else on your system.) An approximate way of measuring the time in seconds that it takes for a function to run is this:

**import time**

**before\_foo = time.time()**

**foo() # call the function foo**

**after\_foo = time.time()**

**print('foo() took', (after\_foo - before\_foo),**

**'seconds to run.')**

Add timing code to a 1000-step **EuropeanCallOption** test from your Homework 2 code, and to the same tests in your **VanillaOptions.py** file and your **VanillaOptions\_vectorized.py** file. Report your three timing tests in a comment in your **VanillaOptions\_vectorized.py** file. Does using inheritance have an impact on runtime in this case? What impact does vectorized NumPy code have?

1. **Simulation (15 points)**
2. Your **VanillaOptions\_vectorized.py** code has both **binomialPrice()** and **BSMPrice()** functions for the **EuropeanCallOption** and **EuropeanPutOption** classes.

Add **simPrice()** functions for the **EuropeanCallOption** and **EuropeanPutOption** classes that uses ***simulation*** to price these options. Each **simPrice()** function should take a **precision** keyword parameter with a default value of **0.005** that determines the width of the 95% Confidence Interval around the computed price. That is, by default, there should be a 0.95 probability that the true price is within the interval (computed\_price +/-0.005).

Add test cases similar to those you have used for the other pricing methods. Save and test.

Perform timing tests for the **EuropeanCallOption** price using the same contract definition parameters you used in part 1, above, for **simPrice()** with the default precision and with precisions of 0.01 and 0.001. Report these timing results in a comment in your code.

***REMEMBER*** to put all team members’ names (Andrew IDs) into your source code file.Put your **OptionPrice2.py**, **VanillaOptions.py** and **VanillaOptions\_vectorized.py** files into a **Team***N***\_HW2.zip** archive, where *N* is your team number, and upload to Canvas.