Name: Vincent Weimer

Date: 20201230

**Groups A&B – Exact Pricing Methods**

**Readme.md – Summary and Outline of the design for Option Pricing**

In the project OptionPricing.sln you will find the following class structure and design concerning pricing European and American options:

Base Class

* Option.hpp

Derived Classes

* EuropeanOption.hpp
* PerpetualAmericanOption.hpp

Global Functions:

* D1
* MeshArray
* Put-call parity functions

I have chosen for this design for multiple reasons. The base Option class has been built so that other types of Options can be derived from this class and inheritance can be applied. To make it maintainable, I state all the option parameters (T, K, S, sig, r, b, type) in the base class and made them protected. In this way, we do not have to state them double in each derived Option class and only the derived classes still have access to these data members. There is one main function that calculates Option prices for Call and Put options. This has been made a pure virtual function in the Base class, which makes it an abstract class. The function is also overloaded, such that takes in an S option parameter argument and 1 function that doesn’t need an input argument. This makes it flexible for a user to choose from which Pricing function is needed in which situation. Also, these functions must be overwritten in each of the specific Option classes that derive from the base class.

Furthermore, I have implemented a Print() function that is similar to the ToString() functions that were implemented throughout the course. When calling this function, we can easily see what type of Option we are dealing with and what its parameters are. Also, I implemented getters and setters in the Option class such that any derived class can use this functionality.   
  
In the derived classes, Functionality regarding pricing options for a range of Options and for an inputted Option matrix is implemented. As an assumption, I have only implemented the Greeks in the European Option class for this exercise.

In the global functions file, AccompaniedOptionFunctions, we find functions for D1, Put-call Parity, and Mesh Array. D1 is calculated both in the Option Price and in the Greeks, so it makes sense to not write this formula out in every public member function, but this can just be called as a global function. Same for the MeshArray() function. This function does not below to a class and can be called from each derived Option Class or in Main as well. In this Project, both European Options and American Options call the MeshArray() function.

**Group A - Exact Solutions of One-Factor Plain Options**

**Part 1:**

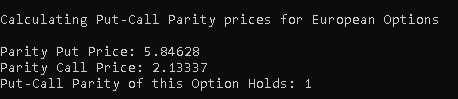
1. *Implement the above formulae for call and put option pricing using the data sets Batch 1 to Batch 4. Check your answers, as you will need them when we discuss numerical methods for option pricing.*

I implemented the Black-Scholes formulae for both Put and Call options in my designed Option classes. Please see results in the table below. The calculated Call and Put prices in C++ match the Call and Put prices from the homework document.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Batch** | **Call Price (Document)** | **Put Price (Document)** | **Call Price (C++)** | **Put Price (C++)** |
| 1 | 2.13337 | 5.84628 | 2.13337 | 5.84628 |
| 2 | 7.96557 | 7.96557 | 7.96557 | 7.96557 |
| 3 | 0.204058 | 4.07326 | 0.204058 | 4.07326 |
| 4 | 92.17570 | 1.24750 | 92.1757 | 1.2475 |

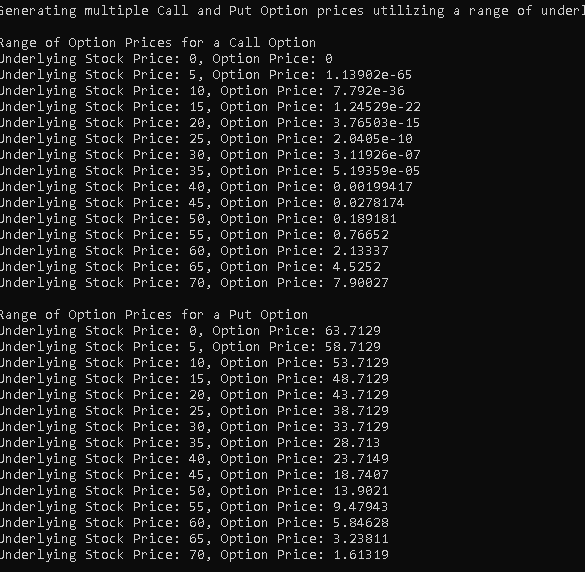
1. *Apply the put-call parity relationship to compute call and put option prices. For example, given the call price, compute the put price based on this formula using Batches 1 to 4. Check your answers with the prices from part a). Note that there are two useful ways to implement parity: As a mechanism to calculate the call (or put) price for a corresponding put (or call) price, or as a mechanism to check if a given set of put/call prices satisfy parity. The ideal submission will neatly implement both approaches*

I implemented the put-call parity relationship for both approaches as global functions in a separate file, AccompaniedOptionFunctions, and calculated Put prices given a Call price and Put prices given a Call price. Also, I verified Put-Call parity via a global implemented function, and for all 4 batches put-call parity relationship holds. See below the output for Batch 1:



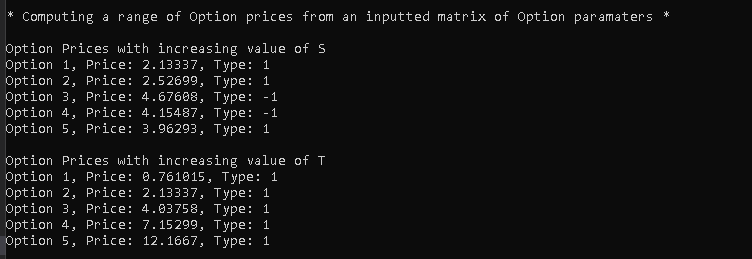
1. *Say we wish to compute option prices for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector. This entails calling the option pricing formulae for each value S and each computed option price will be stored in a std::vector object. It will be useful to write a global function that produces a mesh array of doubles separated by a mesh size h.*

To answer this question I implemented a global function, called MeshArray() to create a vector of underlying values of S. Then, I created a public member function, called RangeofOptionPrices() in EuropeanOption class that calls the global function MeshArray() and calculates the option prices given the values of S in the mesh array and prints these out to the screen:



1. *Now we wish to extend part c and compute option prices as a function of i) expiry time, ii) volatility, or iii) any of the option pricing parameters. Essentially, the purpose here is to be able to input a matrix (vector of vectors) of option parameters and receive a matrix of option prices as the result. Encapsulate this functionality in the most flexible/robust way you can think of.*

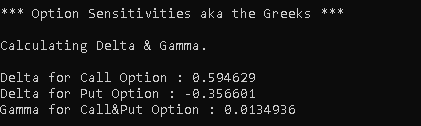
To answer this question, I manually create a matrix (vector of vectors) with an increasing value of a particular parameter while keeping the other parameters constant. Then, I created a function that takes in and processes this matrix and calculate option prices and store them in a vector. Then print the prices to the screen and the type of the option (Call, Put). In the second example below, I calculated option prices for a matrix of options with an increasing value of T for Batch 1. What we can see is that when T is small (0.1), the Option price is low and when T is large (2), the Option value increases to 12.1667. This can be explained by Option pricing theory implying that the value of an option increases if the time to maturity increases.



**Part 2: Option Sensitivities, aka the Greeks**

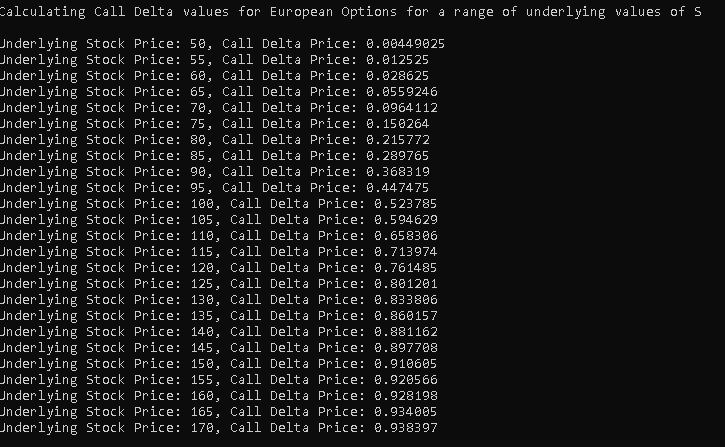
1. *Implement the above formulae for gamma for call and put future option pricing using the data set: K = 100, S = 105, T = 0.5, r = 0.1, b = 0 and sig = 0.36. (exact delta call = 0.5946, delta put = -0.3566).*

I have implemented the formulae for Delta and Gamma in the EuropeanOption Class. Please see below the results for above stated Option parameters:



1. *We now use the code in part a to compute call delta price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and it entails calling the above formula for a call delta for each value S and each computed option price will be store in a std::vector object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.*

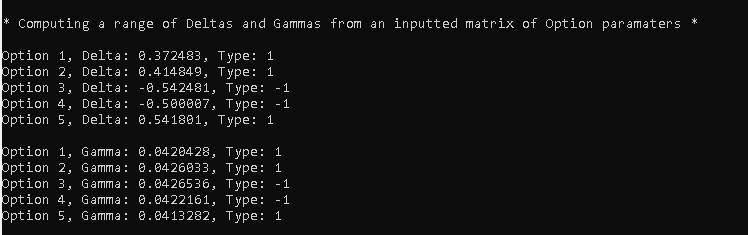
To answer this question, I reuse the global function MeshArray() that is called in the public member function RangeofDeltas() in EuropeanOption Class. Please see below the results where interval h is set to 5:



As we can observe, when the underlying stock price S = 105, we see the same value for Delta that we have calculated in part a.

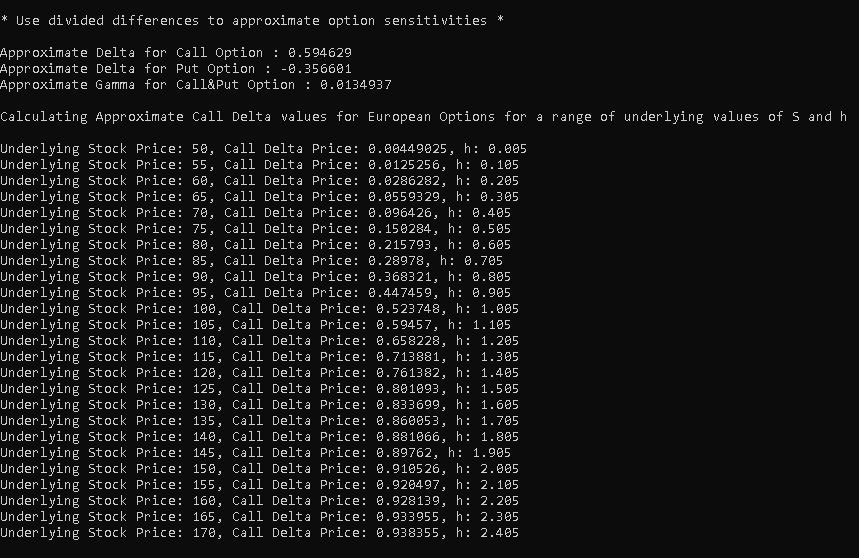
1. *Incorporate this into your above matrix pricer code, so you can input a matrix of option parameters and receive a matrix of either Delta or Gamma as the result*.

To answer this question, I reuse the same matrix created in Part a.d and reuse the function that takes in the matrix and calculates the corresponding option prices. For Delta, you must input “Delta” as a parameter and for Gamma you have to input “Gamma” in the function. Please see the results below:



1. *We now use divided differences to approximate option sensitivities. In some cases, an exact formula may not exist (or is difficult to find) and we resort to numerical methods. In general, we can approximate first and second-order derivatives in S by 3-point second order approximations, for example:*

Firstly, I implemented the formulas for ApproximateDelta and ApproximateGamma in my EuropeanOption class design. I then first calculated Approximate Delta and Approximate Gamma using the formulas with the same parameters as in Question a with a very low h (0.0005). Here, we can see that they are almost identical. Then, I used the same settings to calculate Approximate Call Deltas for a range of S and a changing h for every calculated Delta. What we can see is that when h is low, Approximate Delta is very close to the Exact Delta, but starts to diverge more and more when h starts to increase:



**Group B: Perpetual American Options**

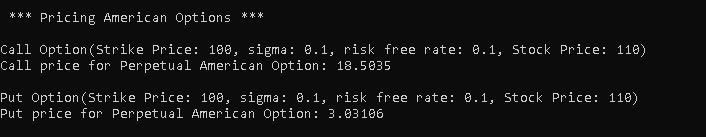
1. *Program the above formulae, and incorporate into your well-designed options pricing classes.*

I have implemented the formulae for Perpetual American Options in my class designs.

1. *Test the data with K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110 (check C = 18.5035, P = 3.03106).*

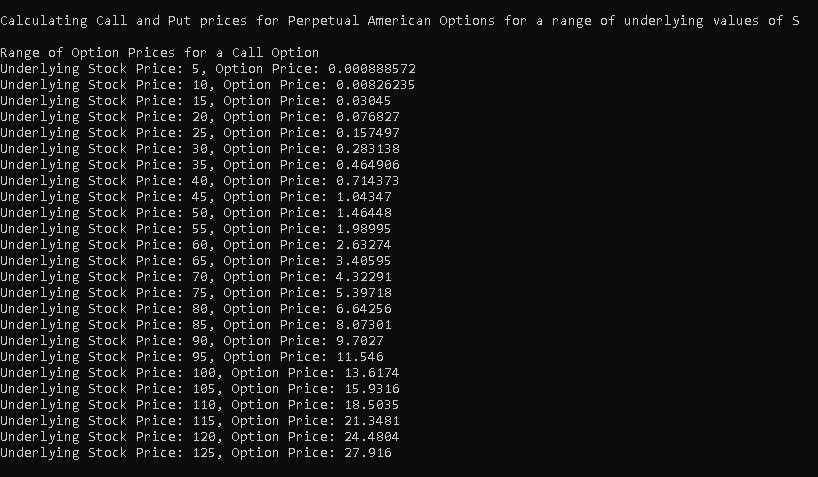
After running the program, I get the same results as stated in the homework handout.

See below the output from the program:



1. *We now use the code in part a) to compute call and put option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and this exercise entails calling the option pricing formulae in part a) for each value S and each computed option price will be stored in a std::vector object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.*

For this exercise I could reuse again the global function MeshArray(). This formula is then called in RangeofOptionPrices() in the PerpetualAmericanOption class. In Main, the user can give it the required parameters to output the underlying values of S and the associated Option prices. Please see below the output from the program:





From these outputs we can also see that American Options are more expensive than European Options. This is because European Options can only be exercised at time of expiration. Especially Perpetual American Put options are expensive when S is much lower than K.

1. *Incorporate this into your above matrix pricer code, so you can input a matrix of option parameters and receive a matrix of Perpetual American option prices.*

For this question, I created a separate matrix of Option parameters with an increasing value of S while keeping the other parameters constant. It is built such that it will calculate any option (Call, Put) with an increasing or decreasing value of any of the Option parameters.

