C++ Programming for Financial Engineering Level 9 Group A&B Writeup

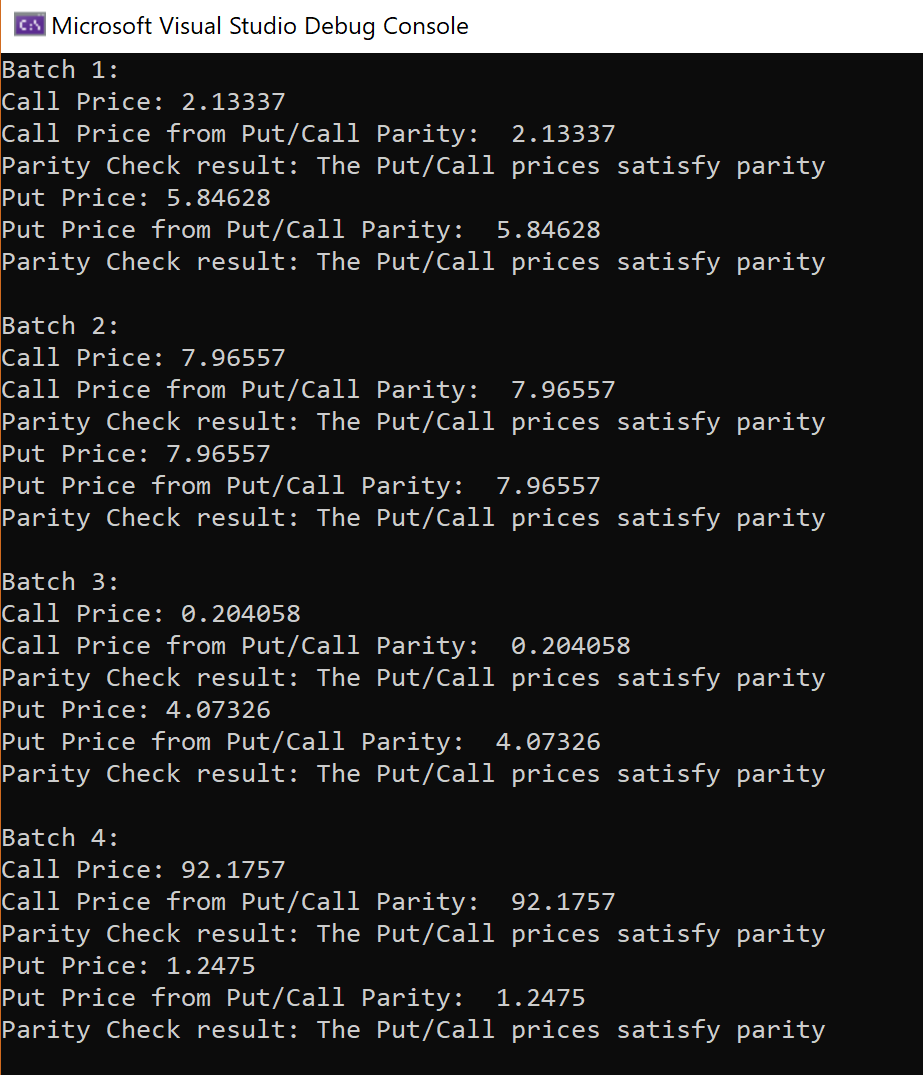
QuantNet

Minghan Li

10/27/2019

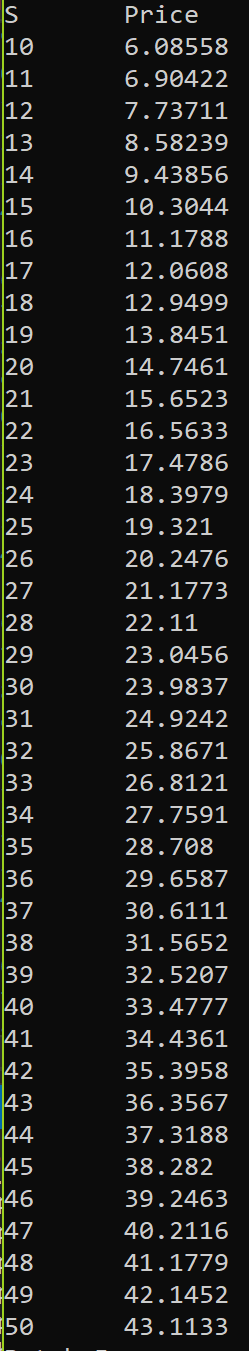
**Part One: Questions and Answers**

1. **Exact Solutions of One-Factor Plain Options**
2. and b)

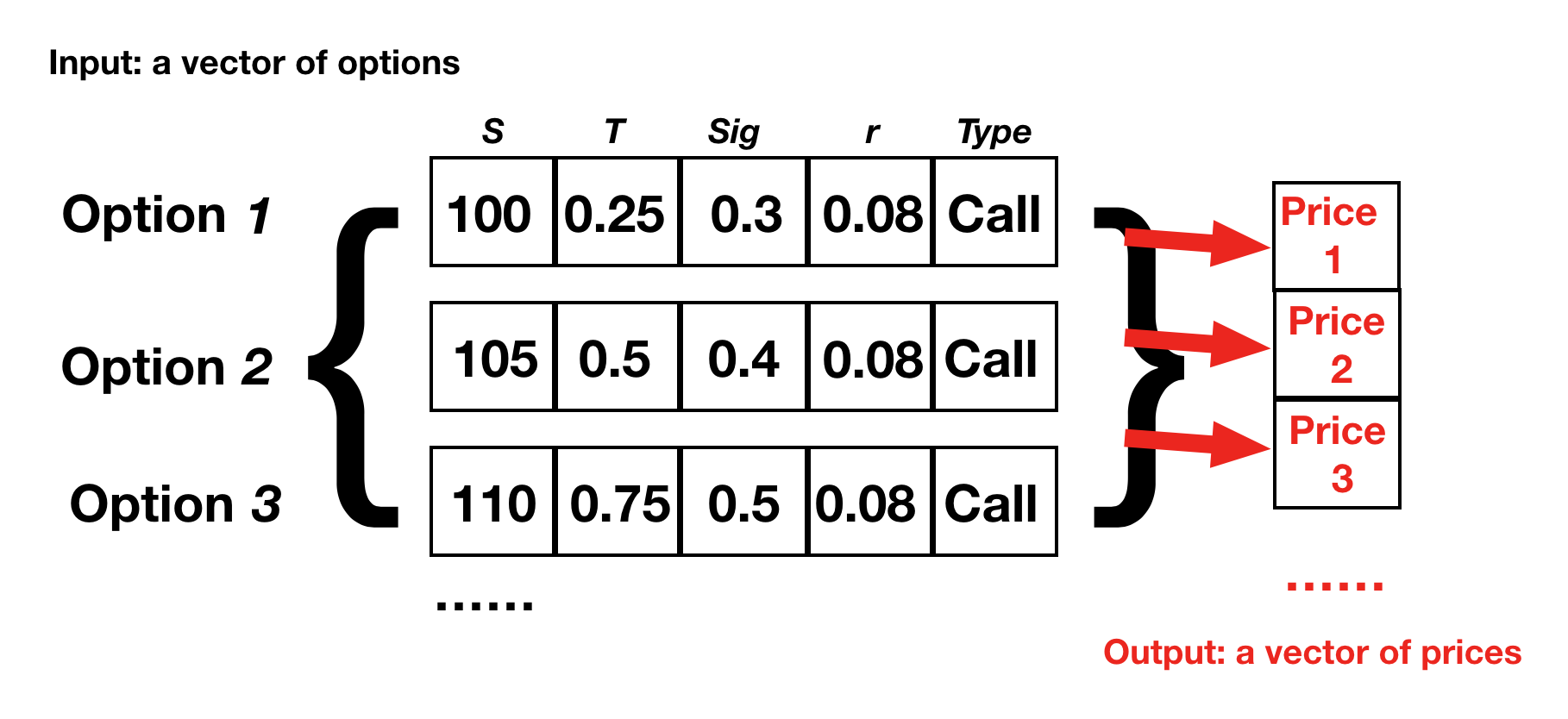


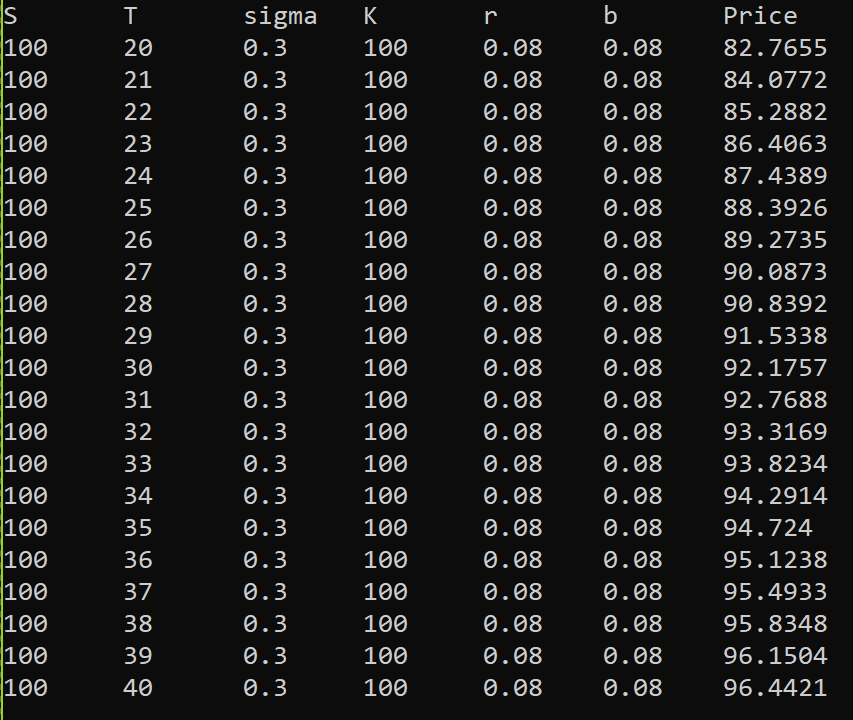
Batch 1 to 4 satisfy the put-call parity relationship under default tolerance of 0.000001. The user can modify the tolerance for parity relationship satisfaction to see different results.

c)

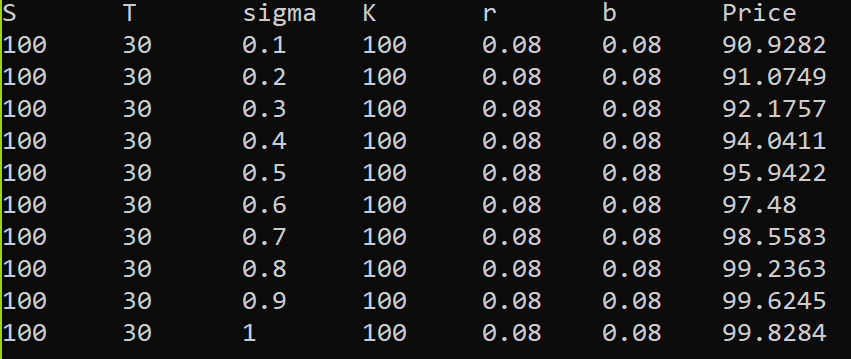
 We would like to compute option prices for a monotonically increasing range of underlying values of S. For demonstration purpose, we try to price Batch 4 as a call option with the mesh array of underlying prices from 10 to 50. We first use the global function MeshArray to create a vector of doubles separated by 1, and then pass the vector as an argument to the overloaded Price() function of European Option. Using the global PrintPriceMesh function, we get a nicely formatted column of underlying prices and the computed call option prices.

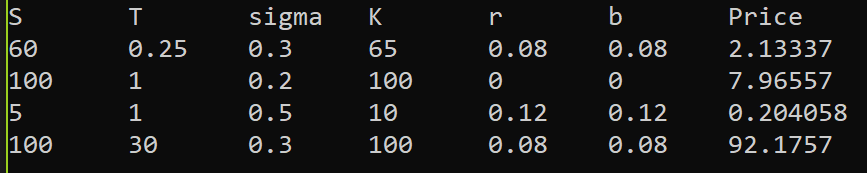
d)



 We create a mesh for Expiry time T from 20 to 40, and use the EuropeanMatrix function to construct a matrix of Batch 4 call options with different expiry time. The format of the matrix is illustrated by the figure above.

The matrix is then passed to the overloaded Price function for EuropeanOption class for a vector of prices for the options in the matrix.

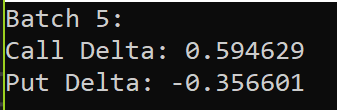
 Similarly, we create a matrix of Batch 4 call options with different volatility and price them.



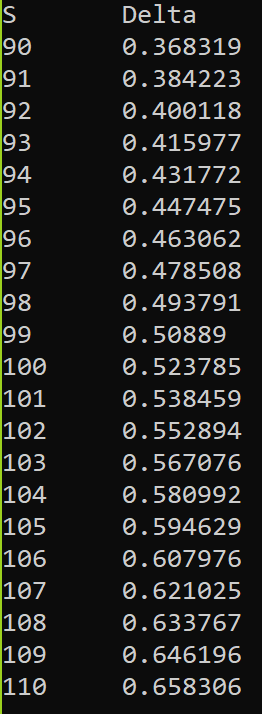
More generally, we can price a matrix of different options. The above result comes from pricing a matrix of call options consisting of option parameters of Batch 1 to 4.

**Option Sensitivities, aka the Greeks**

a)

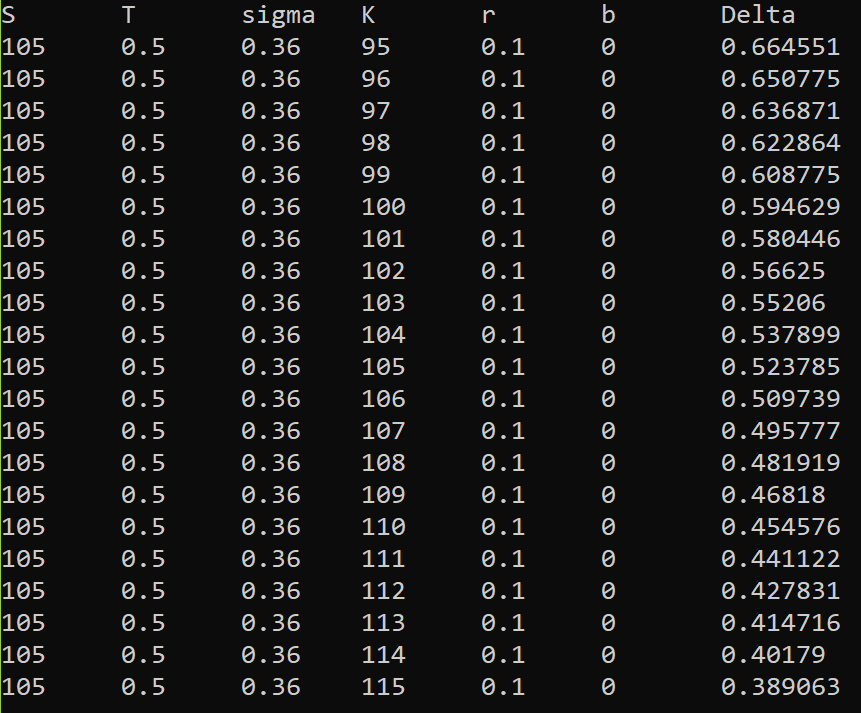


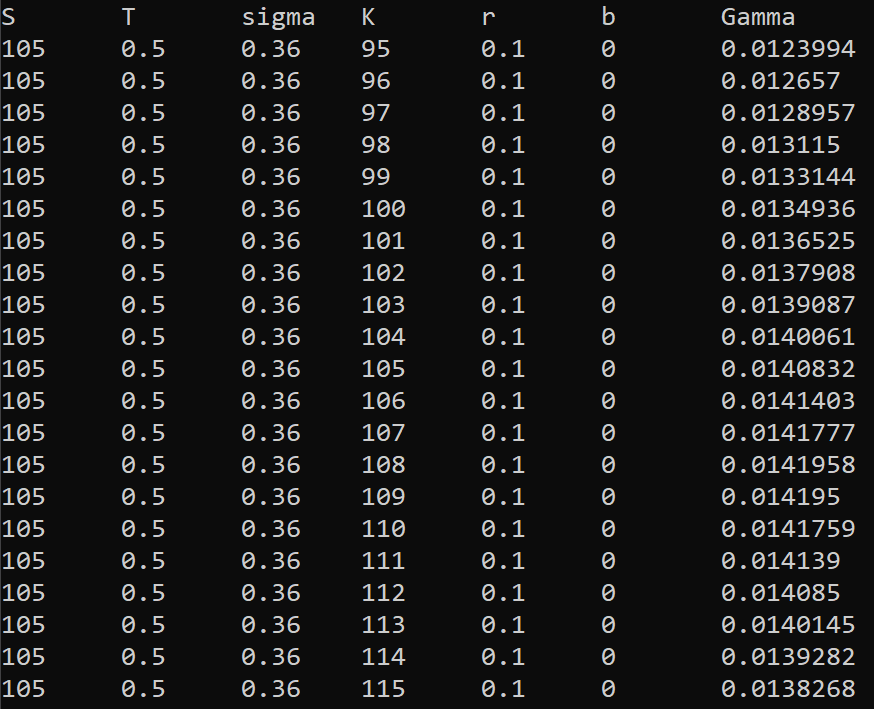
b)

In this problem, we are asked to compute call delta price for a monotonically increasing range of underlying values of S.

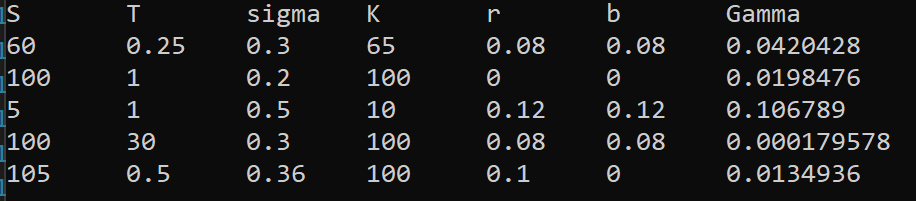
We use the same option from part (a). We create a mesh for underlying values of S from 90 to 110. We pass the mesh to overloaded Delta function for EuropeanOption class, and print the result.

c)

 For illustration, we created a mesh for Strike Price K from 95 to 115. We create a matrix based on Batch5 call option with different K. We pass the matrix to overloaded Delta function. We print the resulting Delta values and matrix.

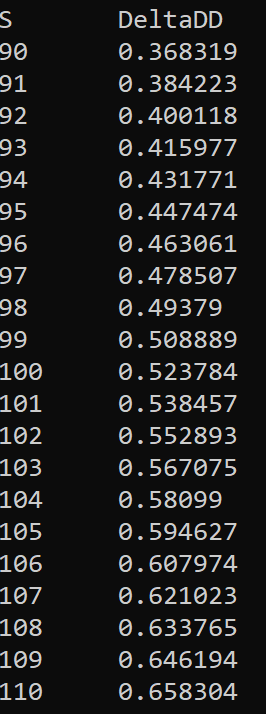
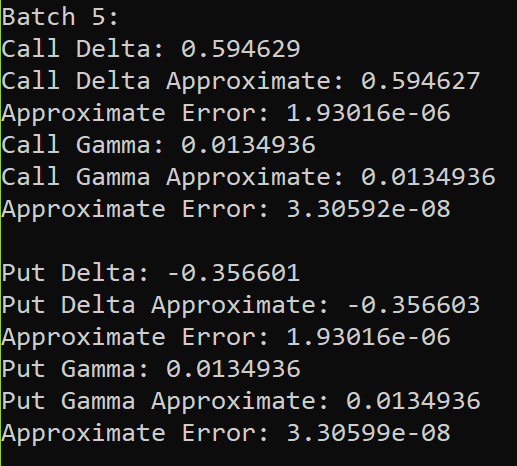


Similarly, we pass the matrix to overloaded Gamma function. We print the resulting Gamma values and matrix.

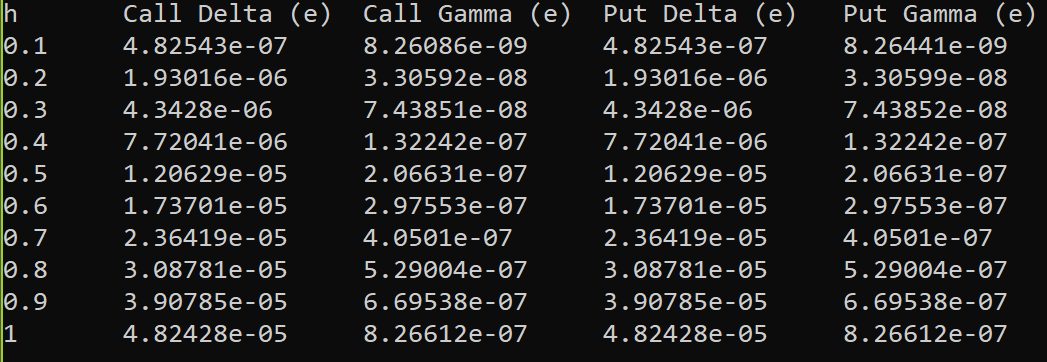


More generally, the user can input a matrix of option parameters and receive a vector of either Delta or Gamma as the result. For the illustration above, we input a parameter matrix for Batch 1 to 5 and obtained their respective Gamma.

d)



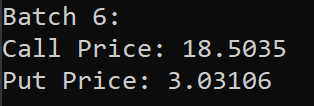
In this section, we first perform similar task to a) and b). We calculate Delta and Gamma for Batch5, but instead of using exact solution, we use the new DeltaDD and GammaDD functions with parameter h = 0.2 for divided difference method. The approximated solution is then subtracted from the exact solution to obtain the error for the approximation. We also use the overloaded DeltaDD function to perform the task of b) for divided difference approximation. The user can also do the same thing for GammaDD.



Next, we perform an error analysis for approximation of Delta and Gamma for Batch5. As shown in the result, the error for all approximations increases as h increases.

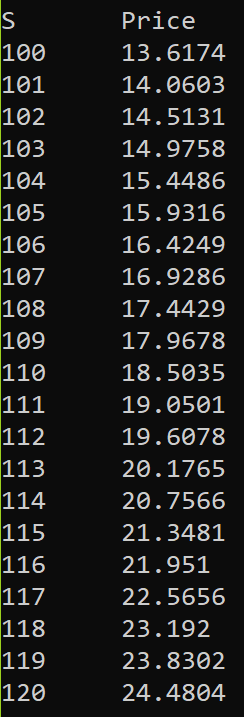
**B. Perpetual American Options**

a) and b)

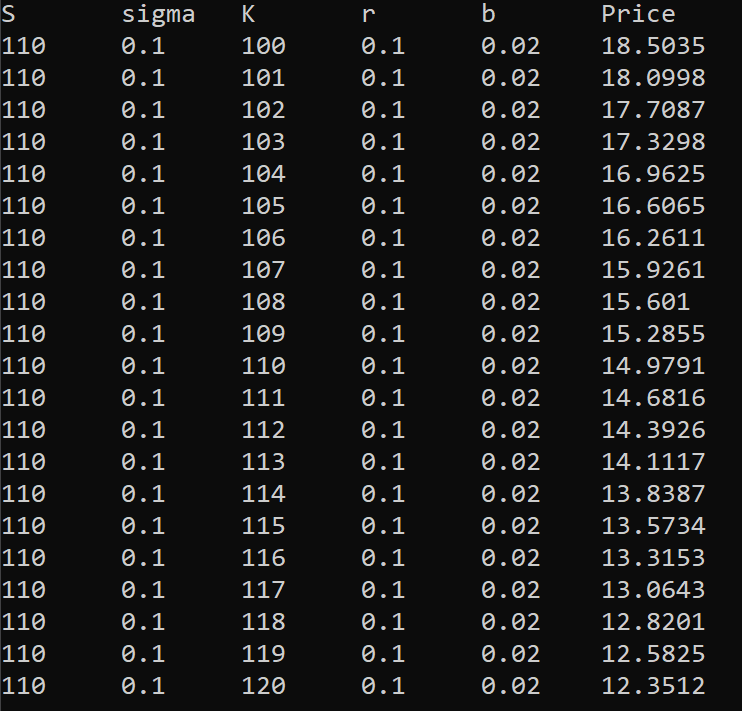


Calculate the call and put prices for Batch6 by calling the Price function for AmericanPerpetual Class.

c)

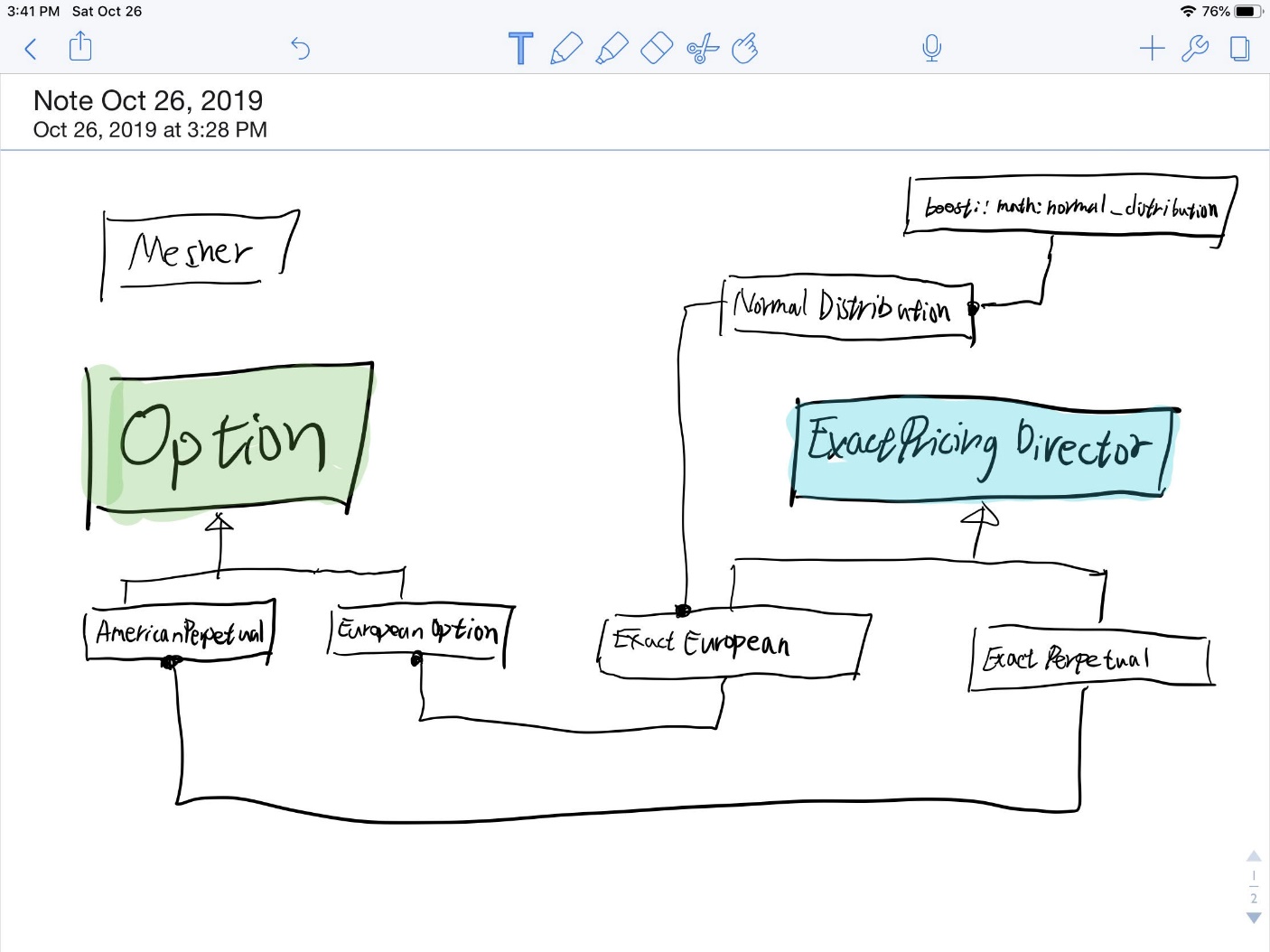
 Create a S mesh from 100 to 120. Price Batch6 call option using overloaded Price function for a vector of S inputs.

d)

 We test the matrix pricer for Perpetual American option with a matrix of Batch6 call option with different strike prices. The user can pass a more generic matrix of parameters for distinct Perpetual American options to the price function of AmericanPerpertual class and receive a vector of option prices using the exact solution method.

**Part Two: Justification for Design**

Overview:



This option pricing program is designed to follow an object-oriented approach and the single responsibility principle, while outsourcing the container and algorithms to Standard Template Library and Boost Library. The option data are stored in collection of option classes derived from a single parent class called Option. Member functions are implemented for each type of option classes to calculate various pricing components for options, but the actual calculations are delegated to another collection of classes derived from a parent class called ExactPricingDirector.

The original goal was to implement polymorphic pricing functions for a single option class with a data member to indicate its option type. However, during the implementation, this approach results in the failure of calculation delegation to the pricing classes, violating the single responsibility principle. As we prefer not to hard code the calculation into the option classes, this approach is abandoned.

The most obvious flaw in the final approach is that many member functions are declared as static, which is far from an ideal implementation. The reason for doing this is to enable flexibility in CalculateArray and CalculateMatrix functions. The last argument in these two functions take in a function pointer, which the two functions then use to calculate components for an array of input prices or a matrix of input parameters. Declaring the member functions as static is only implementation possible in my knowledge to accomplish this task. In the end, the decision was made because the benefit of flexibility in these two functions outweighs the drawbacks of declaring member functions as static. To minimize the dis advantage, the static functions, CalculateArray, and CalculateMatrix are all declared as private functions. Since our task is to price options, and we don’t make any modifications to these member functions and member data during the process, declaring them as static functions won’t cause too much harm.

Classes:

NormalDistribution.hpp:

NormalDistribution class is wrapper class for boost::math::normal\_distribution. The purpose of having NormalDistribution class is primarily to perform the CDF and PDF calculations for Normal (0, 1) distribution in option pricing formulas accurately. For flexibility in the future, NormalDistribution is implemented as a template class, and NormalDistribution objects can be declared for different mean and standard deviations other than 0 and 1.

Mesher.hpp:

This file is modified version of Mesher.hpp provided by Professor Duffy. In this version, we added a MeshArray global function, which generate a vector of doubles separated by equal distance. We also added the EuropeanMatrix and PerpetualMatrix global functions, which generate a matrix of option parameters for the same option, by altering one of the parameters for input option in each row of parameters.

Option.hpp:

Option class is a base class for AmericanPerpetual and EuropeanOption classes. Option class contains 7 essential elements to determine an option's characteristics.

Parameters initialized:

• T (expiry time/maturity). This is a number, e.g. T = 1 means one year. K (strike price).

• sig (volatility).

• r (risk-free interest rate).

• S (current stock price where we wish to price the option).

• C = call option price, P = put option price.

• b = cost of carry

For the default constructor, the order of parameters inputted is according to an instruction thread on QuantNet. The parameters function returns a vector of option parameters according to this order.

EuropeanOption.hpp

This file is modified from EuropeanOption.hpp from DataSim.

In this version, we added the following:

1. Public Price(), Delta(), and Gamma() functions, each of which is implemented to take no

argument, a vector of doubles (underlying prices), and a matrix of option parameters.

2. Private functions CallPrice(double U), PutPrice(double U), CallDelta(double U), PutDelta(double U), CallPutGamma(), and CallPutGamma(double U) are added to perform the different functions as well as to perform specific calculations in Price(), Delta(), and Gamma() functions.

3. A pointer to ExactEuropean Pricing class, to which the actual calculations are delegated.

4. DeltaDD and GammaDD functions to approximate Gamma and Delta of the option using the divided difference method.

5. PriceParity and its helper functions to check Put/Call price parity

Modifications: The EuropeanOption class is now a derived class of Option class

AmericanPerpetual.hpp:

This file contains the function declaration for class AmericanPerpetual, a derived class from Option. The class structure is similar to the EuropeanOption class, except the T variable in the parent Option class is always initialized to -100. The reason is that the T parameter technically does not exist for Perpetual American Options, and this initialization helps us to detect complicated problems in the future. Accordingly, the T parameter is made completely isolated from the public use. No getter or setter functions are implemented, and the parameter function does not return T. The Gamma and Delta functions were also not implemented for this class. The pricing component is delegated to a dynamically allocated ExactPerpetual pricing class.

ExacPricingDirector.hpp:

ExactPricingDirector class is a parent class for all classes that perform exact pricing calculations for options. This class reflects an orginal attempt to imbed a pointer to a pricing director in Option class and store the corresponding different derived classes of pricing directors in the derived classes of Option class. This attempt failed but is still worth exploring.

ExactEuropean.hpp:

ExactEuropean is a derived class from ExactPricingDirector. ExactEuropean class performs the calculations of Price, Gamma, and Delta for European Options using exact pricing formulas. Many of the member functions and member data are declared as static, which is not an optimal implementation; however, this is the only implementation possible so far to pass them into CalculateArray and CalculateMatrix functions as function pointers. The benefit of this level of flexibility is far more significant for the design. The static functions are mostly made private to avoid interference from the user.

ExactPerpetual.hpp:

ExactPerpetual is another derived class from ExactPricingDirector. The general structure is similar to ExactEuropean class. ExactPerpetual class performs the calculations of price of Perpetual American Options using exact pricing formulas. CalculateArray function takes only 5 input arguments, different from CalculateArray function in ExactEuropean class, as Perpetual American Options don't have expiry time T.