**Documentation**

**Design justifications:**

I use one base option class as a parent class for those commonly used price calculating functions, since all types of options need them. I set all price calculating functions pure virtual.

Also, I create a function to store the array matrix input into a vector of vector, and a function to add more options at run time (to demonstrate the flexibility of using a vector over an array).

The European option is designed as a derived class. In this class:

1. Price, Delta, Gamma for call and put calculations are private functions.
2. I use one Price, Delta, and Gamma public functions to determine whether it is a call or a put, and return the result calculated by the private functions.

**Data sets:**

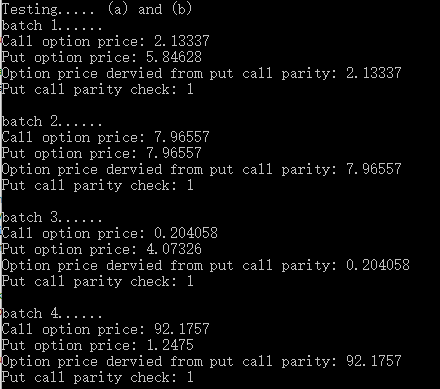
Batch 1: T = 0.25, K = 65, sig = 0.30, r = 0.08, S = 60 (then C = 2.13337, P = 5.84628).

Batch 2: T = 1.0, K = 100, sig = 0.2, r = 0.0, S = 100 (then C = 7.96557, P = 7.96557).

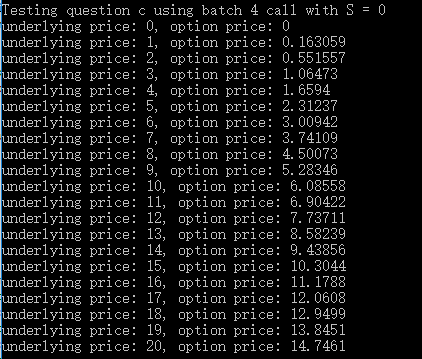
Batch 3: T = 1.0, K = 10, sig = 0.50, r = 0.12, S = 5 (C = 0.204058, P = 4.07326).

Batch 4: T = 30.0, K = 100.0, sig = 0.30, r = 0.08, S = 100.0 (C = 92.17570, P = 1.24750).

1. **Questions:**
2. **a)** 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.
3. **b)** 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.
4. **Answer:** For batch 1 and 2 I use pointer to initialize and create an object on the heap. For batch 3 and 4 I use regular initialization and create an object on the stack.



**c)** 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<double> object. It will be useful to write a global function that produces a mesh array of doubles separated by a mesh size h.

1. **Answer:** Set a starting price and the range to which the price goes up. Then I create a vector to store the parameters and calculate the price. I test batch 4 call with S starting from 0, the result shown as follows:
2. 
3. **d)** 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.
4. **Answer:**

I find it the best to input an array of array first, since an array initialization allows a programmer to add or delete corresponding data (rows and columns) in a visually direct way.

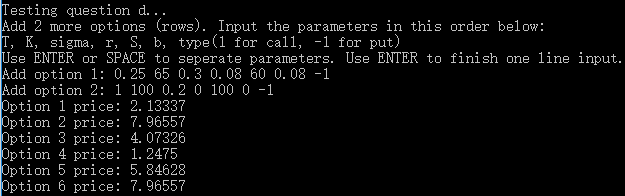
Another advantage of using an array to input is that there is no limit on the size of the array. The programmer can input as many rows as he/she wants.

Also, the number of parameters can be easily adjusted.

Then I store the array of array into a vector of vector (a function in my base option class), so that a programmer can further modify the initial data, e.g., modify, add, or delete some elements or rows at run time.

Moreover, I create a function (in my base option class) to add more rows at run time to demonstrate.

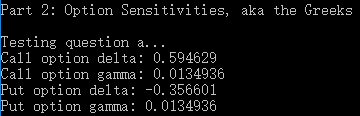
Finally, I calculate the prices of using the option parameters in the vector of vector and store them into a vector to display.



**Option Sensitivities, aka the Greeks**

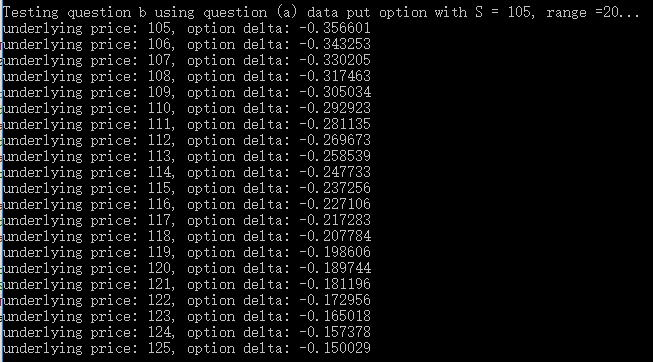
a) Implement the above formulae for gamma for call and put future option pricing using the data set: K = 100, S = 105, T = 0.5, r = 0.1, b = 0 and sig = 0.36. (exact delta call = 0.5946, delta put = -0.3566).

**Answer:**



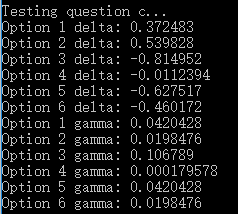
b) We now use the code in part a to compute call delta price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and it entails calling the above formula for a call delta for each value S and each computed option price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

**Answer:** Similar to question (c) in part 1:



c) 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.

**Answer:** Similar to question (d) in part 1 (right now the matrix has 6 options):



d) 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.

The objective of this part is to perform the same calculations as in parts a and b, but now using divided differences. Compare the accuracy with various values of the parameter h (In general, smaller values of h produce better approximations but we need to avoid round-offer errors and subtraction of quantities that are very close to each other). Incorporate this into your well-designed class structure.

**Answer:** Similar to question (a) only in this case we adjust the S input.

To sum up, approximated results come closest to exact solutions when h = 0.000005. If h goes smaller than 0.000005, the results start to diverge.

