Results:
A. Exact Solutions of One-Factor Plain Options

Answer the following questions:
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.

```
A1. (a)*******************************************

Batch 1 Call Price: 2.133368445   Put Price: 5.84628221
Batch 2 Call Price: 7.965567455   Put Price: 7.965567455
Batch 3 Call Price: 0.2040578815   Put Price: 4.073262249
Batch 4 Call Price: 92.17570384   Put Price: 1.247499171
```

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.

```
A1. (b)*******************************************

Batch 1 Call Price from parity: 2.133368445   Put Price from parity: 5.84628221
Batch 2 Call Price from parity: 7.965567455   Put Price from parity: 7.965567455
Batch 3 Call Price from parity: 0.2040578815   Put Price from Parity: 4.073262249
Batch 4 Call Price from parity: 92.17570384   Put Price from Parity: 1.247499171
If Batch 1 follows Put-Call Parity: Yes  Yes
If Batch 2 follows Put-Call Parity: Yes  Yes
If Batch 3 follows Put-Call Parity: Yes  Yes
If Batch 4 follows Put-Call Parity: Yes  Yes
```

By using two methods, we can see that the put/call parity satisfied.

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.

An sample output of S= 60,61,62,63,64,65.

```
A1. (c)*******************************************

The Underlying asset value are: 60, 61, 62, 63, 64, 65,
The corresponding price of Option are: 2.133368445, 2.526987551, 2.963167342, 3.441960011, 3.962931205, 4.525197069,
```

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.

```
A1. (d)*******************************************

The matrix of the input is:
S    K    r     T    sig    b  type
0.25, 65, 0.3, 0.08, 0.08, 60, 1,

0.25, 65, 0.3, 0.08, 0.08, 60, 0,

1, 100, 0.2, 0, 0, 100, 1,

1, 100, 0.2, 0, 0, 100, 0,

The price of Option with such an parameter matrix are: 2.133368445, 5.84628221, 7.965567455, 7.965567455,
```

Option Sensitivities, aka the Greeks

Answer the following questions:

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).

```
A2. (a)*******************************************

The delta of the call option is: 0.5946286597   of the put option is: -0.3566007648
The gamma of the call option is: 0.01349363711   of the put option is: 0.01349363711
```

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.

An sample output of 100,101,102,103,104,105

```
A2. (b)*******************************************

The Underlying asset value are: 100, 101, 102, 103, 104, 105,
The corresponding delta of Option are: 0.5237852528, 0.5384588716, 0.5528942892, 0.5670764954, 0.5809919707, 0.5946286597,
```

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.

```
A2. (c)*******************************************

The matrix of the input is:
S    K    r     T    sig   b   type
0.5, 100, 0.36, 0.1, 0, 105, 1,

0.5, 100, 0.36, 0.1, 0, 105, 0,

The Delta of Option with such an parameter matrix are: 0.5946286597, -0.3566007648,
The Gamma of Option with such an parameter matrix are: 0.01349363711, 0.01349363711,
```

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, for example:

In this case the parameter h is 'small' in some sense. By Taylor's expansion you can show that the above approximations are second order accurate in h to the corresponding derivatives.
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.

```
A2. (d)*******************************************

The delta of the call option is: 0.5946286597 of the put option is: -0.3566007648
let's apply dividend difference method with different h.
when h=1, the delta of call option: 0.5945804169   of the put option: -0.3566490076
when h=0.1, the delta of call option: 0.5946281772   of the put option: -0.3566012473
when h=0.01, the delta of call option: 0.5946286549   of the put option: -0.3566007696
when h=0.001, the delta of call option: 0.5946286597   of the put option: -0.3566007648


The Gamma of the call option is: 0.01349363711 of the put option is: 0.01349363711
let's apply dividend difference method with different h.
when h=1, the Gamma of call option: 0.0134928105   of the put option: 0.0134928105
when h=0.1, the Gamma of call option: 0.01349362885   of the put option: 0.01349362885
when h=0.01, the Gamma of call option: 0.01349363686   of the put option: 0.01349363707
when h=0.001, the Gamma of call option: 0.01349363288   of the put option: 0.01349363998
```

B. Perpetual American Options

a) Program the above formulae, and incorporate into your well-designed options pricing classes.
b) Test the data with K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110 (check C = 18.5035, P = 3.03106).

```
B1. (a) (b)******************************************

The price of the Perpetual American call option is: 18.50349988   of the put option is: 3.031060383
```

c) 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<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.
an sample output of 105,106,107=108,109,110

```
B1. (c)******************************************

The Underlying asset value are: 105, 106, 107, 108, 109, 110,
The corresponding price of Option are: 15.93161414, 16.42489999, 16.92861155, 17.44286864, 17.96779131, 18.50349988,
```

d) 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.

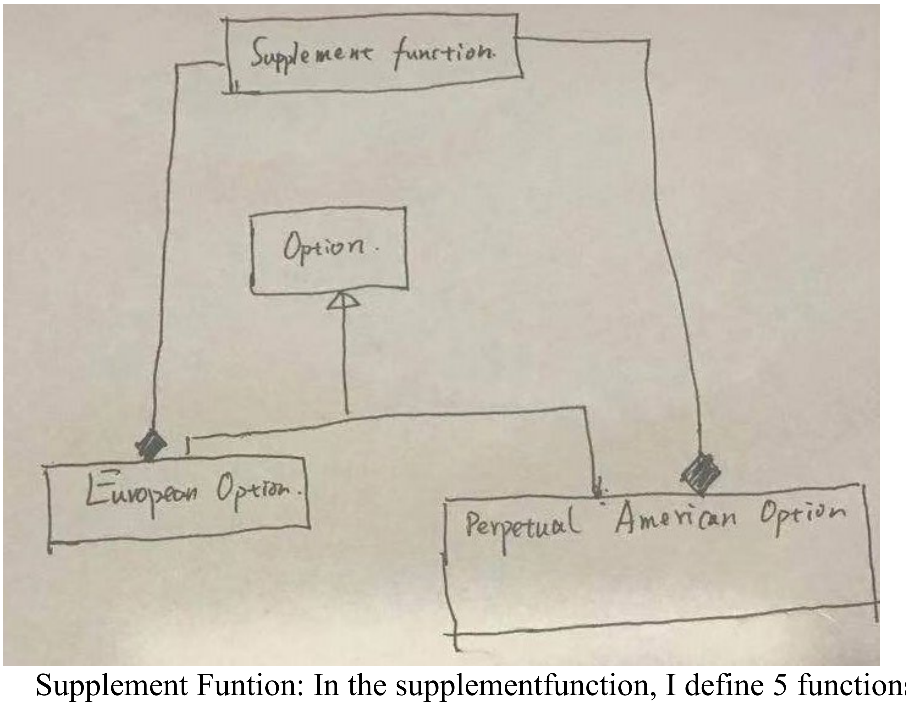
```
B. (d)******************************************

The matrix of the input is:
S    K    r     T    sig   b  type
100, 0.1, 0.1, 0.02, 110, 1,

100, 0.1, 0.1, 0.02, 110, 0,

The price of Option with such an parameter matrix are: 18.50349988, 3.031060383,
```

Explanation:



Supplement Funtion: In the supplementfunction, I define 5 functions that will be extensively used in the project. N(double x) returns the cdf of Normal distribution. n(double x) returns the pdf of normal distribution. print_vector helps to output the vector. print_matrix helps to output the matrix. And Msher() produces the MeshArray of double that are separated by a mesh size h.

Option: The Option class is the base class for Europran Option and Perprtual American Option, which contains member m_S,m_K,m_T,m_r,m_sig,m_b, m_type (C= call option, P=put option). It also contains the respective getter and setter function for these members.

EuropeanOption: European Option is a derived class from Option class. It contains the methods to calculate the Price, Gamma, Delta by exact method check the put-call parity and use the devided difference to approximate. I also enable it to deal with input parameter matrix and return the result as a vector.

PerprtualAmericanOption: PerprtualAmericanOption is a derived class from Option class. Different from the European Option, it does not have a expiry date so we set the T to a default value. It contains the methods to calculate the Price using exactsolution. I also enable it to deal with input parameter matrix and return the result as a vetcor

I could have set up a MatrixPricer to better encapsulate the functionality of receiving an input parameter matrix. But due to the time limitation, I could not do that, maybe I can show that to you on our final exam, if you wishes.