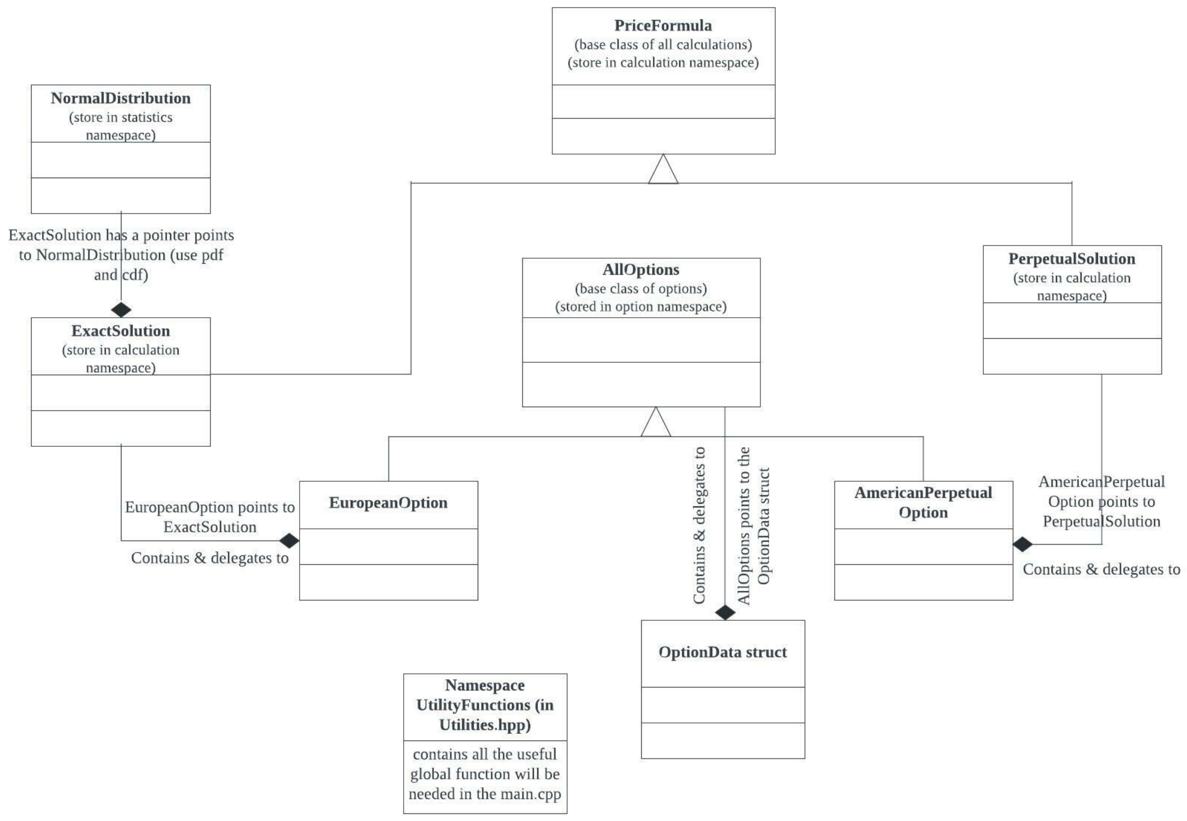
Class Design Structure


PriceFormula
(base class of all calculations)
(store in calculation namespace)
NormalDistribution
(store in statistics namespace)
ExactSolution has a pointer points to NormalDistribution (use pdf and cdf)
ExactSolution
(store in calculation namespace)
AllOptions
(base class of options)
(stored in option namespace)
PerpetualSolution
(store in calculation namespace)
EuropeanOption points to ExactSolution
Contains & delegates to
EuropeanOption
Contains & delegates to
AllOptions points to the OptionData struct
AmericanPerpetual Option
AmericanPerpetual Option points to PerpetualSolution
Contains & delegates to
OptionData struct
Namespace UtilityFunctions (in Utilities.hpp)
contains all the useful global function will be needed in the main.cpp

Please refer to the above image as my class hierarchy and structure.

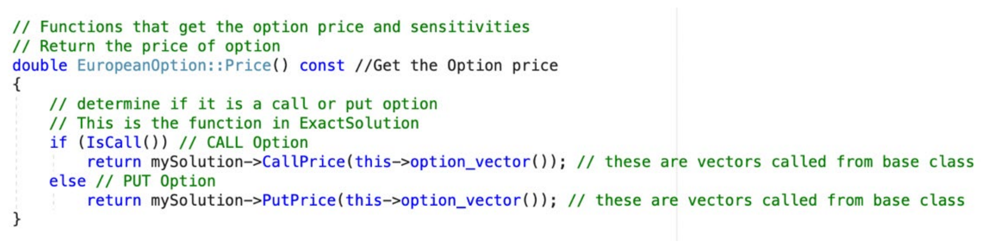
Abstract Base PriceFormula Class (have derived class of ExactSolution and PerpetualSolution)
There are different types of calculating the price, also there are different types of options that could be calculated. Based on this, I created this base class. As for the future, more calculations might be added under this class.

- **ExactSolution Class** (derived class of PriceFormula)

This is a heavy calculation class that is right now used to calculate all of the needed Option price (both put and call), Option delta (both put and call sensitivity), Gamma, Vega, and Theta for European Option.

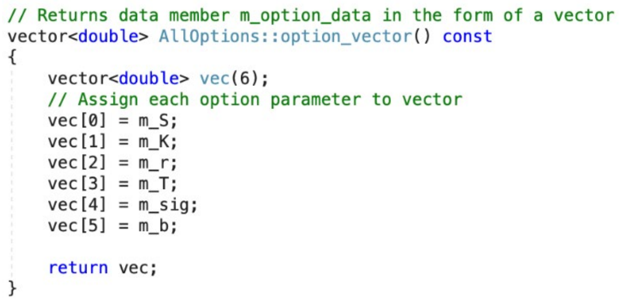
```
// Put price calculation, passing the vector
double PutPrice(const vector<double>& OP) const;
```

I directly pass in the vector of all the parameters (K, S, T, r, sig, b).
In the EuropeanOption.cpp, I use a pointer to call the value of the price by passing the vector of these parameters already defined in my base class AllOptions.



```
// Functions that get the option price and sensitivities
// Return the price of option
double EuropeanOption::Price() const //Get the Option price
{
    // determine if it is a call or put option
    // This is the function in ExactSolution
    if (IsCall()) // CALL Option
        return mySolution->CallPrice(this->option_vector()); // these are vectors called from base class
    else // PUT Option
        return mySolution->PutPrice(this->option_vector()); // these are vectors called from base class
}
```

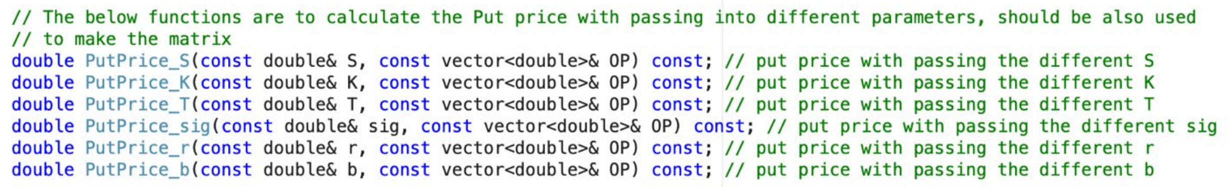
(image from EuropeanOption.cpp)



```
// Returns data member m_option_data in the form of a vector
vector<double> AllOptions::option_vector() const
{
    vector<double> vec(6);
    // Assign each option parameter to vector
    vec[0] = m_S;
    vec[1] = m_K;
    vec[2] = m_r;
    vec[3] = m_T;
    vec[4] = m_sig;
    vec[5] = m_b;

    return vec;
}
```

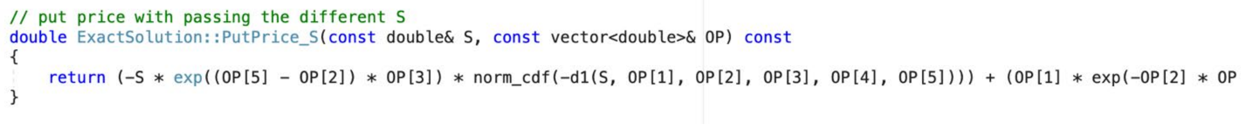
(image from AllOptions.cpp)

As required in part (c)&(d), I created a couple of different functions that could receive different parameters each time, so we could change a parameter (K, S, T…) at a time to get the corresponding price.



```
// The below functions are to calculate the Put price with passing into different parameters, should be also used
// to make the matrix
double PutPrice_S(const double& S, const vector<double>& OP) const; // put price with passing the different S
double PutPrice_K(const double& K, const vector<double>& OP) const; // put price with passing the different K
double PutPrice_T(const double& T, const vector<double>& OP) const; // put price with passing the different T
double PutPrice_sig(const double& sig, const vector<double>& OP) const; // put price with passing the different sig
double PutPrice_r(const double& r, const vector<double>& OP) const; // put price with passing the different r
double PutPrice_b(const double& b, const vector<double>& OP) const; // put price with passing the different b
```

(how they are declared in my ExactSolution.hpp)



```
// put price with passing the different S
double ExactSolution::PutPrice_S(const double& S, const vector<double>& OP) const
{
    return (-S * exp((OP[5] - OP[2]) * OP[3]) * norm_cdf(-d1(S, OP[1], OP[2], OP[3], OP[4], OP[5]))) + (OP[1] * exp(-OP[2] * OP
}
```

(example of how they are implemented in the .cpp)

- **PerpetualSolution Class** (derived class of PriceFormula): basically the same logic as the ExactSolution class, this will do the calculation for the American Perpetual Option (calculations are totally different for European Option and American Perpetual Option).

OptionData struct
The public member contains: S, K, T, r, sig, b, type, model. And I also got a parameterized constructor, so later in the main, we could just use this parameterized to pass the data to this OptionData struct.
The reason of making it a struct: I did this OptionData struct because I think this could potentially enhance future reusability as there might be more option classes added.

Let's now move to the Options Class to see how the calculations are called!

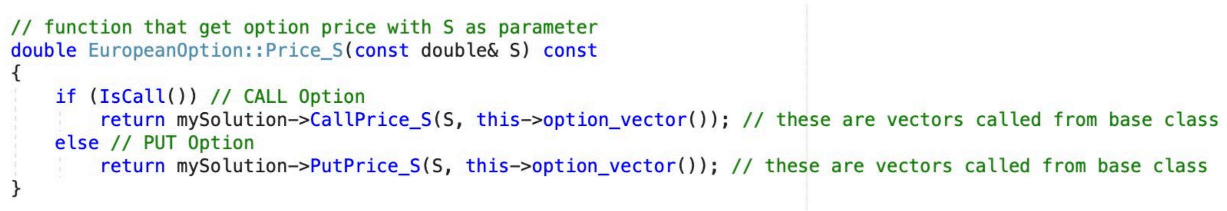
Abstract base class: AllOptions (have derived class of EuropeanOption and AmericanPerpetualOption)
I create a pointer point to the OptionData struct in order to have the access to the data. Pass-by-pointer allows me to use virtual function, and might be more efficient, which I will elaborate on more in this write-up in later section.
Because there are two options in our project: European plain and American Perpetual, I decided to make a parent class that stored the member data so the two derived class could inherit those data. Although the calculations of American Perpetual Option are quite different from those of European Option, common data members are shared. Also, I declare some pure virtual functions inside the base class that will be used in my derived class. The prototype will be the same, and I might just override it, so make much more sense and easier. (such as Price(), vector<vector<double>> Price_Matrix() ...)
For future reusability, more options might be attributed to this base class.

- **EuropeanOption Class** (derived class of AllOptions)

One of the private member data is a pointer that points to the ExactSolution class. In this case, I am able to use the calculation function in the ExactSolution class by passing the data member (K, S, T …) from the protected data member of my base class (either by passing parameters or passing in vectors). I chose to do pass-by-pointer because I think this is more efficient than pass-by-value as the address and the size of a pointer is small, as it will store on the heap, and I could just delete it after the assignment is done. Also, no calculations will be done inside EuropeanOption class, so it's pretty clean inside this class, and I can easily refer to it.

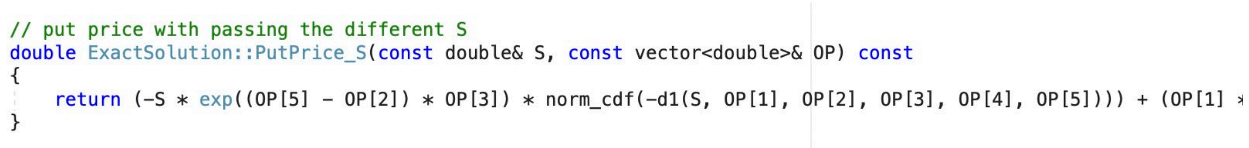
For example, as I need to use the function in the ExactSolution.cpp that could allow to change one parameter at a time, I created the functions that require one parameter in EuropeanOption.cpp, and in the implementation, it points to the ExactSolution to pass that parameter in.

This is how it works:



```
// function that get option price with S as parameter
double EuropeanOption::Price_S(const double& S) const
{
    if (IsCall()) // CALL Option
        return mySolution->CallPrice_S(S, this->option_vector()); // these are vectors called from base class
    else // PUT Option
        return mySolution->PutPrice_S(S, this->option_vector()); // these are vectors called from base class
}
```

(in EuropeanOption.cpp)

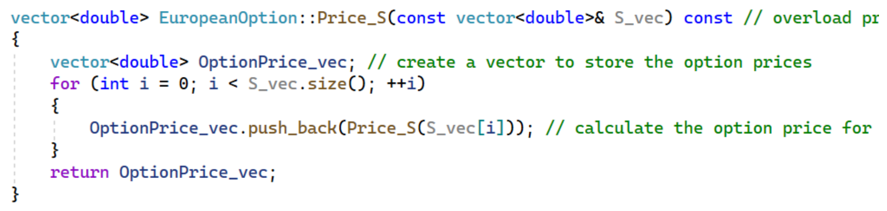


```
// put price with passing the different S
double ExactSolution::PutPrice_S(const double& S, const vector<double>& OP) const
{
    return (-S * exp((OP[5] - OP[2]) * OP[3]) * norm_cdf(-d1(S, OP[1], OP[2], OP[3], OP[4], OP[5]))) + (OP[1] *
}
```

(in ExactSolution.cpp)

In order to calculate and store the parameter in the vectors **((c) of the requirement)**, I also created an overloaded vector<double> Price_S function that has underlying asset price vector as parameter , and it stores the option prices corresponding to each element in stock price vector,

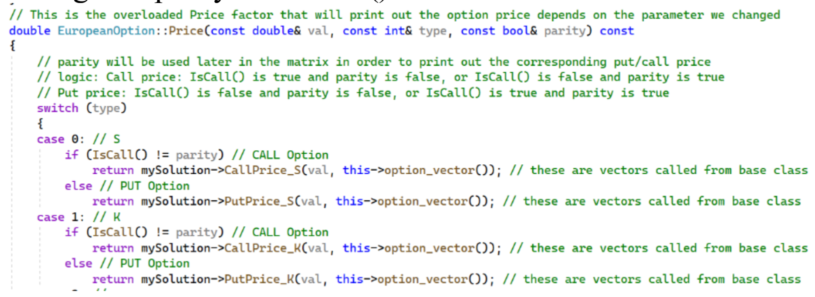
and then calculates the option price for the given stock price, and push it into the option price vector.
Below is the coding detail:



```
vector<double> EuropeanOption::Price_S(const vector<double>& S_vec) const // overload pr
{
    vector<double> OptionPrice_vec; // create a vector to store the option prices
    for (int i = 0; i < S_vec.size(); ++i)
    {
        OptionPrice_vec.push_back(Price_S(S_vec[i])); // calculate the option price for
    }
    return OptionPrice_vec;
}
```

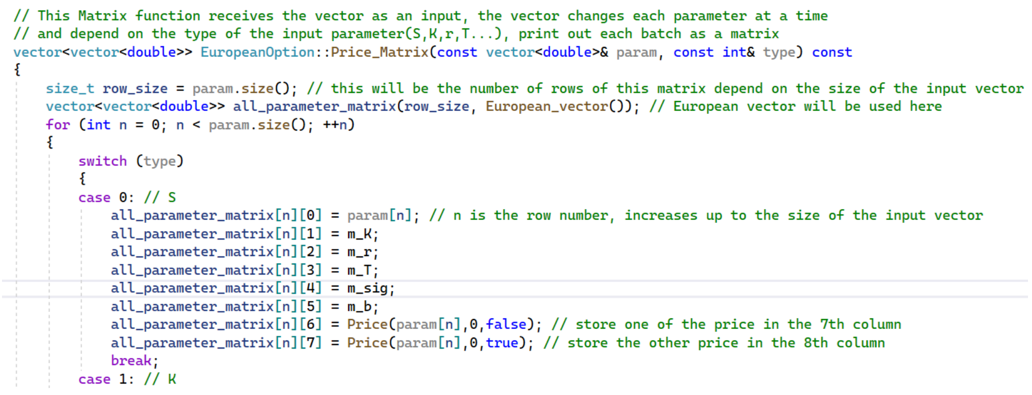
The logic: for each number stored in this S_vec (which will be passed in later by user in the main), the Price_S will calculate each result price and push it back to the OptionPrice_vec vector, finally it will return that vector. (In order to access all those resulting prices, we will utilize a global Print function to print each element out, and this will be shown later.)

As to fulfill the requirement **of (d),** I would need to create a function that will receive the parameters return a matrix of the resulting prices. In this case, I utilize vector of vector.
In order to achieve this, I basicially did it in two ways.
And in order to ensure the most flexibility, as it could print out both put and call price, I wrote a new overloaded Price function inside EuropeanOption class (also defined as pure virtual in the base class, because will be used in American Option too). And this could actually replace all the Price_S, Price_K functions for simplicity (but I kept them there just in case).
The logic of parity and IsCall() is written in the comment



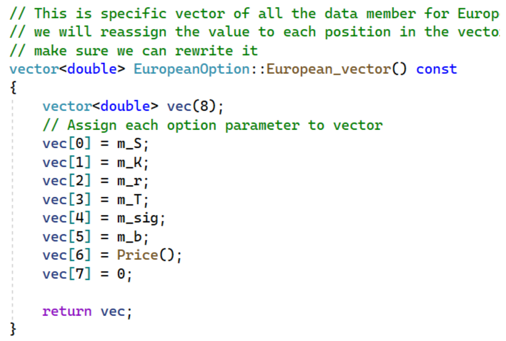
```
// This is the overloaded Price factor that will print out the option price depends on the parameter we changed
double EuropeanOption::Price(const double& val, const int& type, const bool& parity) const
{
    // parity will be used later in the matrix in order to print out the corresponding put/call price
    // logic: Call price: IsCall() is true and parity is false, or IsCall() is false and parity is true
    // Put price: IsCall() is false and parity is false, or IsCall() is true and parity is true
    switch (type)
    {
    case 0: // S
        if (IsCall() != parity) // CALL Option
            return mySolution->CallPrice_S(val, this->option_vector()); // these are vectors called from base class
        else // PUT Option
            return mySolution->PutPrice_S(val, this->option_vector()); // these are vectors called from base class
    case 1: // K
        if (IsCall() != parity) // CALL Option
            return mySolution->CallPrice_K(val, this->option_vector()); // these are vectors called from base class
        else // PUT Option
            return mySolution->PutPrice_K(val, this->option_vector()); // these are vectors called from base class
```

__**The first way**__ of this matrix is to __accept the vector__ (as the one we use it in part c), and the int type, return the matrix as the output. (Because I will utilize it in the American Option too, I decided to make it a pure virtual in the base class)



```
// This Matrix function receives the vector as an input, the vector changes each parameter at a time
// and depend on the type of the input parameter(S,K,r,T...), print out each batch as a matrix
vector<vector<double>> EuropeanOption::Price_Matrix(const vector<double>& param, const int& type) const
{
    size_t row_size = param.size(); // this will be the number of rows of this matrix depend on the size of the input vector
    vector<vector<double>> all_parameter_matrix(row_size, European_vector()); // European vector will be used here
    for (int n = 0; n < param.size(); ++n)
    {
        switch (type)
        {
        case 0: // S
            all_parameter_matrix[n][0] = param[n]; // n is the row number, increases up to the size of the input vector
            all_parameter_matrix[n][1] = m_K;
            all_parameter_matrix[n][2] = m_r;
            all_parameter_matrix[n][3] = m_T;
            all_parameter_matrix[n][4] = m_sig;
            all_parameter_matrix[n][5] = m_b;
            all_parameter_matrix[n][6] = Price(param[n],0,false); // store one of the price in the 7th column
            all_parameter_matrix[n][7] = Price(param[n],0,true); // store the other price in the 8th column
            break;
        case 1: // K
```

European_vector() shown below is the function that stores vector of all the parameters, and the all-parameter-matrix would utilize the size of it but re-write values. (I made this vector just to ensure that I initialize the values, and could reassign the value to it)

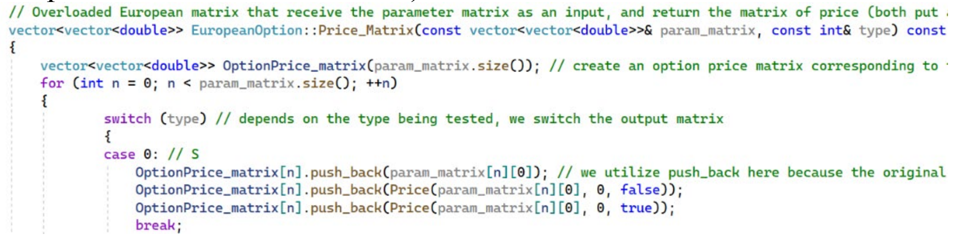


```
// This is specific vector of all the data member for Europ
// we will reassign the value to each position in the vecto
// make sure we can rewrite it
vector<double> EuropeanOption::European_vector() const
{
    vector<double> vec(8);
    // Assign each option parameter to vector
    vec[0] = m_S;
    vec[1] = m_K;
    vec[2] = m_r;
    vec[3] = m_T;
    vec[4] = m_sig;
    vec[5] = m_b;
    vec[6] = Price();
    vec[7] = 0;

    return vec;
}
```

The type (S=0, K=1, r=2, T=3, sig=4, b=5) will be later specified by the client in the main test program. In this case, only the specified parameters get changed while the others stay the same (they will be stored from column 1-6). The 7th and 8th column will calculate the price based on the parameter we changed and store the corresponding price there.
(it utilized the overloaded Price() function as shown below)

As n increases, we will actually move from one row to the next.
(the result will be shown below in the problem section)

The second way accept the matrix we just did, and this time it would only print out the changing parameter and the two option prices in a matrix.
This is **the overloaded Price_Matrix() function** that receives the matrix and the int type, return the matrix as the output. (Because I will utilize it in the American Option too, I decided to make it a pure virtual in the base class)



```
// Overloaded European matrix that receive the parameter matrix as an input, and return the matrix of price (both put
vector<vector<double>> EuropeanOption::Price_Matrix(const vector<vector<double>>& param_matrix, const int& type) const
{
    vector<vector<double>> OptionPrice_matrix(param_matrix.size()); // create an option price matrix corresponding to
    for (int n = 0; n < param_matrix.size(); ++n)
    {
            switch (type) // depends on the type being tested, we switch the output matrix
            {
            case 0: // S
                OptionPrice_matrix[n].push_back(param_matrix[n][0]); // we utilize push_back here because the original
                OptionPrice_matrix[n].push_back(Price(param_matrix[n][0], 0, false));
                OptionPrice_matrix[n].push_back(Price(param_matrix[n][0], 0, true));
                break;
```

I utilize push_back here because the original matrix does not have any values
(results will be shown later in the problem section)

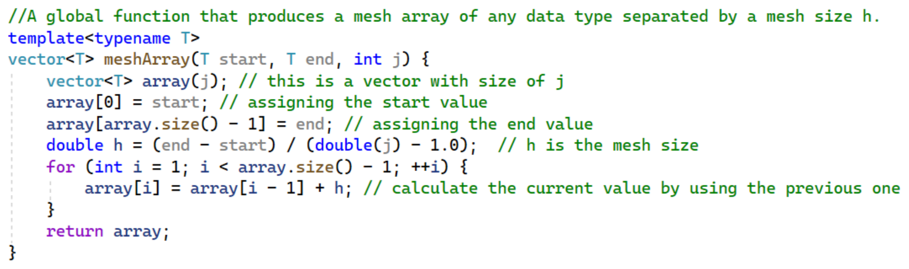
- **AmericanPerpetualOption Class** (derived class of AllOptions)

One of the private member data is a pointer that points to the PerpetualSolution class. Same logics as EuropeanOption (the above functions basically applies here too). But American Option don't need to exercise on the expiry date (can call earlier). We generally can't get an exact solution for the American Optione except this Perpetual one. However, the perpetual price is the time-homogeneous price and is the same as the normal price when the expiry price T tends to infinity. So, we will not have T in our equation.

Virtual Function Usage: because I use the pointer throughout my Options class, I decided to use the virtual function (declared in my base class and overridden in my two derived class), so that it could achieve runtime polymorphism. In this case it would really support the object-oriented programming. Also, it provide prototype because the virtual functions of parent class get redefined in my derived class (the common function that will be used in both classes, such as Price(), Price_S(), Price_Matrix()…).

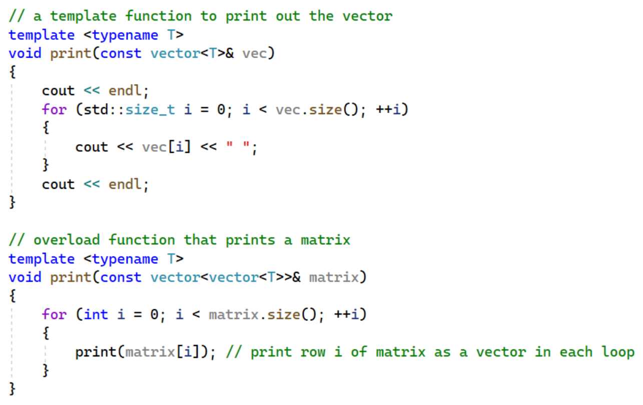
Namespace UtilityFunction
Here I have 3 global functions: 2 overloaded Print(), 1 vector
This namespace contains all the useful global functions. Because the print and vector function are needed in both European Option and Perpetual American Option, and might even more options in the future, it would be optimal, in my perspective, to declare them as global functions. In this case, when the client needs to input the data member in the main test program, those will have relatively easy access.

Also, I make all these global functions **as template function** because there might be different kinds of parameters passing in later on, not limited to double.



```
//A global function that produces a mesh array of any data type separated by a mesh size h.
template<typename T>
vector<T> meshArray(T start, T end, int j) {
    vector<T> array(j); // this is a vector with size of j
    array[0] = start; // assigning the start value
    array[array.size() - 1] = end; // assigning the end value
    double h = (end - start) / (double(j) - 1.0);  // h is the mesh size
    for (int i = 1; i < array.size() - 1; ++i) {
        array[i] = array[i - 1] + h; // calculate the current value by using the previous one
    }
    return array;
}
```

As for the **vector global Function**, I decided to make the user pass in the three parameters so that it could be really flexible: the start value, the end value, and the intended distance. I initialize the start value of the array as the value passed in by user: array[0] = start. Therefore, instead of starting from inputting the number from array[0], I decided we should start from array[1] by utilizing the value of array[0] that is input in by the user. So in the for loop, I set the start point to be i = 1. And in this case, we could utilize the previous value + mesh size to calculate the array[i]. And this function will finally return the array.



```
// a template function to print out the vector
template <typename T>
void print(const vector<T>& vec)
{
    cout << endl;
    for (std::size_t i = 0; i < vec.size(); ++i)
    {
        cout << vec[i] << " ";
    }
    cout << endl;
}

// overload function that prints a matrix
template <typename T>
void print(const vector<vector<T>>& matrix)
{
    for (int i = 0; i < matrix.size(); ++i)
    {
        print(matrix[i]); // print row i of matrix as a vector in each loop
    }
}
```

As for the **print()** global function, I set them as the template function due to the same reason: flexibility. Also, I decided to overload the print() for vector and for matrix so that it would be fairly easy for the client to test in the main program. (one accepts vector<T>, and the other accepts vector<vector<T>>)

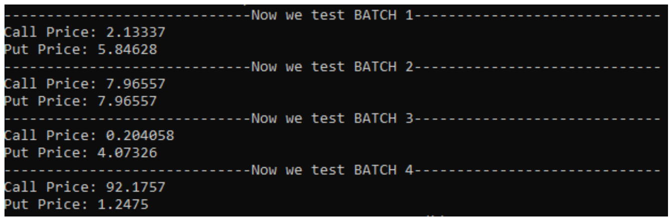
Namespace Usage:
In order to make the class well-organized, I decided to use namespace.
Everything inside this project is under my name's namespace (namespace Christine)
Besides, I have 4 different namespaces inside

1. Options (this namespace contains: AllOptions[the base class], EuropeanOption[derived class], and AmericanPerpetualOption[derived class])
2. Statistics (this namespace contains: NormalDistribution)
3. Calculations (this namespace contains: PriceFormula[the base class], ExactSolution[derived class], PerpetualSolution[derived class])
4. UtilityFunctions(this namespace contains 3 useful global template functions: meshArray(), and two overloaded Print(): one accepts vector<T>, and the other accepts vector<vector<T>>)

Questions:

Group A PART I Exact Solutions of One-Factor Plain Options

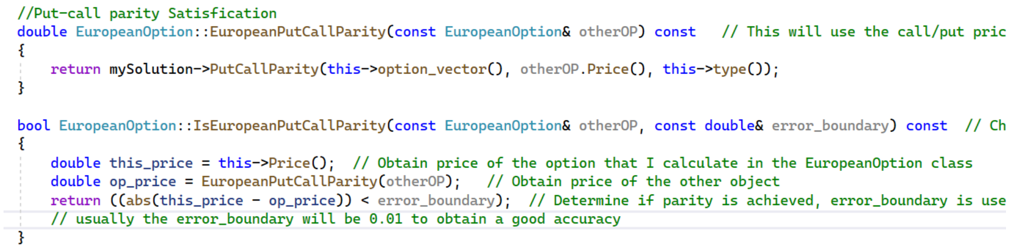
a) Below are the results of each batch



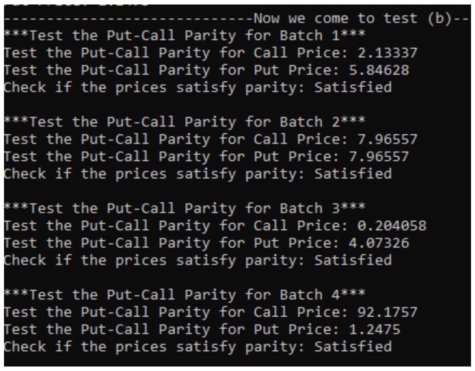
```
----------------------------Now we test BATCH 1----------------------------
Call Price: 2.13337
Put Price: 5.84628
----------------------------Now we test BATCH 2----------------------------
Call Price: 7.96557
Put Price: 7.96557
----------------------------Now we test BATCH 3----------------------------
Call Price: 0.204058
Put Price: 4.07326
----------------------------Now we test BATCH 4----------------------------
Call Price: 92.1757
Put Price: 1.2475
```

b) Below are the result of satisfy parity in 2 ways (one return double, one return bool)

1st way: calculate the corresponding Put/Call price by input Call/Put price
2nd way: a bool function setting up the boundary error (< 0.01) to check if the test is met

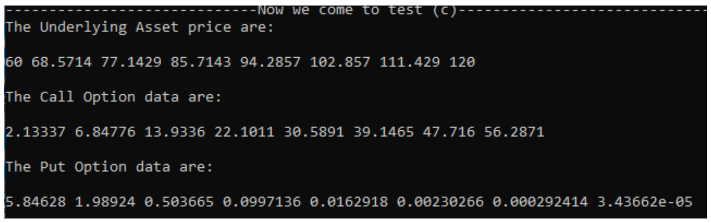


```
//Put-call parity Satisfication
double EuropeanOption::EuropeanPutCallParity(const EuropeanOption& otherOP) const   // This will use the call/put pric
{
    return mySolution->PutCallParity(this->option_vector(), otherOP.Price(), this->type());
}

bool EuropeanOption::IsEuropeanPutCallParity(const EuropeanOption& otherOP, const double& error_boundary) const  // Ch
{
    double this_price = this->Price();  // Obtain price of the option that I calculate in the EuropeanOption class
    double op_price = EuropeanPutCallParity(otherOP);   // Obtain price of the other object
    return ((abs(this_price - op_price)) < error_boundary);  // Determine if parity is achieved, error_boundary is use
    // usually the error_boundary will be 0.01 to obtain a good accuracy
}
```



```
-----------------------------Now we come to test (b)--
***Test the Put-Call Parity for Batch 1***
Test the Put-Call Parity for Call Price: 2.13337
Test the Put-Call Parity for Put Price: 5.84628
Check if the prices satisfy parity: Satisfied

***Test the Put-Call Parity for Batch 2***
Test the Put-Call Parity for Call Price: 7.96557
Test the Put-Call Parity for Put Price: 7.96557
Check if the prices satisfy parity: Satisfied

***Test the Put-Call Parity for Batch 3***
Test the Put-Call Parity for Call Price: 0.204058
Test the Put-Call Parity for Put Price: 4.07326
Check if the prices satisfy parity: Satisfied

***Test the Put-Call Parity for Batch 4***
Test the Put-Call Parity for Call Price: 92.1757
Test the Put-Call Parity for Put Price: 1.2475
Check if the prices satisfy parity: Satisfied
```

c) Below are both the input underlying asset price (range: 60, start value: 60, end value: 120) and the output of call/put price (I use the batch 1 data)

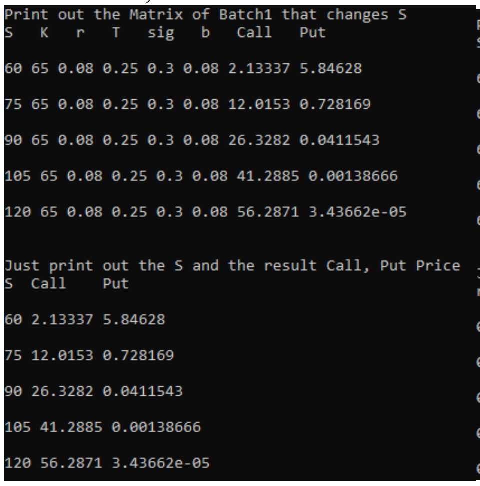


```
-----------------------------Now we come to test (c)-----------------------------
The Underlying Asset price are:

60 68.5714 77.1429 85.7143 94.2857 102.857 111.429 120

The Call Option data are:

2.13337 6.84776 13.9336 22.1011 30.5891 39.1465 47.716 56.2871

The Put Option data are:

5.84628 1.98924 0.503665 0.0997136 0.0162918 0.00230266 0.000292414 3.43662e-05
```

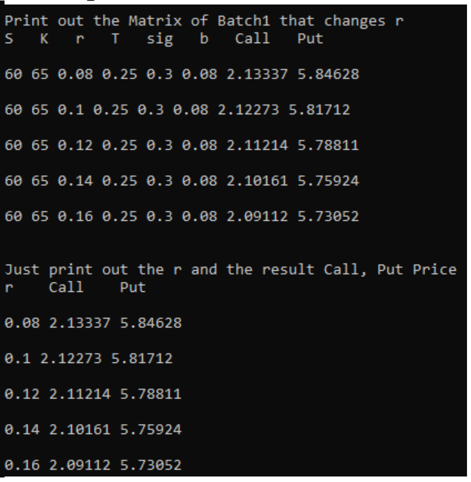
Patterns to notice: as the underlying asset price(S) goes up, the Call Option price goes up, and the Put Option goes down.

As purchasing the call option, we expect the S to go up as much as possible so that we could earn more net profits (we only have to pay the strike price to purchase the stock). Because of this, as the S goes up, the price of Call Option must go up. The same logic applies here with the put, the buyer must expect the asset price to go down, so as it goes up, the price of put option must go down.

d) Below are the matrices for each different input (S, K, T, r, sig, b) (I use the batch 1 data) Notice this only change one parameter at a time, while keep the other one the same (I just print out S and r, but I create the vector for each parameter that could be used to test)



```
Print out the Matrix of Batch1 that changes S
S   K   r   T   sig   b   Call   Put

60 65 0.08 0.25 0.3 0.08 2.13337 5.84628

75 65 0.08 0.25 0.3 0.08 12.0153 0.728169

90 65 0.08 0.25 0.3 0.08 26.3282 0.0411543

105 65 0.08 0.25 0.3 0.08 41.2885 0.00138666

120 65 0.08 0.25 0.3 0.08 56.2871 3.43662e-05

Just print out the S and the result Call, Put Price
S  Call    Put

60 2.13337 5.84628

75 12.0153 0.728169

90 26.3282 0.0411543

105 41.2885 0.00138666

120 56.2871 3.43662e-05
```

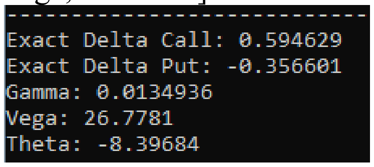


```
Print out the Matrix of Batch1 that changes r
S   K   r   T   sig   b   Call   Put

60 65 0.08 0.25 0.3 0.08 2.13337 5.84628

60 65 0.1 0.25 0.3 0.08 2.12273 5.81712

60 65 0.12 0.25 0.3 0.08 2.11214 5.78811

60 65 0.14 0.25 0.3 0.08 2.10161 5.75924

60 65 0.16 0.25 0.3 0.08 2.09112 5.73052

Just print out the r and the result Call, Put Price
r    Call    Put

0.08 2.13337 5.84628

0.1 2.12273 5.81712

0.12 2.11214 5.78811

0.14 2.10161 5.75924

0.16 2.09112 5.73052
```

When r increases, call price decreases, and the put price also decreases

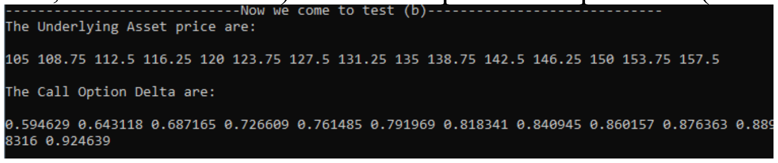
A PART II Option Sensitivities, aka the Greeks

a) Below are the results of this given batch (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)). [output are put/call delta, gamma, vega, and theta]



```
----------------------------
Exact Delta Call: 0.594629
Exact Delta Put: -0.356601
Gamma: 0.0134936
Vega: 26.7781
Theta: -8.39684
```

b) Below are both the input underlying asset price (range: 52.5, interval: 3.75, start value: 105, end value: 157.5) and the output of call/put delta (I use the above given batch data)

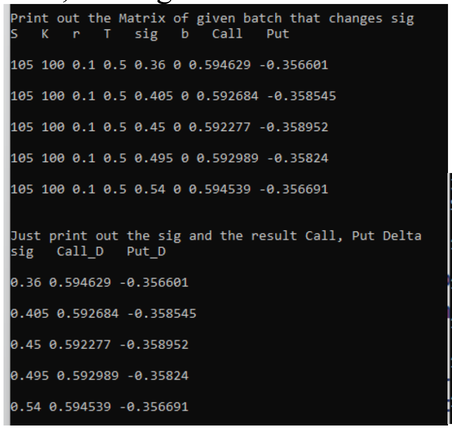


```
----------------------------Now we come to test (b)----------------------------
The Underlying Asset price are:

105 108.75 112.5 116.25 120 123.75 127.5 131.25 135 138.75 142.5 146.25 150 153.75 157.5

The Call Option Delta are:

0.594629 0.643118 0.687165 0.726609 0.761485 0.791969 0.818341 0.840945 0.860157 0.876363 0.889
8316 0.924639
```

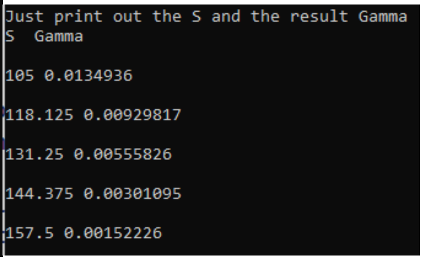
Here, Delta measures how an option's price will react to changes to the underlying stock price (S). Call delta is always positive while put delta is always negative because the call option price moves in the same direction as the stock price while the put option price moves in the opposite.

c) Below are the matrices for each different input (S, K, T, r, sig, b) (I use the given batch data in PARTII) and the output of call/put delta, and gamma

Delta measures how option price will change, while gamma (2nd derivative) measures how delta will change
For Delta, I changed the sig.
For Gamma, I changed the S

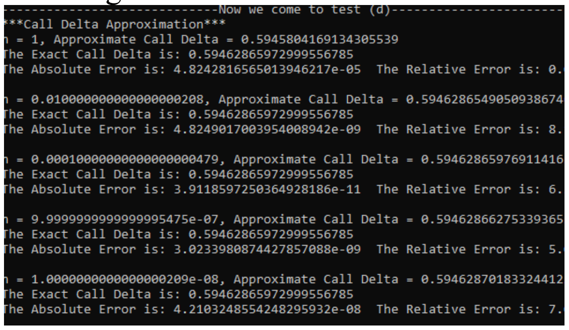


```
Print out the Matrix of given batch that changes sig
S   K   r   T   sig   b   Call   Put

105 100 0.1 0.5 0.36 0 0.594629 -0.356601

105 100 0.1 0.5 0.405 0 0.592684 -0.358545

105 100 0.1 0.5 0.45 0 0.592277 -0.358952

105 100 0.1 0.5 0.495 0 0.592989 -0.35824

105 100 0.1 0.5 0.54 0 0.594539 -0.356691


Just print out the sig and the result Call, Put Delta
sig   Call_D   Put_D

0.36 0.594629 -0.356601

0.405 0.592684 -0.358545

0.45 0.592277 -0.358952

0.495 0.592989 -0.35824

0.54 0.594539 -0.356691
```



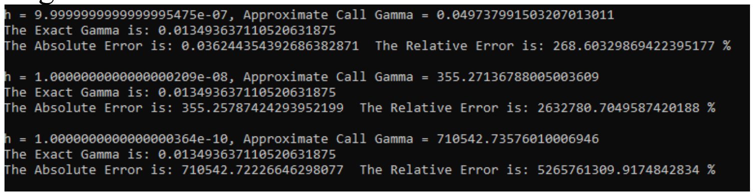
```
Just print out the S and the result Gamma
S  Gamma

105 0.0134936

118.125 0.00929817

131.25 0.00555826

144.375 0.00301095

157.5 0.00152226
```

d) Below we will see the call/put delta, and call gamma using approximation, and the difference (absolute error and relative error) between them and the exact call/put delta and the gamma



```
----------------------------Now we come to test (d)----------------------
***Call Delta Approximation***
h = 1, Approximate Call Delta = 0.5945804169134305539
The Exact Call Delta is: 0.5946286597299556785
The Absolute Error is: 4.8242816565013946217e-05  The Relative Error is: 0.

h = 0.0100000000000000000208, Approximate Call Delta = 0.5946286549050938674
The Exact Call Delta is: 0.5946286597299556785
The Absolute Error is: 4.8249017003954008942e-09  The Relative Error is: 8.

h = 0.000100000000000000000479, Approximate Call Delta = 0.5946286597691141 6
The Exact Call Delta is: 0.5946286597299556785
The Absolute Error is: 3.9118597250364928186e-11  The Relative Error is: 6.

h = 9.9999999999999995475e-07, Approximate Call Delta = 0.5946286627533936 5
The Exact Call Delta is: 0.5946286597299556785
The Absolute Error is: 3.0233980874427857088e-09  The Relative Error is: 5.

h = 1.0000000000000000209e-08, Approximate Call Delta = 0.5946287018332441 2
The Exact Call Delta is: 0.5946286597299556785
The Absolute Error is: 4.2103248554248295932e-08  The Relative Error is: 7.
```

As we can see here, as h decreases from 1 to 0.01, the error between approximated price and the exact price decreased a lot. Also it continuously decreases till h=0.0001 However, when h becomes less than 0.0001, we can see the error starts to become bigger. Smaller h does generally produce a more accurate approximation. However, h cannot be too small. Although the absolute error is still < 0.01 when h is below that, a relatively good h value might be around 0.0001.

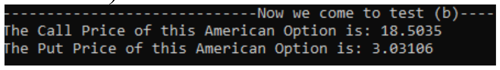


```
h = 9.9999999999999995475e-07, Approximate Call Gamma = 0.049737991503207013011
The Exact Gamma is: 0.013493637110520631875
The Absolute Error is: 0.036244354392686382871  The Relative Error is: 268.60329869422395177 %

h = 1.0000000000000000209e-08, Approximate Call Gamma = 355.27136788005003609
The Exact Gamma is: 0.013493637110520631875
The Absolute Error is: 355.25787424293952199  The Relative Error is: 2632780.7049587420188 %

h = 1.0000000000000000364e-10, Approximate Call Gamma = 710542.73576010006946
The Exact Gamma is: 0.013493637110520631875
The Absolute Error is: 710542.72226646298077  The Relative Error is: 5265761309.9174842834 %
```

In fact, as shown in the result above, when h is smaller than 10^-7, we could see that the error starts to become larger than 0.01, which is bad. So we should keep h within the range of roughly 10^-7 to 1

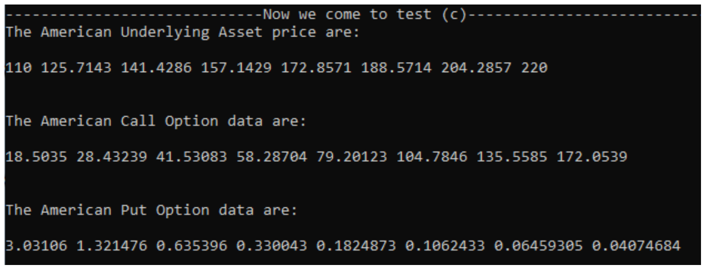
Group B Perpetual American Options

b) The test batch is K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110 (check C = 18.5035, P = 3.03106). Below is the result of the output



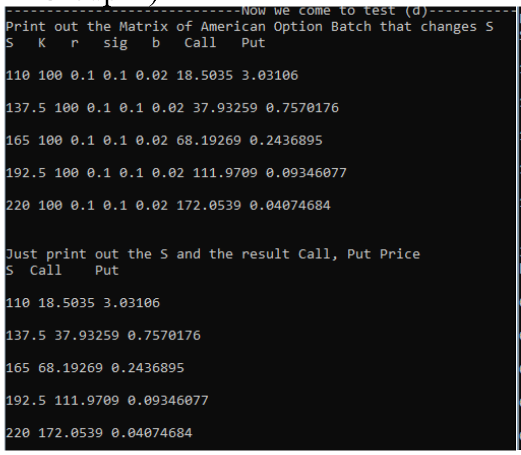
```
----------------------------Now we come to test (b)----
The Call Price of this American Option is: 18.5035
The Put Price of this American Option is: 3.03106
```

c) Below are both the input underlying asset price (range: 110, start value: 110, end value: 220) and the output of call/put price (I use the given batch data in Group B)

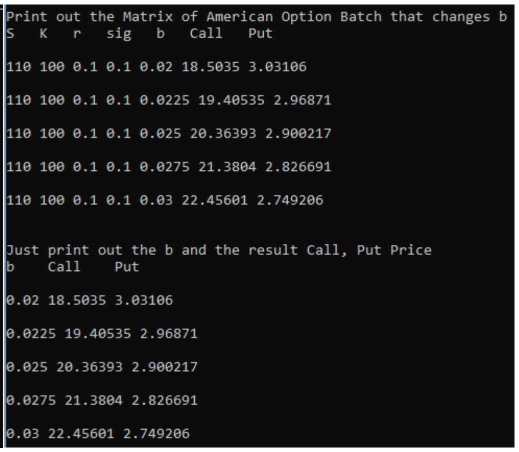


```
----------------------------Now we come to test (c)-------------------------
The American Underlying Asset price are:

110 125.7143 141.4286 157.1429 172.8571 188.5714 204.2857 220


The American Call Option data are:

18.5035 28.43239 41.53083 58.28704 79.20123 104.7846 135.5585 172.0539


The American Put Option data are:

3.03106 1.321476 0.635396 0.330043 0.1824873 0.1062433 0.06459305 0.04074684
```

d) Below are the matrix for each different input (S, K, T, r, sig, b) (I use the given batch data in Group B)



```
---------------------------Now we come to test (d)----------
Print out the Matrix of American Option Batch that changes S
S   K   r   sig   b   Call   Put

110 100 0.1 0.1 0.02 18.5035 3.03106

137.5 100 0.1 0.1 0.02 37.93259 0.7570176

165 100 0.1 0.1 0.02 68.19269 0.2436895

192.5 100 0.1 0.1 0.02 111.9709 0.09346077

220 100 0.1 0.1 0.02 172.0539 0.04074684


Just print out the S and the result Call, Put Price
S  Call    Put

110 18.5035 3.03106

137.5 37.93259 0.7570176

165 68.19269 0.2436895

192.5 111.9709 0.09346077

220 172.0539 0.04074684
```



```
Print out the Matrix of American Option Batch that changes b
S   K   r   sig   b   Call   Put

110 100 0.1 0.1 0.02 18.5035 3.03106

110 100 0.1 0.1 0.0225 19.40535 2.96871

110 100 0.1 0.1 0.025 20.36393 2.900217

110 100 0.1 0.1 0.0275 21.3804 2.826691

110 100 0.1 0.1 0.03 22.45601 2.749206


Just print out the b and the result Call, Put Price
b    Call    Put

0.02 18.5035 3.03106

0.0225 19.40535 2.96871

0.025 20.36393 2.900217

0.0275 21.3804 2.826691

0.03 22.45601 2.749206
```