C++ Level 9 Group A&B

bondxue

December 2, 2017

1 Justifications for design decision

1.1 base Option class

I create base Option class for EuropeanOption and AmericanoOption classes, so that common data and functionality can be put in a base class. In my design, $Price()$ function is common for both classes, and I believe it should be the basic function if more option types involved. Thus I put $Price()$ function in the base class and derived by EuropeanOption and AmericanoOption classes.
Although EuropeanOption and AmericanoOption classes have some common data, since I believe other option types may have different data, to make it generic, I decide not to put these common data in the base class.

1.2 OptionData Structure

I create OptionData struct for both EuropeanOption and AmericanoOption classes, since American option does not have maturity date parameter T. Also I add the stock price S parameter into each OptionData struct so that each batch will be regarded as an instance of OptionData. What's more, I put the OptionData struct outside of the option classes since it will increase the reusability of the struct.

1.3 OptionType enum

For the similar reason as I create OptionData struct, I create option type enum for European and American option separately. Since I create getter and setter for option type in each class, I delete the original $toggle()$ function in the class. In this way, it is much easier to access and modify the type of the option object.

1.4 global function $print()$

Global overloaded $print()$ function is created to print the vector and matrix easily, which can reduce the length of the test file.

1.5 global function $CreateMesh()$

I create a global function that creates a mesh array of doubles with size n. I use the vector size n instead of the mesh size h as the input parameter to avoid the situation that (startpoint - endpoint) / h may not be an integer, which will cause an non uniform size between the second last and last points in the vector.

1.6 Normal cdf and pdf functions

I adopt the normal cdf and pdf functions in the boost library to make my code more clear and efficient.

1.7 Constructor with parameters

In EuropeanOption and AmericanoOption classes, I provide two ways to construct an option with parameters. One is to use the OptionData struct as input, the other is to use option data separately as parameters, which can increase the flexibility of the design.

1.8 $PutCallParity\_Checking()$ function

For the $PutCallParity\_Checking()$ function, instead of checking the left hand side is exactly equal to the right hand side, I adopt a tolerance value $tol = 10^{-6}$. The underlying reason is that since we use approximation to obtain $N(d_1)$ and $N(d_2)$ in the Black-Scholes formula, the call and put prices calculation is approximate, which causes the left side and right side of put-call parity cannot be exactly the same. That's why I add tol to avoid this situation.

1.9 Inline function for setters and getters

I design the setters and getters in EuropeanOption and AmericanoOption classes as default inline functions, which can speed up the execution of short function, since the code of such function will be copied in place instead of calling that function.

1.10 overload functions

I make the $price()$ overloading for
1. regular price calculation using option data
2. price calculation using put-call parity
3. price matrix calculation with parameter matrix input which is very flexible for users to use different input arguments.

Similarity, I make $price\_S()$ overloading as well for
1. price calculation with stock price input
2. price vector calculation with stock price vector input. which is also very flexible.

1.11 Price function with one option parameter

I create price function with one option parameter, such as $Price\_S()$, $Price\_K()$, etc. for both EuropeanOption and AmericanoOption classes, which allows the user only change one specific option parameter of one option object to see the corresponding option price. Since the data type of each option data is the same, i.e., double, I cannot create them as the overload $Price()$ function. They must have their own specific name, which is actually more clear to use.

1.12 $Price()$ function

To use $Price()$ as an overload function with input parameter matrix and output a price matrix, it is noticeable that the input parameter matrix must be organized in a specific way. Otherwise, the output may goes wrong. That is to say the parameter matrix should contain all the parameter vectors in the order of S_vec, K_vec, T_vec, r_vec, sig_vec, b_vec for EuropeanOption object, i.e.,

$$\text{input parameter matrix} = \begin{bmatrix} S_1 & S_2 & S_3 & \cdots \\ K_1 & K_2 & K_3 & \cdots \\ T_1 & T_2 & T_3 & \cdots \\ r_1 & r_2 & r_3 & \cdots \\ sig_1 & sig_2 & sig_3 & \cdots \\ b_1 & b_2 & b_3 & \cdots \end{bmatrix} \tag{1}$$

Therefore, each element in the input matrix will be the variate for the European option, while the rest of the parameters for the option will remain the same, then we can obtain one updated option price for each element in the input matrix. Each price will be obtained by using the price function with one option parameter introduced above, e.g., $Price\_S()$, $Price\_K()$. Then the output of the $Price()$ function will be

$$\text{output price matrix} = \begin{bmatrix} P_{S_1} & P_{S_2} & P_{S_3} & \cdots \\ P_{K_1} & P_{K_2} & P_{K_3} & \cdots \\ P_{T_1} & P_{T_2} & P_{T_3} & \cdots \\ P_{r_1} & P_{r_2} & P_{r_3} & \cdots \\ P_{sig_1} & P_{sig_2} & P_{sig_3} & \cdots \\ P_{b_1} & P_{b_2} & P_{b_3} & \cdots \end{bmatrix}. \tag{2}$$

Similar method I apply to obtain the the AmericanOption price matrix and EuropeanOption Delta matrix. It is noticeable that for American option, the row of maturity date T_vec does not exist.

1.13 pricing function and greeks

I create the European and American pricing functions and Greeks outside of the option classes as global functions, then include them in the option classes. The reason I create them as global functions is it will be more flexible to reuse by others.

1.14 Finite Difference Approximation function $Delta\_f()$ and $Gamma\_f()$

I put $Delta\_f()$ and $Gamma\_f()$ inside of the source file of EuroOption class, not like other price and Greek functions I create in separate header files is due to that I find it is much efficient to design $Delta\_f()$ and $Gamma\_f()$ functions if I apply the $Price\_S()$ function to calculate Δ and Γ directly.

1.15 namespaces

To avoid name conflicts and make classes well-organized, I put
1. base class Option into namespace: OPTION
2. European option related class and functions (EuropeanOption class, EuropeanOption-Pricing functions, EuropeanOptionGreek functions and Normal functions) into namespace: OPTION::EUROPEANOPTION
3. AmericanOption class, AmericanOption functions into namespace: OPTION::AMERICANOPTION
However, CreateMesh() and print() functions are put outside of any namespace to increase the reusability for other purposes.

2 Answers to the questions

2.1 Exact solutions of One-Factor Plain Options

Problem a. The call and put prices using the Batch 1 to Batch 4 data sets are shown as follows, which are the same as C and P provided.

```
==================Group A Part I (a)======================
Batch 1: C1 = 2.13337, P1 = 5.84628
Batch 2: C2 = 7.96557, P2 = 7.96557
Batch 3: C3 = 0.204058, P3 = 4.07326
Batch 4: C4 = 92.1757, P4 = 1.2475
```

Fig. 1: call and put prices using batch data

Problem b. Using Method 1: calculate the call/put price for a corresponding put/call price, the results are show as follows.

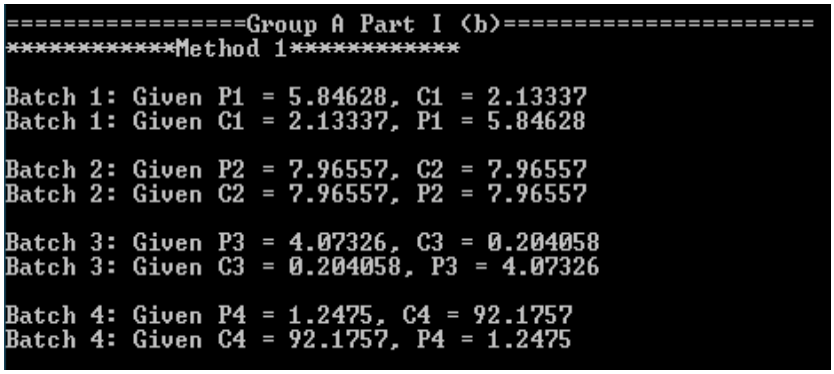

Fig. 2: put-call parity checking using Method 1

Using Method 2: Given a set of put/call prices, check whether it satisfies parity. The results are shown as follows.

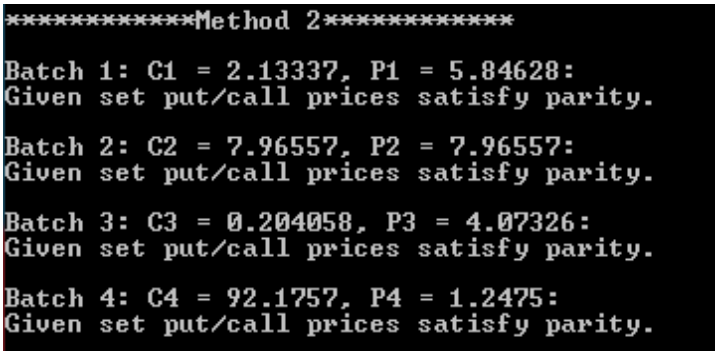

Fig. 3: put-call parity checking using Method 2

Problem c. Using Batch 1 data, given a monotonically increasing stock price vector *S_vec* as input, the corresponding call and put option price vectors are shown as follows.

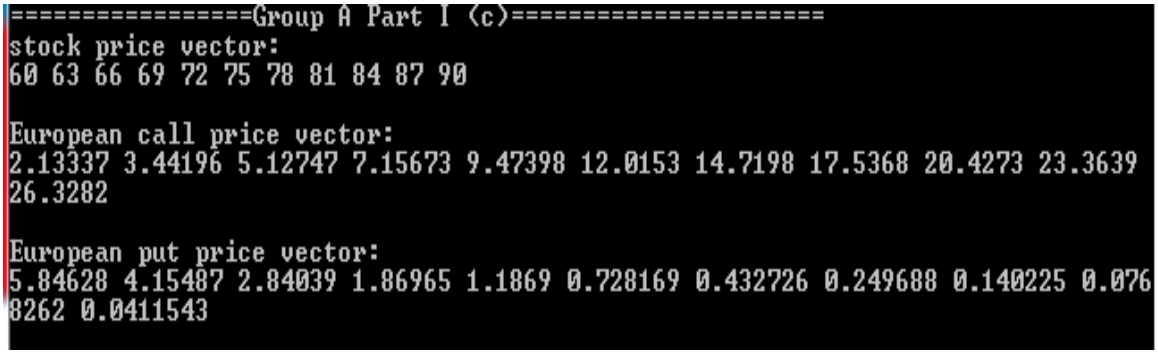

Fig. 4: input a stock price vector and output a option price vector

We can see that when stock price S monotonically increases, the corresponding call option price will also monotonically increase. However, the corresponding put option price will monotonically decrease, approaching to 0.

Problem d. Using Batch 1 data, given a matrix of option parameters as input, then output a option price matrix. The mechanism is shown in Section 1.12. The input matrix is structured as Eq. (1), and the output matrix is structured as Eq. (2). The results are shown as follows.

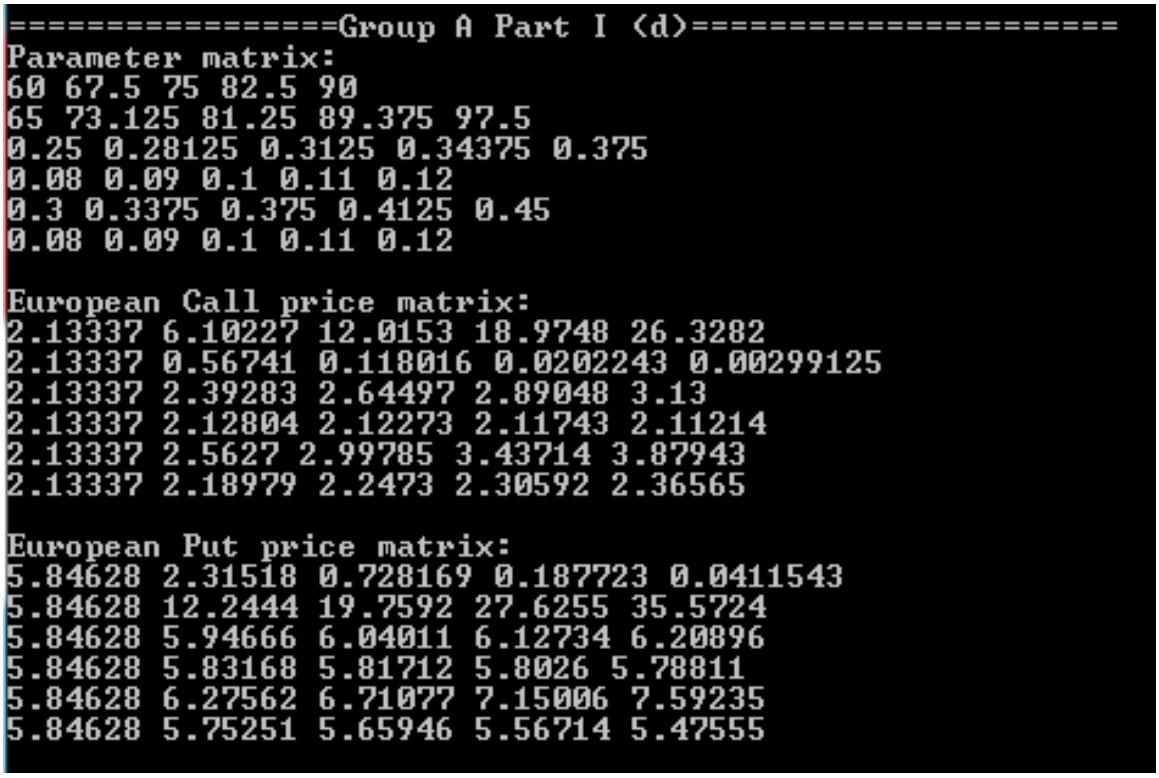

Fig. 5: input option parameter matrix and output option price matrix

We can see that
- when S increases, call price increases and put price decreases
- when K increases, call price decreases and put price increases
- when T increases, call price increases and put price increases
- when r increases, call price deceases and put price decreases
- when sig increases, call price increases and put price increases
- when b increases, call price increases and put price decreases

2.2 Option Sensitivities, aka the Greeks

Problem a. Implement Batch 5, the results are shown as follows.

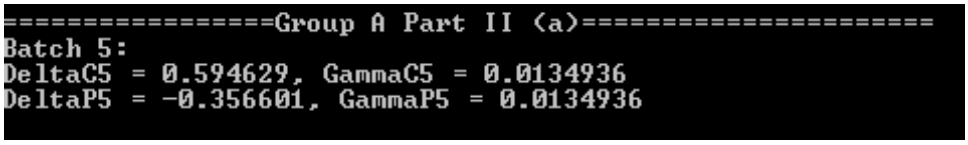

Fig. 6: Delta and Gamma of call and put options using Batch 5 data

Problem b. With a monotonically increasing stock price vector, the corresponding call Delta vector is shown as follows.

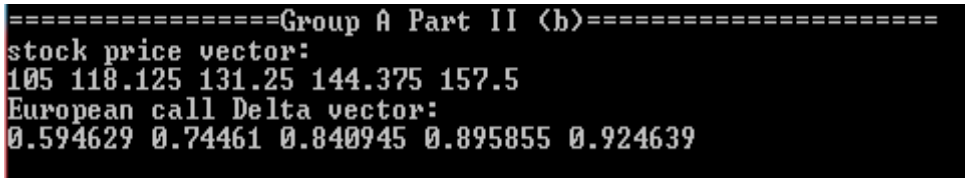

Fig. 7: call Delta vector with stock price vector input

We can see that with the increasing stock price, the call Delta will also increase.

Problem c. The input parameter matrix structure is the same as Eq. (1), the output call Delta matrix has the structure as

$$\text{output Delta matrix} = \begin{bmatrix} \Delta_{S_1} & \Delta_{S_2} & \Delta_{S_3} & \cdots \\ \Delta_{K_1} & \Delta_{K_2} & \Delta_{K_3} & \cdots \\ \Delta_{T_1} & \Delta_{T_2} & \Delta_{T_3} & \cdots \\ \Delta_{r_1} & \Delta_{r_2} & \Delta_{r_3} & \cdots \\ \Delta_{sig_1} & \Delta_{sig_2} & \Delta_{sig_3} & \cdots \\ \Delta_{b_1} & \Delta_{b_2} & \Delta_{b_3} & \cdots \end{bmatrix} . \tag{3}$$

The results are shown as follows.

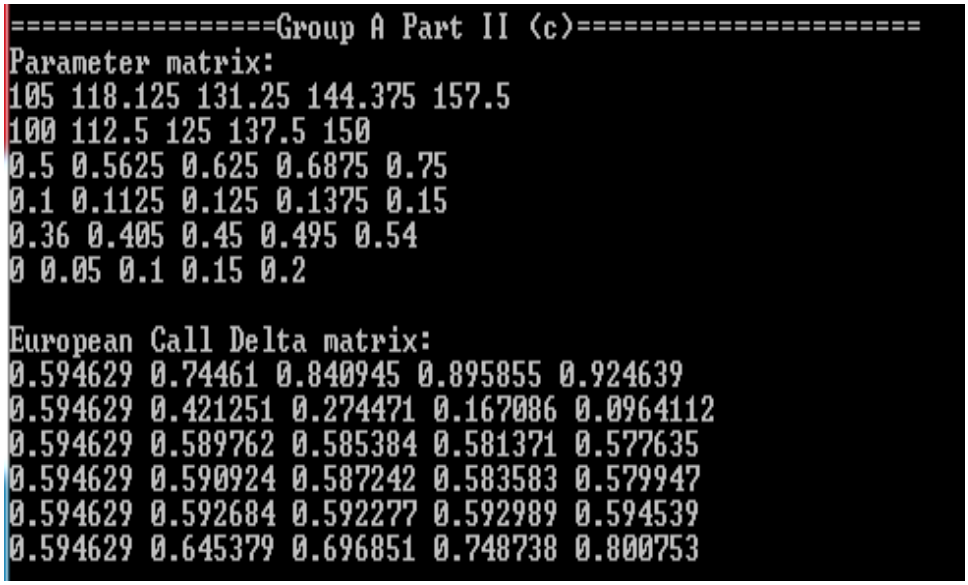

Fig. 8: call Delta matrix with option parameter matrix

We can see that
- when S increases, call Delta increases.
- when K increases, call Delta decreases.
- when T increases, call Delta decreases.
- when r increases, call Delta decreases.
- when sig increases, call Delta decreases then increases.
- when b increases, call Delta increases.

Problem d. The results are shown as follows.

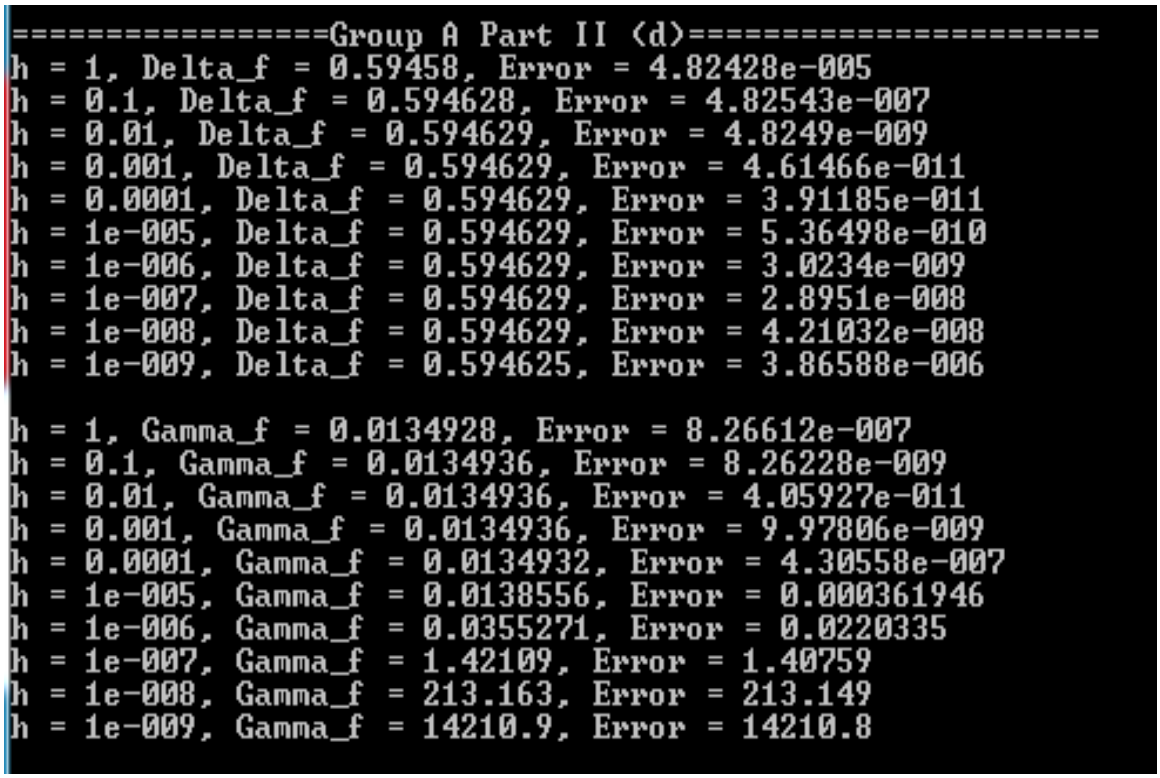

Fig. 9: finite difference approximation and the corresponding errors of Delta and Gamma of call option for Batch 5

For Δ_f, the approximation become more precise when h decreases, until $h = 0.0001$. The best approximation is within about 10^{-10} of Δ_{BS}. However, for values of h smaller than 0.0001, the finite difference approximations deteriorates very quickly.
For Γ_f, when $h \leq 10^{-7}$, the value of Γ_f increases dramatically. The finite difference approximations of Γ_f become more precise while h decreases to 0.01, but is much worse after that. The best approximation is within 10^{-10} of Γ_{BS}.
The underlying reason is that we compute $\Delta_{BS} = e^{(b-r)T}N(d_1)$ and FDM approximation using

$$\Delta_f = \frac{C(S+h) - C(S-h)}{2h} .$$

It uses the numerical approximation of $N(d_1)$ and $N(d_2)$ from the boost library which is accurate within a certain point. When h becomes too small, it will cause the round-off errors to calculate $C(S+h)$ and $C(S-h)$, then Δ_f will become less accurate. Similar problem will happen on Γ_f, since

$$\Gamma_f = \frac{V(S+h) - 2V(S) + V(S-h)}{h^2} .$$

When h is too small, the calculation of $V(S+h)$ and $V(S-h)$ will face the round-off errors, which causes the less accurate Γ_f we obtain.

2.3 Perpetual American Options

Problem b. Testing Batch 6 data, the results are as follows.

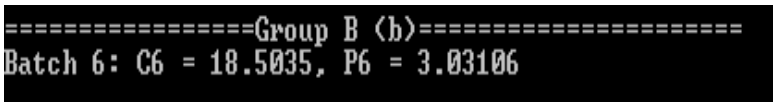

Fig. 10: American call and put prices using Batch 6

Problem c. With increasing stock price vector as input, the output American call and put price vectors are shown as follows.

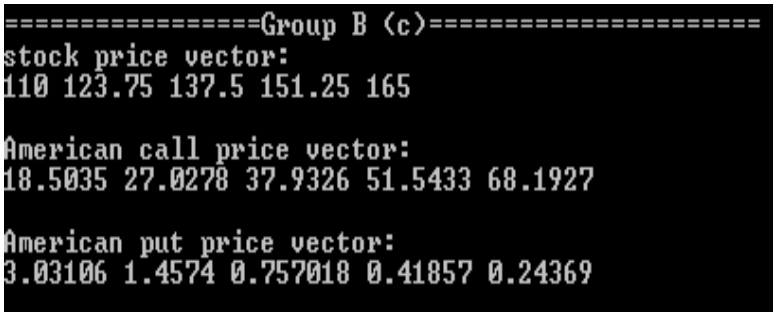

Fig. 11: American call and put price output vector with input stock price vector using Batch 6

We can see that with increasing stock price, the American call price increases, but American put price first increases then decreases.

Problem d. The input option parameter matrix is

$$\text{input parameter matrix} = \begin{bmatrix} S_1 & S_2 & S_3 & \cdots \\ K_1 & K_2 & K_3 & \cdots \\ r_1 & r_2 & r_3 & \cdots \\ sig_1 & sig_2 & sig_3 & \cdots \\ b_1 & b_2 & b_3 & \cdots \end{bmatrix} \tag{4}$$

The output American option price matrix is

$$\text{output price matrix} = \begin{bmatrix} P_{S_1} & P_{S_2} & P_{S_3} & \cdots \\ P_{K_1} & P_{K_2} & P_{K_3} & \cdots \\ P_{r_1} & P_{r_2} & P_{r_3} & \cdots \\ P_{sig_1} & P_{sig_2} & P_{sig_3} & \cdots \\ P_{b_1} & P_{b_2} & P_{b_3} & \cdots \end{bmatrix} . \tag{5}$$

The results are shown as follows.

```
=================Group B (d)======================
Parameter matrix:
110 123.75 137.5 151.25 165
100 112.5 125 137.5 150
0.1 0.105 0.11 0.115 0.12
0.1 0.1125 0.125 0.1375 0.15
0.02 0.0225 0.025 0.0275 0.03

American Call price matrix:
18.5035 27.0278 37.9326 51.5433 68.1927
18.5035 14.2511 11.2825 9.13352 7.53115
18.5035 17.9874 17.5205 17.0958 16.7079
18.5035 19.483 20.4798 21.4886 22.5052
18.5035 19.4053 20.3639 21.3804 22.456

American Put price matrix:
3.03106 1.4574 0.757018 0.41857 0.24369
3.03106 7.09189 15.1703 30.1803 56.5514
3.03106 2.95472 2.88254 2.81414 2.74921
3.03106 3.93722 4.88329 5.85746 6.85143
3.03106 2.79587 2.57809 2.37684 2.1912

Press any key to continue . . .
```

Fig. 12: American call and put price outpu matrix with input option parameter matrix using Batch 6