

MFE C++ - Assignment 5

Due 4pm Jan 20, 2022 Pacific Standard Time

This assignment is worth 30 points

Please upload your code for the assignment as a MULTIPLE C++ files (option.h and option.cpp) and also include a screenshot of the console output after running the program.

Building an Option Class Hierarchy

We are going to return to European options (calls and puts) and build a class hierarchy of options. In the next assignment (Assignment 6), we will extend this hierarchy.

In this assignment, you are to create two files: option.h and option.cpp. In **option.h**, include all your class declarations. In **option.cpp**, include the implementation of ALL your class methods. See Lecture 6 for an explanation of how to split files into header and cpp files and how to move method definitions/implementations outside of a class declaration.

Again: YOU MUST SPLIT YOUR CLASSES INTO DECLARATIONS AND DEFINITIONS/IMPLEMENTATIONS. PUT ALL METHOD IMPLEMENTATIONS (UNLESS OTHERWISE NOTED) IN **options.cpp**

Create a Project with Multiple files

1. Download assignment5.cpp from bCourses. You will need this file to run the test code. No changes are required for this file and you do not need to turn it in.
2. Create an **options.h** header file. Put your class declarations here. See Lecture 6 for how to use #ifndef/#define/#endif to 'wrap' your declarations so that they are only included once.
3. Create an **options.cpp** file where you will place your class method implementations. You may also want to grab the 'helper' functions from Assignment 3 (normalCDF, etc) and place them in options.cpp for use with your code.

You will need to include other header files (like vector & algorithm) to get your code to work. Some may need to be included in the header file so that your project compiles without error.

For this assignment, please make all class properties **protected** and all class methods **public**.

Option class

1. Create an abstract class named Option
2. Add the following Properties:
 - a. double Strike – the strike for the option
 - b. double Sigma – the annual volatility of the option
 - c. double T – the time to expiry of the option
 - d. double RiskFreeRate – the risk free rate associated with the option
3. Create a constructor for the Option class that has the following signature and initializes the class's properties: `Option(double strike, double sigma, double t, double riskFreeRate)`
4. Create the following **pure virtual** methods:
 - a. `virtual double getValue(double spot) = 0;` Derived classes should return the value of the option for the given spot value.
 - b. `virtual double getIntrinsicValue(double spot) = 0;` Derived classes should return the intrinsic (exercise) value of the option for the given spot.
 - c. `virtual double calcBlackScholesValue(double spot)=0;` Derived classes should return the Black Scholes value for the option.
5. Create the following (non-virtual) method:

`double getDelta(double spot)` - this method should return the difference between the value of the option with a 1% increase in spot and the value of the option with the given spot. This is a common way that traders assess the “delta” Greek value of an option.

EuropeanOption class

1. Create an abstract class named `EuropeanOption` that is publicly derived from `Option`
2. Add a constructor to `EuropeanOption` that has a similar signature to the one in `Option` and that calls the `Option` constructor in its initialization list.
3. Implement the `getValue()` method by returning the value returned by `calcBlackScholesValue()`. That is, the European option's value is the Black Schole's value. You won't implement `calcBlackScholesValue` just yet.

EuropeanCall class

1. Create a class named `EuropeanCall` that is publicly derived from `EuropeanOption`
2. Add a constructor to `EuropeanCall` that has a similar signature to the one in `Option` and that calls the `EuropeanOption` constructor in its initialization list.
3. Implement the method `getIntrinsicValue(double spot)` that returns $\max(\text{spot} - \text{Strike}, 0)$
4. Implement the `calcBlackScholesValue()` method that returns the Black Scholes value for a European call. Refer to the assignment 3 solution and Lecture 4 for the Black Scholes formula and a possible implementation.

EuropeanPut class

1. Create a class named `EuropeanPut` that is publicly derived from `EuropeanOption`
2. Add a constructor to `EuropeanPut` that has a similar signature to the one in `Option` and that calls the `EuropeanOption` constructor in its initialization list.
3. Implement the method `getIntrinsicValue(double spot)` that returns $\max(\text{Strike} - \text{spot}, 0)$
4. Implement the `calcBlackScholesValue()` method that returns the Black Scholes value for a European put. Refer to Lecture 4 for the Black Scholes formula.

Position Class

Implement a Position class. It is ok to keep all the implementation for this class in the header file.

```
class Position {
public:
    double weight;
    Option* option;

    Position() : weight(0), option(NULL) {}
    Position(double w, Option* o) : weight(w), option(o) {}
};
```

Portfolio Class

1. Create a Portfolio class. This class will hold a **vector** of Positions which represents a portfolio of options on a single security.
2. Add a protected **vector<Position> Positions** property
3. Create a default constructor for Portfolio
4. Add a **void addPosition(const Position& pos)** method which adds the pos parameter to the Positions vector
5. Add a **double getValue(double spot)** which calculates the weighted sum of option values for the portfolio: $value = \sum weight_i * option_i \rightarrow getValue(spot)$
6. Add a **double getDelta(double spot)** which calculates the weighted sum of option delta values for the portfolio: $value = \sum weight_i * option_i \rightarrow getDelta(spot)$

Add Risk Metrics to Portfolio class

We are going to add two measurements of risk to the portfolio class, Value-at-Risk (VaR) and Expected Shortfall. We are going to use a similar technique for both measurements.

We will simulate the daily movement of the underlying equity and create a vector holding the portfolio value for the simulated daily move in the spot. Here is the basic recipe:

```
dt = 1.0/252.0 // there are roughly 252 trading days per year
vol = sqrt(dt) * sigma
drift = dt*(riskFreeRate - 0.5*vol*vol)

for the specified number of iterations N:
    calculate a new spot value as:
        spot_new = spot * exp(drift + vol*standardNormalRandomVariable)
    calculate the portfolio value for spot_new
    store the portfolio value in a vector called vals

sort the vector in ascending order using sort from the standard
library
```

$$VaR = \text{Current Portfolio Value} - \text{vals}[0.05 * N - 1]$$
$$\text{Expected Shortfall} = \text{Current Portfolio Value} - \frac{1}{0.05 * N} \sum_{i=0}^{0.05*N-1} \text{vals}[i]$$

A couple of notes:

1. $0.05*N$ is the 5th percentile of the total number of simulations. So, if N is 100, $0.05*N = 5$
2. Our version of VaR is looking at the 5th percentile expected outcome (1-sided). In other words, 95% of the time, our expected outcome will have a lower loss than our VaR
3. Our version of Expected Shortfall is 'integrating' of the bottom 5% of the simulated portfolio returns (technically, simulated changes in portfolio value). So, expected shortfall tells us, on average, what the expected loss in the lower tail will be on a daily basis.
4. Note that VaR and ES are reported as *positive* numbers even though they represent losses.

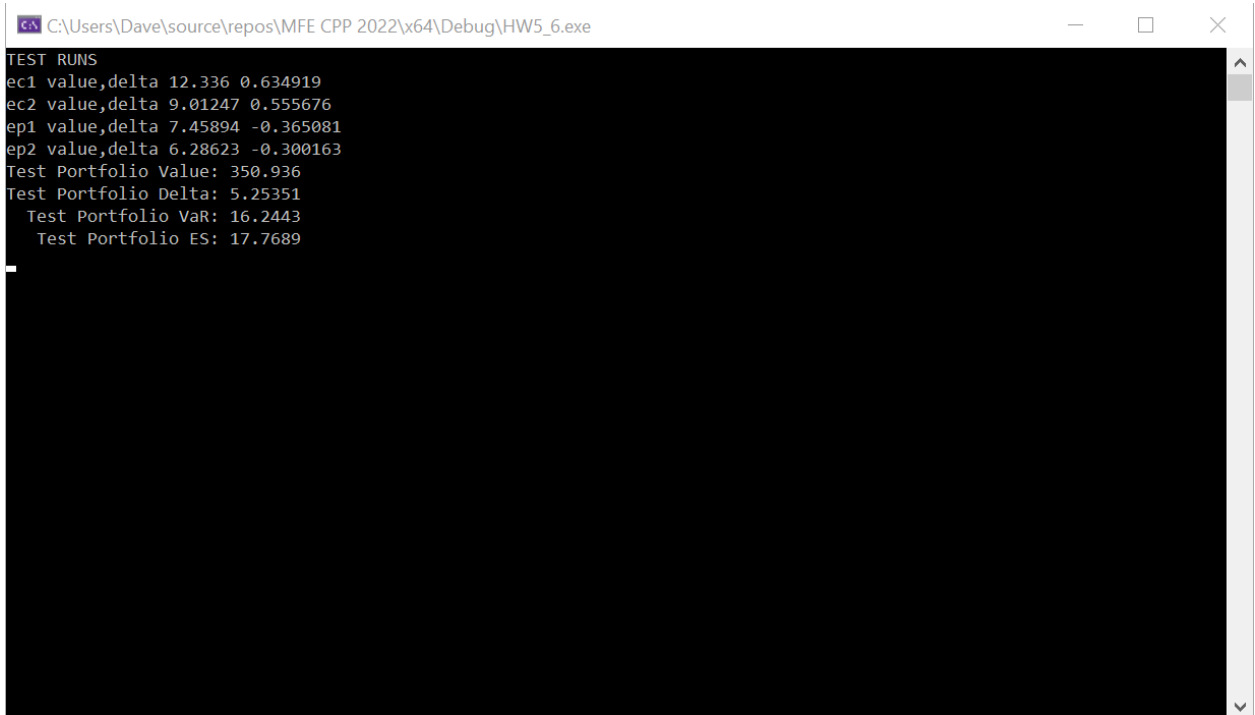
Your Risk Tasks

1. Add a `double calcDailyVaR(double spot, double sigma, double riskFreeRate, int N)` method to the Portfolio class which follows the methodology above and calculates VaR according to the formula. Note that sigma & riskFreeRate will not necessarily match the sigma & risk free rates for the options in the portfolio. N is the number of simulated spot values to create.
2. Add a `double calcDailyExpectedShortfall(double spot, double sigma, double riskFreeRate, int N)` method to the Portfolio class which follows the methodology above and

calculates Expected Shortfall. according to the formula. Note that sigma & riskFreeRate will not necessarily match the sigma & risk free rates for the options in the portfolio. N is the number of simulated spot values to create.

Run Your Program and Take a Screenshot of the Results

A screenshot of the “test” part of the program is included below



```
C:\Users\Dave\source\repos\MFE CPP 2022\x64\Debug\HW5_6.exe
TEST RUNS
ec1 value,delta 12.336 0.634919
ec2 value,delta 9.01247 0.555676
ep1 value,delta 7.45894 -0.365081
ep2 value,delta 6.28623 -0.300163
Test Portfolio Value: 350.936
Test Portfolio Delta: 5.25351
Test Portfolio VaR: 16.2443
Test Portfolio ES: 17.7689
```

Submit your code (options.h and options.cpp) along with a screenshot of your program execution.