MFE C++ - Assignment 6

Due 4pm Jan 27, 2022 Pacific Standard Time

Jan 18 - Binomial tree pseudocode updated

This assignment is worth 30 points

Please upload your code for the assignment as a MULTIPLE C++ files (options.h, options.cpp and assignment6.cpp) and also include a screenshot of the console output after running the program.

Building an Option Class Hierarchy - Continued

We return to Assignment 5's European option hierarchy and add in both American options and a European Knockout call option. To accomplish this, we will expand our hierarchy and add in handling of binomial trees.

You can either expand your code from Assignment 5 or base your answer on the code published in the solution to Assignment 6.

IMPORTANT: Your binomial tree implementation will be handled in the Option class and not in derived classes.

In Assignment 5, we put our method definitions in options.cpp. It is ok to keep them in options.h for Assignment 6. Your call.

Create a Project with Multiple files

1. Download assignment6.cpp from bCourses. You will need this file to run the test code. You will also need to alter the file for Q1, Q2 and Q3 below.
2. Place your **options.h** and **options.cpp** from Assignment 5 in the project or transcribe the code from the published solution.

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.

Option class

1. Starting with the Option class from Assignment 5, add the following **protected** *pure virtual* method: `virtual double getCurrentExerciseValue(double spot, double t) = 0;`

2. Add the following **protected** virtual method: `virtual double calcBinomialNodeValue(double spot, double t, double time_val)`

   The default implementation should return the value returns by **getCurrentExerciseValue** if that value is greater than time_val. Otherwise, it should return time_val. The knockout option class defined below will override this method.

3. Add the following **public** method:

   ```
   double calcBinomialTreeValue(double spot, int treeDepth)
   ```

   The method should use the following pseudocode to implement a binomial tree:

   ```
   dt = T/treeDepth
   u = exp(Sigma * sqrt(dt))
   d = 1/u
   p = (exp(RiskFreeRate) - d)/(u-d)

   declare vector called vals
   # setup the initial tree node values
   for i = 0 to treeDepth:
      s = spot * u ^ (2*i-treeDepth)
      add getCurrentExerciseValue(s, T) to end of vals

   # walk the tree back thru time
   t = T
   for j = treeDepth-1 down to 0:
      t = t - dt
      for i = 0 to j:
         s = spot * u^(2*i-j)
         v_time = exp(-RiskFreeRate*dt)*(p*vals[i+1] + (1-p)*vals[i])
         v = calcBinomialNodeValue(s, t, v_time)
         vals[i] = v
   return vals[0]
   ```

   Note that all for loops are *inclusive*. For example, for (int i = 0; i <= treeDepth; i++) for the first loop. We use the virtual method calcBinomialNodeValue to set tree node values as we "walk backward" through time. European options and American options use the same implementation of calcBinomialNodeValue and the knockout option uses a different one. The difference between American and European options is enforced within the **getCurrentExerciseValue** method.

EuropeanOption class

1. Add an implementation of `double getCurrentExerciseValue(double spot, double t)`

   If t == T, then the method should return the value of getIntrinsicValue(spot). Otherwise, it should return 0 (since European options can only be exercised at expiry).

AmericanOption class

1. Create a class named AmericanOption that is publicly derived from Option

2. Add a constructor to AmericanOption that has a similar signature to the one in Option and that calls the Option constructor in its initialization list.

2. Implement the getValue(spot) method by returning the value returned by `calcBinomialTreeValue` (spot, 500). That is, we use a binomial tree of depth 500 to calculate our option value.
3. Add an implementation of `double getCurrentExerciseValue(double spot, double t)`

   That returns the value of getIntrinsicValue(spot)..

AmericanCall class

1. Create a class named AmericanCall that is publicly derived from AmericanOption

2. Add a constructor to AmericanCall that has a similar signature to the one in Option and that calls the AmericanOption constructor in its initialization list.

3. Implement the method getIntrinsicValue(double spot) that returns max(spot – 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.

AmericanPut class

1. Create a class named AmericanPut that is publicly derived from AmericanOption

2. Add a constructor to AmericanPut that has a similar signature to the one in Option and that calls the AmericanOption constructor in its initialization list.

3. Implement the method getIntrinsicValue(double spot) that returns max(Strike - 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.

NOTE: AmericanCall & AmericanPut basically will look identical to EuropeanCall & EuropeanPut but derive from AmericanOption rather than EuropeanOption.

EuropeanCallKnockout

1. Create a class named EuropeanCallKnockout that is publicly derived from EuropeanCall

   This class is going to implement a call option that is "knocked out" if the spot goes above the barrier. Refer to Lecture 9 for information about

2. Add a **protected** double property called **Barrier**

3. Add a constructor to EuropeanCallKnockout that has the following signature:
   ```
   EuropeanCallKnockout(double strike, double barrier, double sigma, double expiry,
   double riskFreeRate)
   ```

   a. The constructor should call the EuropeanCall constructor and also set the Barrier property.

4. Implement the method getIntrinsicValue(double spot) that returns max(spot – Strike, 0) if spot < Barrier and 0 if spot > Barrier

5. Implement the getValue(spot) method by returning the value returned by `calcBinomialTreeValue` (spot, 500). That is, we use a binomial tree of depth 500 to calculate our option value.

6. Implement the calcBinomialNodeValue method. If spot >= Barrier, return 0. Otherwise return Option::calcBinomialNodeValue(spot, t, time_val).

Additions to main() in assignment6.cpp

Add code to the end of main() to answer the following questions.:

Q1: What is the minimum tree depth required such that ap1.calcBinomialTreeValue(100, depth) is within 0.01 of ap1.getValue(100)?

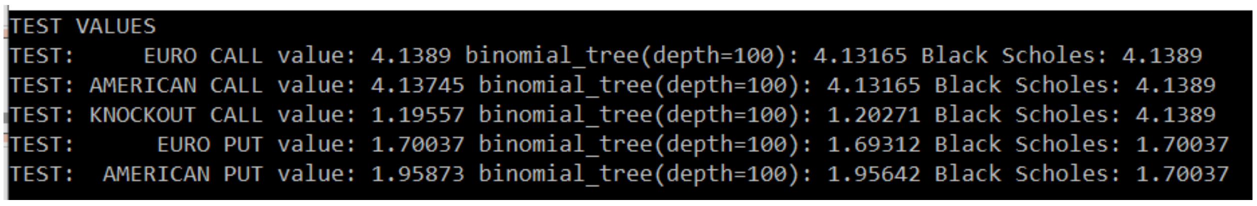
Q2: For spot values between 50 & 150, what is the biggest difference between ap1.getValue(spot) – ac1.getValue(spot)? What is the spot that has the biggest difference in value? Use increments of $1 to do your analysis.

Q3: What is the maximum spot such that ecko1.getValue(spot) >= 0.8 * ec1.getValue(spot) ?

Refer to the code to understand the variable names (ap1, ec1, etc). You can search for answers in whatever way you see fit.

Run Your Program and Take a Screenshot of the Results

A screenshot of the "test" part of the program is included below



```
TEST VALUES
TEST:     EURO CALL value: 4.1389 binomial_tree(depth=100): 4.13165 Black Scholes: 4.1389
TEST: AMERICAN CALL value: 4.13745 binomial_tree(depth=100): 4.13165 Black Scholes: 4.1389
TEST: KNOCKOUT CALL value: 1.19557 binomial_tree(depth=100): 1.20271 Black Scholes: 4.1389
TEST:      EURO PUT value: 1.70037 binomial_tree(depth=100): 1.69312 Black Scholes: 1.70037
TEST:  AMERICAN PUT value: 1.95873 binomial_tree(depth=100): 1.95642 Black Scholes: 1.70037
```

Submit your code (options.h, options.cpp and assignment6.cpp) along with a screenshot of your program execution and your answers to Q1, Q2 and Q3.