IB9JHO Programming Assignment

In this assignment you will write C++ code which uses the CRR binomial method to value vanilla European/American options for an arbitrary underlying asset with a fixed annual dividend yield. You will implement the binomial method using two different algorithms which should give the same result, allowing you to verify you have implemented them correctly. In addition, you will verify that your calculated solution approaches the analytical solutions for European options given by Black-Scholes formalism if the time intervals are sufficiently small.

You may use any standard library data structures/functions and any language features up to C++17 that you wish. However, do not use any third-party libraries which are not part of the standard. Your code should be clearly formatted and annotated with comments where appropriate. A short report (a few pages at most) should accompany your code which explains the problem, the methodology you followed, and an assessment of the limitations of your code (details below). You have some flexibility in how the code is implemented and tested, but your solution should contain the class structure and functions specified in this document so that it can easily be tested. **It is very important that your final submission only contains the following files, and these files can be compiled without needing modification. Marks will be deducted if this is the case:**

- OptionValueCalculator.hpp
- OptionValueCalculator.cpp

- CallOptionValueCalculator.hpp
- CallOptionValueCalculator.cpp

- PutOptionValueCalculator.hpp
- PutOptionValueCalculator.cpp

- BinomialOptionPricer.hpp
- BinomialOptionPricer.cpp

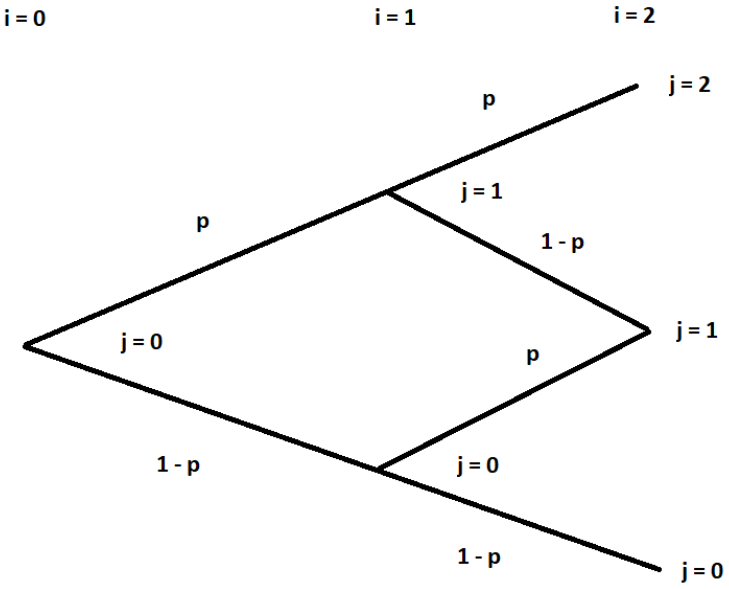
- report.pdf

Forward Recursion method (European options):

This method starts from timestep zero and accumulates the option value from the next timestep using the recurrence relation:

(1) $$V_i^j = e^{-r\nabla t}(pV_{i+1}^{j+1} + (1-p)V_{i+1}^j)$$

Where V_i^j is the option value, *r* is the risk-free interest rate, ∇t is the time increment between each timestep, *p* is the risk-neutral probability of an upwards jump, *i* is the timestep/tree depth of the current node, and *j* is height of the current node. This can be visualised in the figure below for a tree of depth 2:



The recurrence relation can be implemented by utilising recursion in C++. Note that you will need to keep track of *i* and *j* in your recursive function calls. The recursion terminates at the base of the binomial tree where *i* is equal to the depth of the tree (*i = 2* in the above figure). For a tree of depth *n*, the values at the base of the tree are the values of the option at the expiry time, and can be computed as follows:

(2) $$V_n^j = \max\left(S_0 u^{2j-n} - k, 0\right)$$ for call options

(3) $$V_n^j = \max\left(k - S_0 u^{2j-n}, 0\right)$$ for put options

Where S_0 is the asset price at time zero (*i = 0*) and *k* is the strike price.

Forward Recursion method (American options):

The difference in the calculation for American options in comparison to European options is that they can be exercised by the contract buyer before the expiry time. This means that the exercise value E_i^j needs to be calculated at each node:

(4) $E_i^j = \max(S_0 u^{2j-i} - k, 0)$ for call options

(5) $E_i^j = \max(k - S_0 u^{2j-i}, 0)$ for put options

The recurrence relation then becomes.

(6) $V_i^j = \max(e^{-r\nabla t}(pV_{i+1}^{j+1} + (1-p)V_{i+1}^j), E_i^j)$

The rest of the algorithm is identical to the version for European options.

Backward Induction method:

The backward induction method starts from the last timestep and computes the option value at all the base nodes using equation 2 (calls) or 3 (puts). Next, it uses the recurrence relation from equation 1 (European options) or equation 6 (American options) to compute the value of all the nodes at the previous timestep. This process is repeated until time zero. You will need to use a two layered loop. For a tree of depth *n*, the first loop is over timesteps from *i = n – 1* down to *i = 0*. The inner loop is from node height *j* = 0 to *j = i*.

Note that you will need to use an array to store the values at each timestep. However, you only need one array, as the values calculated at timestep *i* replace the values calculated at timestep *i + 1*. In particular, for each *j*, V_i^j replaces V_{i+1}^j in the array.

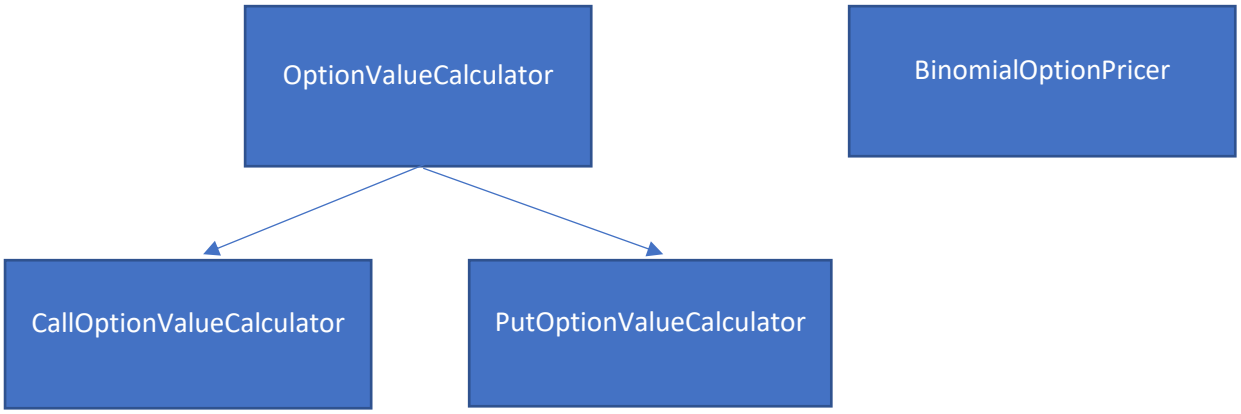
Black Scholes (European options):

For European options, you should find that using larger trees/more timesteps cause your binomial algorithm to approach the analytical solution for the fair value of the option in the risk-free framework imposed by the Black-Scholes model. The analytical solutions for calls and puts are omitted here as they are widely known and easy to find. However, to implement the Black Scholes solution to test against your code, you will require a define a function to calculate the CDF of the normal distribution $N(x)$. In C++ this can be done by using the *erfc()* library function with the following transform:

$$N(x) = \frac{erfc(-\frac{x}{\sqrt{2}})}{2}$$

Classes:

Your final submission is **required** to contain the following class structure: and functions so that your solution can be tested:



Where the Blue arrows indicate inheritance. You will not be marked on the efficiency (runtime speed) your code runs, but accuracy is very important. Your solution should be thoroughly tested in all the different cases.

- BinomialOptionPricer – Contains all the implementation for valuing options.
- OptionValueCalculator – Abstract class with a virtual member function that calculates the value of a vanilla option at expiry time given the price of the underlying asset at expiry time.
- CallOptionValueCalculator – Overrides the virtual member function in OptionValueCalculator so that the function values call options. **The strike price for the option is stored in the object.**
- PutOptionValueCalculator – Overrides the virtual member function in OptionValueCalculator so that the function values put options. **The strike price for the option is stored in the object.**

Member Functions for BinomialOptionPricer:

The BinomialOptionPricer class must contain the following member functions:

- `BinomialOptionPricer(unsigned int max_depth, double growth_factor, double growth_probability);`

- `double BinomialOptionPricer::price_vanilla_option_european_recursion(unsigned int depth, double delta_t, double S0, double r, const OptionValueCalculator& value_calc)`

- `double BinomialOptionPricer::price_vanilla_option_american_recursion(unsigned int depth, double delta_t, double S0, double r, const OptionValueCalculator& value_calc)`

- `double BinomialOptionPricer::price_vanilla_option_european(unsigned int depth, double delta_t, double S0, double r, const OptionValueCalculator& value_calc)`

- `double BinomialOptionPricer::price_vanilla_option_american(unsigned int depth, double delta_t, double S0, double r, const OptionValueCalculator& value_calc)`

The functions are described as follows:

- BinomialOptionPricer – constructor for the option pricing class.

- price_vanilla_option_european_recursion – price a European option using the forward-recursion method.

- price_vanilla_option_american_recursion – price an American option using the forward-recursion method.

- price_vanilla_option_european - price a European option using the backward-induction method.

- price_vanilla_option_american - price an American option using the backward-induction method.

The parameters for the above functions are explained below in the order they appear:

- max_depth - the absolute maximum depth of the binomial tree which is created when pricing options using the BinomialOptionPricer object. This can be stored in the object and used as a common-sense check that the requested depth when valuing options is not too large.

- growth_factor - the factor that the underlying asset grows by in an upwards jump.

- growth_probability – the risk-neutral probability of an upwards jump.

- depth – the depth (number of timesteps) of the binomial tree to construct when pricing an option.

- delta_t – the amount of time between each timestep in the tree.

- S0 – the initial price of the underlying asset as timestep zero.

- r – the risk-free interest rate.

- value_calc – this should be used by the function to calculate the value of the option at expiry time by utilising polymorphism.

General Advice:

- When testing/debugging, start with a tree depth of 2 so that you can verify the values in your code by hand and identify where things are going wrong. Then try with bigger trees.

- Start by getting European call options right for both the forward recursion method and the backward induction method. Then it is relatively straightforward to extend this to put options, and finally American call/put options.

- A key thing to remember is that you have two different methods to calculate the same thing. Given the same inputs, the outputs should be identical within machine precision. This is a good thing, as if the two different sources agree for a few cases, it verifies that you likely implemented the algorithms correctly.

- Do not take it for granted that your code will work in all four cases, even if it appears to work in one case. The only way to know for sure that things work is to test them.

- You may want to write your tests for BinomialOptionPricer in its own class to keep things organised, this is a common practice and a good habit to start building.

- It is important to start working on the assignment early, so you have time to ask questions through the forum or the support classes if you get stuck or need things clarified. You are encouraged to communicate with others, but do not share actual coded solutions and copy from each other (it is usually easy to tell if you did so do not risk it).

Report:

The report should be concisely written and relatively short. A good report will be well-presented show insight into the task which was set. A rough structure for the report could be as follows:

- Introduce the problem.

- Briefly discuss the methodology followed to solve the problem. You do not have to reproduce all the equations, just briefly outline the theory you used and the steps you followed to get the code working.

- Perform some analysis of your solution, good things to focus on here would be: how big should you make your trees to ensure that your solution is accurate? How well do both algorithm's scale with tree size and why? How do the analytical Black Scholes solutions compare to the numerical solutions? A few figures/tables would be useful here.

- Summarise what you learned from the programming project and your subsequent analysis in a few sentences.