IB9JHO Programming Assignment 1

In this assignment you will write C++ code to value vanilla European/American options using the CRR binomial method for an underlying asset with a fixed annual dividend yield.

You will implement the CRR method using two different algorithms which should give the same result, helping you to verify you have implemented them correctly. Then you will investigate the conditions that your calculated solutions approach the analytical solutions for vanilla options given by Black-Scholes formalism. The final task is a coding challenge where your solution will be benchmarked against the other members of the cohort.

You may use any standard C++20 library data structures/functions in your solution, but you should not use any third-party libraries. Your code should be clearly formatted and annotated with detailed comments. You have some flexibility in how the code is implemented and tested, but your solution should contain the file structure and functions specified in this document and build on the starter github repository which is provided. **It is important to follow the submission instructions below:**

1. Don't add any extra files to the repository apart from report.pdf in the main directory used for answering tasks 3 and 4.

2. You should only need to modify the files below for the assignment. Don't modify any other files.
    a. src/option_pricing_functions.cpp
    b. src/main.cpp
    c. src/option_pricing_convergence.cpp
    d. tests/benchmark_option_pricing_functions.cpp
    e. tests/test_exercise_price_calculator.cpp
    f. tests/test_option_pricing_functions.cpp

3. When you are ready to submit you should first push all your code to the assignment repository as done in the labs.

4. Once you have pushed your final submission. In the repository on github, go to Code -> Download Zip and submit the zip file through the module page. The repository should match the zip file you submit exactly. Be sure to check this.

Black Scholes (European options):

Using larger trees/more timesteps in the binomial algorithm causes the result to approach the analytical solution for the value of the option in the risk-free framework imposed by the Black-Scholes model. You should implement the function price_vanilla_option_bs to help you check this. The analytical solutions for calls and puts can be found in the lecture slides. The Black Scholes solution requires one to calculate the normal cumulative distribution function $N(x)$. In C++ this is usually implemented by using the *erfc()* library function with the following transform:

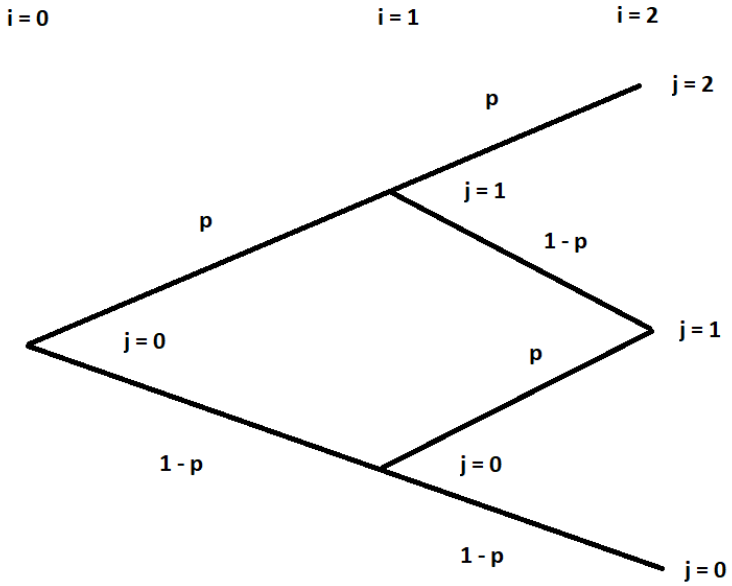
$$N(x) = \frac{erfc(-\frac{x}{\sqrt{2}})}{2}$$

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, $\nabla\ t$ 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 a recursive function 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*), *u* is the up-jump factor and *k* is the strike price. Note that the tree is recombinant (*u * d = 1*) for down-jump factor *d*, so only the up-jump factor is needed to calculate the spread at expiry time.

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

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

The recurrence relation then becomes:

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

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 in an array 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 use a nested loop with two layers. For a tree of depth *n*, the outer loop is over timesteps from $i = n - 1$ down to $i = 0$. The inner loop is from node height $j = 0$ to $j = i$. You need to use an array to store the values at each timestep. A single one-dimensional array of size $n + 1$ is sufficient to price an option with a tree of depth *n*. This is because the values calculated at timestep *i* will replace the values calculated at timestep $i + 1$. That is, for all $j < i$ at timestep *i*, V_i^j will replace V_{i+1}^j in the array.

Task 1. Implementing the Algorithms:

The assignment requires you to complete the functions listed below so they work as described**.** You are free to add any additional functions you require in option_pricing_functions.c. **Do NOT modify the function signatures or names. The input variables must stay the same. Moreover, do NOT modify the header file option_pricing_functions.h.**

- ```
  void price_vanilla_option_european_bs(double S0, double r, double volatility,
  double strike_price, double dividend_yield, double expiration_time, double*
  call_price, double* put_price)
  ```

- ```
  double price_vanilla_option_european_recursion(unsigned int i, unsigned int j,
  unsigned int depth, double S0, double r,
  double volatility, double strike_price, double dividend_yield, double
  expiration_time, option_fxn exercise_profit_calculator)
  ```

- ```
  double price_vanilla_option_american_recursion(unsigned int i, unsigned int j,
  unsigned int depth, double S0, double r,
  double volatility, double strike_price, double dividend_yield, double
  expiration_time, option_fxn exercise_profit_calculator)
  ```

- ```
  double price_vanilla_option_european_induction(
  unsigned int depth, double S0, double r,
  double volatility, double strike_price, double dividend_yield,
  double expiration_time, option_fxn exercise_profit_calculator)
  ```

- ```
  double price_vanilla_option_american_induction(
  unsigned int depth, double S0, double r,
  double volatility, double strike_price, double dividend_yield,
  double expiration_time, option_fxn exercise_profit_calculator)
  ```

The functions should be implemented as follows:
  ```

- price_vanilla_option_european_bs – calculate both call and put prices for European options using the Black-Scholes analytical solution. The outputs are stored in the call_price and put_price variables
- 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_induction - price a European option using the backward-induction method.

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

Task 2. Testing:

Using only the functions exposed in the header file option_pricing_functions.h, demonstrate that the algorithms from task 1 are correctly implemented by writing test cases in test_option_pricing_functions.cpp. You should verify that the recursive and inductive methods have the same outputs (within machine precision) with a range of different input values. You can also manually calculate some cases for small tree sizes to compare your code to.

Task 3. Benchmarking (answer in report.pdf):

Generate benchmarking results for the two algorithms using the code in benchmark_option_pricing_functions.cpp. You can edit the file as required to examine the performance at different depths and run the benchmarks through the testing tab in vscode. **Write a short summary explaining your observations and why they occur in the report.**

Task 4. Convergence (answer in report.pdf):

Investigate the convergence behaviour of the inductive algorithm. Namely, the relationship between tree depth and agreement with theoretical Black Scholes solutions. Generate convergence results by editing option_pricing_convergence.cpp as required, and then running the executable build/generate_convergence_output. This will generate printed output to convergence_output/output.csv in a form that can be imported to excel/python for visualisation.

Write a short summary of your investigation in the report. Include figures you generated from the convergence data.

Some possible talking points are:

- What tree depth should we use to get an accuracy of 4 significant figures?
- How should we estimate the accuracy of an American option without an analytical solution to Black Scholes?

Task 5. Benchmarking challenge:

After all other tasks are complete there are two other functions to implement in option_pricing_functions.cpp:

- `std::vector<double> price_vanilla_option_european_induction_call(double S0, double r, std::vector<double>& volatility_range, double strike_price, double dividend_yield, double expiration_time);`

should price European call options for any range of strictly increasing volatilities.

- `std::vector<double> price_vanilla_option_european_induction_call(double S0, double r, std::vector<double>& volatility_range, std::vector<double>& strike_price_range, double dividend_yield, double expiration_time);`

should price European call options for any range of strictly increasing volatilities and strike prices. The length of the output vector will be `volatility_range.size()` x `strike_price_range.size()` with prices[j * `volatility_range.size()` + i] corresponding to the i'th strike price and the j'th volatility.

The challenge for both functions is to get the fastest function that returns the correct results for a range of 10 volatilities and 10 x 10 volatilities and strikes respectively. You should include the output of a benchmark to show your final result.

Hints:

- The depth is not given as an input to the function. You should decide the depth based on the parameters or set it to a fixed value.
- Use the functions defined in the previous tasks to help you write test cases.
- Start with the simplest implementation possible (using the functions from the previous task), then improve it step-by-step, regularly running your test cases to check the results are still correct. You will not gain much credit for this task if the function is found to give incorrect results.

General Advice:

- When testing/debugging, start with a tree depth of 1 so that you can verify the values in your code by hand (or an excel sheet) and identify errors more easily.

- Start with European options as they are simpler. Move on to American options when you are convinced over several test cases the European options are correct.

- You have two different methods (recursion and induction) to calculate the same thing. Given the same inputs, the outputs should be identical within machine precision if you implemented the code correctly. The analytical solution is useful for checking accuracy with respect to the true solution, but less useful for checking that the code implementation is

precise. This is because the error from slow numerical convergence hides programming mistakes.

- Do not assume that your code will work in all cases, even if it appears to work in one case. The only way to know for sure that is to have good test coverage of a range of cases.

- It is important to start working on the assignment early, so you have time to ask questions if you get stuck or need something clarified.