C++ for Finance - The Project

J. PU

2025

1. The deadline for the project is set to December 22th 2025 at 11:00 AM.
2. Changes on teams will not be allowed from October 26th 2025 at 11:00 AM.
3. A .zip file containing your project is to be sent to your TD teacher by e-mail, you must CC all your teammates.
4. The file must contain only .cpp and .h files, its size must not exceed 1MB.
5. The subject of the e-mail must contain "C++ for Finance project 2025".
6. The zipped has to be attached (not hosted on a server), the file has to be named after the surname of your team's representative;
   The team's representative is defined as the student with the first surname in the alphabetical order, excluding: DERASSE, HANNA GUERGUIS and PICARD.
7. If there is a main() function in your project, it has to be removed (our own file with *main()* will be used for the grading process).
8. No additional public members should be created

Part I

Black-Scholes Model (2-3h)

1 Model specification

A European vanilla option has the following characteristics:

- Type: Call or Put (to be modelled with an *enum*)
- Strike price: K
- Expiry date: T

Its price depends on the following market data:

- Underlying price: S
- Interest rate: r

The following parameter is also required in order to price the option:

- Volatility: σ

2 Implementation

1. Implement the abstract class *Option:*
   - with a private member *double _expiry*, along with a getter method *getExpiry()*
   - with a pure virtual method *double payoff(double)*, *payoff()* represents the payoff function of the option, denoted by h in this document
   - write a constructor that initialize *_expiry* with an argument
2. Derive *Option* into another abstract class *EuropeanVanillaOption*:
   - with private attributes *double _strike*
   - write a constructor which initialize *_expiry* and *_strike* with arguments (call the base constructor)
   - the constructor should ensure that the arguments are nonnegative
   - write a **classe enum** *optionType* that has two values: *call* and *put*
   - write an pure virtual method *GetOptionType()* which should return an *optionType* enum
3. Derive *EuropeanVanillaOption* into two classes: *CallOption* and *PutOption*.
   - They should use the constructor of *EuropeanVanillaOption*
   - For a Call option with strike K, the payoff is given by $h(z) = \begin{cases} z - K & \text{if } z \geq K \\ 0 & \text{otherwise.} \end{cases}$
   - For a Put option with strike K, the payoff is given by $h(z) = \begin{cases} K - z & \text{if } K \geq z \\ 0 & \text{otherwise.} \end{cases}$
   - Override the *GetOptionType()* accordingly in the derived classes
4. Create the class *BlackScholesPricer*

- With constructor *BlackScholesPricer(EuropeanVanillaOption* option, double asset_price, double interest_rate, double volatility)*
- Declare *BlackScholesPricer* as a friend class of *EuropeanVanillaOption* in order for the former to access the strike of the latter
- Write the *operator()* which returns the price of the option. The Black-Scholes formula can be found on the internet. (Hint: use *std::erfc*.)
- Write the method *delta()* which returns the Delta of the option

Part II

The Cox-Ross-Rubinstein model (3h)

3 The CRR model

In the CRR model the price of an asset evolves in discrete time steps ($n = 0, 1, 2, \cdots$). Randomly, it can move up by a factor $1+U$ or down by $1+D$ independently at each time step, starting from the spot price S_0 (see Figure below).

$$\begin{tikzcd}
 & & & & \cdots \\
 & & & i=3 \arrow[ur] \arrow[dr] & \\
 & & i=2 \arrow[ur] \arrow[dr] & & \cdots \\
 & i=1 \arrow[ur] \arrow[dr] & & i=2 \arrow[ur] \arrow[dr] & \\
i=0 \arrow[ur] \arrow[dr] & & i=1 \arrow[ur] \arrow[dr] & & \cdots \\
 & i=0 \arrow[ur] \arrow[dr] & & i=1 \arrow[ur] \arrow[dr] & \\
 & & i=0 \arrow[ur] \arrow[dr] & & \cdots \\
 & & & i=0 \arrow[ur] \arrow[dr] & \\
 & & & & \cdots \\
n=0 & n=1 & n=2 & n=3 & \cdots
\end{tikzcd}$$

Figure 1: Binary Tree

As a result, the stock price at step n and node i is:

$$S(n,i) \quad = \quad S_0 (1+U)^i (1+D)^{n-i},$$

where $S_0 > 0, U > D > -1$ and $0 \leq i \leq n$. There is also a risk-free asset which grows by the factor $1 + R > 0$ at each time step (starting at 1 at step 0).

The model admits no arbitrage iif $D < R < U$.

In the CRR model the price $H(n,i)$ at time step n and node i of a **European option** with expiry date N and payoff $h\left(S\left(N\right)\right)$ can be computed using the CRR procedure, which proceeds by backward induction:

- At the expiry date N:

  $$H\left(N,i\right) = h\left(S\left(N,i\right)\right)$$

  for each node $i = 0, \cdots, N$.

- If $H\left(n+1,i\right)$ is already known for all nodes $i = 0, \cdots, n+1$ for some $n = 0, \cdots, N-1$, then

  $$H\left(n,i\right) = \frac{qH\left(n+1,i+1\right) + \left(1-q\right)H\left(n+1,i\right)}{1+R}$$

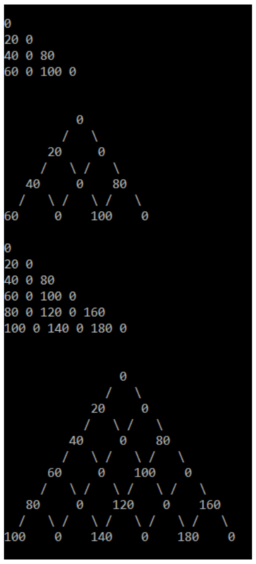
  for each $i = 0, \cdots, n$; and where q is defined by

  $$q = \frac{R-D}{U-D}$$

  is called the **risk-neutral probability**.

4 Implementation

1. Implement a class *BinaryTree* that represents the data structure (path tree) used for the CRR method:
    - It should be a template class *BinaryTree<T>*
    - It should have a member *_depth*, representing N
    - It should contain a private member *_tree*, a vector of vectors (STL) to hold data of type T
    - Implement the setter method *setDepth(int)* a setter for *_depth*, that resizes *_tree* and allocate/deallocate properly the vectors in *tree*
    - Implement the setter method *setNode(int, int, T)* which sets the value stored in *_tree* at the given indices
    - Implement the getter method *getNode(int, int)* which retrives the corresponding value
    - Implement the method *display()* which prints the all the values stored



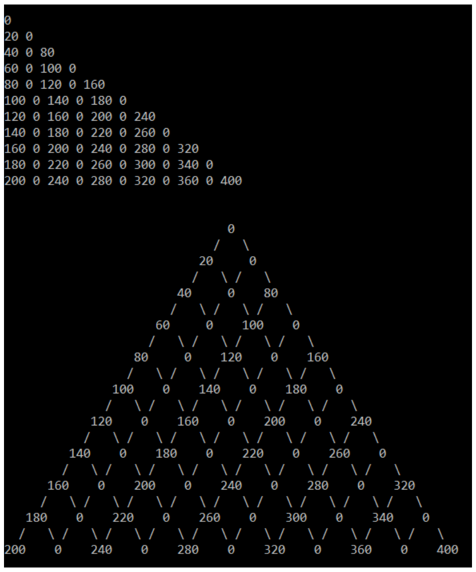
```
0
20 0
40 0 80
60 0 100 0


          0
        /   \
      20     0
     /   \ /   \
   40     0    80
  /   \ /   \ /   \
60     0    100    0

0
20 0
40 0 80
60 0 100 0
80 0 120 0 160
100 0 140 0 180 0


                0
              /   \
            20     0
           /   \ /   \
         40     0    80
        /   \ /   \ /   \
      60     0    100    0
     /   \ /   \ /   \ /   \
   80     0    120    0    160
  /   \ /   \ /   \ /   \ /   \
100    0    140    0    180    0
```

(a) Examples with N=3 and N=5



```
0
20 0
40 0 80
60 0 100 0
80 0 120 0 160
100 0 140 0 180 0
120 0 160 0 200 0 240
140 0 180 0 220 0 260 0
160 0 200 0 240 0 280 0 320
180 0 220 0 260 0 300 0 340 0
200 0 240 0 280 0 320 0 360 0 400


                               0
                             /   \
                           20     0
                          /   \ /   \
                        40     0    80
                       /   \ /   \ /   \
                     60     0    100    0
                    /   \ /   \ /   \ /   \
                  80     0    120    0    160
                 /   \ /   \ /   \ /   \ /   \
               100    0    140    0    180    0
              /   \ /   \ /   \ /   \ /   \ /   \
            120    0    160    0    200    0    240
           /   \ /   \ /   \ /   \ /   \ /   \ /   \
         140    0    180    0    220    0    260    0
        /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \
      160    0    200    0    240    0    280    0    320
     /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \
   180    0    220    0    260    0    300    0    340    0
  /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \ /   \
200    0    240    0    280    0    320    0    360    0    400
```

(b) Examples with N=10

Figure 2: Examples of output by the *display()* function

2. Create the class CRRPricer

   - With constructor *CRRPricer(Option* option, int depth, double asset_price, double up, double down, double interest_rate)*
     - *depth*: N
     - *asset_price*: S_0
     - *up*, *down*, *interest_rate*: U, D, R respectively
   - In the constructor, check for arbitrage
   - Create the tree structure to store the tree of the desired depth (hint: use *BinaryTree* with an appropriate type)
   - Write the method *void compute()* that implements the CRR procedure
   - Write the getter method *get(int, int)* that returns $H(n, i)$.
   - Write the *operator()* which returns the price of the option, it must call *compute()* if needed
   - The CRR method provides also a closed-form formula for option pricing:

     $$H(0,0) = \frac{1}{(1+R)^N} \sum_{i=0}^{N} \frac{N!}{i!(N-i)!} q^i (1-q)^{N-i} h\left(S\left(N,i\right)\right).$$

     Put an optional argument *bool closed_form* that defaults to *false* to the *operator()*. When it is set to true, the above formula should be used instead of the CRR procedure.

3. Similarly to *EuropeanVanillaOption*, design *EuropeanDigitalOption* and its derived classes (*EuropeanDigitalCallOption* and *EuropeanDigitalPutOption*) in order to take into account the following type of options:

   - Digital Call with payoff: $h(z) = 1_{z \geq K}$
   - Digital Put with payoff: $h(z) = 1_{z \leq K}$
   - Enable *BlackScholesPricer* to price digital options as well (closed form formulas also exist for Black-Scholes prices and deltas for digital options); you can overload *BlackScholesPricer*'s constructor.

Part III

Path dependent options and MC (3h)

5 Some option pricing theory

5.1 European options and path-dependent option

We consider a risky asset with the Black-Scholes dynamics:

$$S_t = S_0 e^{\left(r-\frac{\sigma^2}{2}\right)t+\sigma W_t},$$

where $\sigma \in \mathbb{R}^+$ is the volatility and W_t a Wiener process under the risk neutral probability $\mathbb{Q}$.
We denote the price (at time 0) of an option by H_0.
This price can be determined by computing the expected discounted payoff under $\mathbb{Q}$:

$$H_0 = e^{-rT}\mathbb{E}^{\mathbb{Q}}\left[H_T\right],$$

where H_T denotes the payoff of the option at its expiry date T.

5.1.1 European options

In the case of a European option, $H_T = h(S_T)$, where $h : \mathbb{R}^+ \to \mathbb{R}$ is the payoff function of the option, it only depends on the price of the risky asset at maturity.

5.1.2 Path dependent options

For more complex options, the payoff H_T also depends on the price of the risky asset at dates prior to the maturity.
These are called path dependent options.
Let $t_k = \frac{k}{m}T$, for $k = 1, \cdots, m$. A path-dependent option is a financial derivative with payoff at expiry date T:

$$H_T = h\left(S_{t_1}, \cdots, S_{t_m}\right),$$

where $h : \left(\mathbb{R}^+\right)^m \to \mathbb{R}$ is the payoff function.
For instance, the arithmetic Asian Call has the following payoff function:

$$h\left(z_1, \cdots, z_m\right) = \left(\left(\frac{1}{m}\sum_{k=1}^{m} z_k\right) - K\right)^+.$$

5.2 Black-Scholes random paths

The Wiener process W has independent increments, with $W_t - W_s \sim \mathcal{N}\left(0, t-s\right)$ for $0 \leq s < t$.
S_{t_k} can be expressed as

$$S_{t_k} = S_{t_{k-1}} e^{\left(r-\frac{\sigma^2}{2}\right)(t_k-t_{k-1})+\sigma\sqrt{t_k-t_{k-1}}Z_k}$$

Where $Z_1, \cdots, Z_m$ are i.i.d. random variables with distribution $\mathcal{N}(0,1)$.
Let the seqence $\widehat{Z}_1, \cdots, \widehat{Z}_m$ be a i.i.d. sample of $Z_1, \cdots, Z_m$. We refer the sequence $\left(\widehat{S}_{t_1}, \cdots, \widehat{S}_{t_m}\right)$ defined by:

$$\begin{aligned}
\widehat{S}_{t_1} &= S_0 e^{\left(r-\dfrac{\sigma^2}{2}\right)t_1+\sigma\sqrt{t_1}\widehat{Z}_1}, \\
\widehat{S}_{t_k} &= \widehat{S}_{t_{k-1}} e^{\left(r-\dfrac{\sigma^2}{2}\right)(t_k-t_{k-1})+\sigma\sqrt{t_k-t_{k-1}}\widehat{Z}_k}, \text{ for } k = 2, \cdots, m,
\end{aligned}$$

as a Black-Scholes sample path.

5.3 Monte Carlo

Let $\left(\widehat{S}^i_{t_1}, \cdots, \widehat{S}^i_{t_m}\right)$, for $i \in \mathbb{N}$, be a sequence of independent sample paths. By the law of large numbers

$$\mathbb{E}^{\mathbb{Q}}\left[h(S_{t_1}, \cdots, S_{t_m})\right] = \lim_{N\to\infty} \frac{1}{N} \sum_{i=0}^{N-1} h\left(\widehat{S}^i_{t_1}, \cdots, \widehat{S}^i_{t_m}\right).$$

This means that for sufficient large N, we can approximate H_0 using

$$H_0 \approx e^{-rT} \frac{1}{N} \sum_{i=0}^{N-1} h\left(\widehat{S}^i_{t_1}, \cdots, \widehat{S}^i_{t_m}\right)$$

6 Programming

- Augment the *Option* class with *payoffPath* method, taking a *std::vector<double>* as argument, returning $h(S_{t_1}, \cdots, S_{t_m})$.
- The non-overriden version of this function should return $h\left(S_{t_m}\right)$ (calling *payoff(double)*)
- Create a derived class from *Option*: *AsianOption*
  - The constructor takes a *std::vector<double>* as argument, representing $(t_1, \cdots, t_m)$
  - The argument should be stored in a private member, with a getter method *getTimeSteps()*
  - Override *AsianOption::payoffPath(std::vector<double>)* so that

$$h(S_{t_1}, \cdots, S_{t_m}) = h\left(\frac{1}{m}\sum_{k=1}^{m} S_{t_k}\right),$$

    where h on the right hand side is *payoff(double)*. *AsianOption::payoffPath(std::vector<double>)* should **not** be virtual.
- Created *AsianCallOption* and *AsianPutOption*, derived from *AsianOption*.
  - In addition to *std::vector<double>*, their constructor takes a double as argument, defining the strike.
  - They have to have proper implementations of *payoff()*.
- Augment the *Option* class with *bool isAsianOption()*, returning *false* in its non-overriden version, override it in *AsianOption*.
- In *CRRPricer*'s constructor, check if the option is an Asian option, if it is the case, throw an exception.
- Design a singleton class *MT*, encapsulating a *std::mt19937* object. Two public static methods are implemented: *double rand_unif()* and *double rand_norm()*, returning a realization of $\mathcal{U}([0,1])$ and $\mathcal{N}(0,1)$ respectively. Ensure that only one instance of *std::mt19937* can be used in all the program through *MT*.
- Write the *BlackScholesMCPricer* class:
  - The constructor must have signature *(Option* option, double initial_price, double interest_rate, double volatility)*
  - The class should have a private attribute that counts the number of paths already generated **since the beginning of the program** to estimate the price of the option, a getter named *getNbPaths()* needs to give a read access to this attribute.
  - The method *generate(int nb_paths)* generates *nb_paths* new paths of $(S_{t_1}, \cdots, S_{t_m})$ (for European Option, $m = 1$), and **UPDATES** the current estimate of the price of the option (the updating process is the same as in exercise 5 of the TD).
  - The operator *()* returns the current estimate (throw an exception if it is undefined).
  - The method *confidenceInterval()* returns the 95% CI of the price, as a *std::vector<double>* containing the lower bound and the upper bound.
  - The random generation have to be handled by calling *MT::rand_norm()*.
  - No path should be stored in the object
  - Check the prices given by *BlackScholesMCPricer* are in line with those given by *BlackScholesPricer* on vanilla options.

Part IV

Back to CRR (1-2h)

7 American option in the binomial model

In addition to pricing European options, we want to include the ability to price American options in the binomial model.

The holder of an American option has the right to exercise it at any time up to and including the expiry date N. If the option is exercised at time step n and node i of the binomial tree, then the holder will receive payoff $h\left(S\left(n,i\right)\right)$.

The price $H\left(n,i\right)$ of an American option at any time step n and node i in the binomial tree can be computed by the following procedure, which proceeds by backward induction on n:

- At the expiry date N: $H\left(N,i\right)$ has the same value as for the option's European counterpart. Financial interpretation: if not exercised before the expiry, there is no advantage holding an American option over holding a European option.

- If $H\left(n+1,i\right)$ is already known at each node $i \in \{0, \cdots, n+1\}$ for some $n < N$, then

$$H\left(n,i\right) = \max\left\{\underbrace{\frac{qH\left(n+1,i+1\right)+\left(1-q\right)H\left(n+1,i\right)}{1+R}}_{\text{continuation value}}, \underbrace{h\left(S\left(n,i\right)\right)}_{\text{intrinsic value}}\right\}$$

  for each $i \in \{0, \cdots, n\}$.
  Financial interpretation: the option holder chooses the maximum between the continuation value (expected gain if they do not exercise, under the risk-neutral measure) and the intrinsic value (the value of the option if exercised immediately).

In particular, $H\left(0,0\right)$ at the root node of the tree is the price of the American option at time 0.

We would like to compute and store the price of an American option for each time step n and node i in the binomial tree. In addition, we want to compute the early exercise policy, which should be of Boolean type and tells if the American option should be exercised or not for each state of the tree. The early exercise policy should also be stored using an instance of *BinaryTree<bool>*.

8 Black-Scholes as limit of the binomial tree

The binomial model can be used to approximate the Black-Scholes model if N is large.

One of the scheme is to divide the time interval $[0, T]$ into N steps of length $h = \frac{T}{N}$, and set the parameters of the binomial model to be:

$$\begin{aligned} U &= e^{\left(r+\frac{\sigma^2}{2}\right)h+\sigma\sqrt{h}} - 1, \\ D &= e^{\left(r+\frac{\sigma^2}{2}\right)h-\sigma\sqrt{h}} - 1, \\ R &= e^{rh} - 1, \end{aligned}$$

where σ is the volatility and r is the continuously compounded interest rate in the Black-Scholes model.

Implement a method to initialize a Binomial tree as a Black-Scholes approximation (using the Black-Scholes parameters). Compare option prices with the Monte Carlo method and the closed form method for European options.

9 Implementation

1. Augment the *Option* class with *bool isAmericanOption()* which returns false in its non-overriden version.
2. Derive *Option* into *AmericanOption*, and override *isAmericanOption()* properly.
3. Derive *AmericanOption* into *AmericanCallOption* and *AmericanPutOption*, write proper constructors and override their respective *payoff()* methods.
4. Modify *CRRPricer* in order for it to price properly American options; the exercise condition for American options is stored in a *BinaryTree<bool>*, accessible through a getter method *bool getExercise(int, int)*.
   The exercise condition is true when the intrinsic value is larger or equal to the continuous value, it is computed during the CRR procedure.
5. Overload the CRRPricer with *CRRPricer(Option* option, int depth, double asset_price, double r, double volatility)*, which initializes U, D and R as described in Section 6.

10 Test your program!

- Use the test files available on DVL containing the grading *main()* functions.